

**הנחיות להגשת תוכנה ברמת הצעויות יתרה לתואר
שני במדעי המחשב**

**Guidelines for Submitting Excellent Software for M.Sc. in
Computer Science**

גרסה 2.0

ד"ר יורם סgal

כל הזכויות שמורות - Dr. Segal Yoram ©

22-11-2025

תוכן העניינים

5	1 חדש בגרסה 2.0	1
5	פרקים חדשים שנוספו	1.1
5	שיפורים נוספים	1.2
5	מטרת התוספות	1.3
7	2 סקירה כללית	2
7	3 מסמכי פרויקט ותוכנו	3
7	מסמך דרישות המוצר	3.1
7	מסמך ארכיטקטורה	3.2
8	4 תיעוד קוד ומבנה פרויקט	4
8	קובץ EMDAER מקיף	4.1
8	מבנה פרויקט מודולרי	4.2
9	איניות קוד והערות	4.3
9	5 ניהול קונפיגורציה ואבטחת מידע	5
10	קבצי קונפיגורציה	5.1
10	אבטחת מידע	5.2
10	6 בדיקות ואיכות תוכנה	6
10	בדיקות יחידה	6.1
11	טיפול במקרי קיצון ותקלות	6.2
11	תוצאות בדיקה צפויות	6.3
11	7 מחקר וניתוח תוצאות	7
11	חקר פרמטרים	7.1
12	מחברת ניתוח תוצאות	7.2
12	הצגה ויוזאלית של תוצאות	7.3
12	8 ממשק משתמש וחווית משתמש	8
12	קריטריוני איניות	8.1
13	תיעוד ממשק	8.2
13	9 ניהול גרסאות ותיעוד פיתוח	9
13	שיטות עבודה מומלצות עם tiG	9.1
13	ספר הפרומפטים	9.2

14	עלויות ותמחור	10
14	ניתוח עלויות	10.1
14	ניהול תקציב	10.2
14	הרחבת ותחזוקתיות	11
14	נקודות הרחבת	11.1
15	תחזוקתיות	11.2
15	תקני איכות בינלאומיים	12
15	מאפייני איכות מוצר	12.1
16	רישימת בדיקה סופית	13
16	בדיקה טכנית מפורטת	13.1
17	מקורות ותקנים נוספים	14
17	ארגון הפרויקט כחברילה	15
17	קובץ הגדרת החבילה	15.1
17	קבצי <code>__init__.py</code>	15.2
17	מבנה תיקיות מאורגן	15.3
18	שימוש בתניבים ייחסימים	15.4
18	רישימת בדיקה: ארגון כחברילה	15.5
18	דוגמה למבנה נכון	15.6
19	עיבוד מקבילי וביצועים	16
19	הבדל בין Multiprocessing ל-Multithreading	16.1
19	שימוש ב-Multiprocessing	16.2
20	שימוש ב-Multithreading	16.3
20	בטיחות חוטים	16.4
20	רישימת בדיקה: עיבוד מקבילי	16.5
21	עיצוב מודולרי ואבני בניין	17
21	מבנה אבן בנייה	17.1
21	עקרונות עיצוב	17.2
22	ולידציה והגנה	17.3
22	רישימת בדיקה: עיצוב אבני בניין	17.4
23	דוגמה לאבן בנייה טובה	17.5
25	הערה חשובה	18
25	הערה משולבת	18.1

1 החדש גרסה 2.0

גרסה 2.0 של מסמך ההנחיות מרחיבת את הקритריונים להגשת תוכנה ברמת מצוינות ומוסיפה שלושה פרקים חדשים המתמקדים בבדיקה טכנית מעמיקה של הקוד. תוספות אלה משלימות את הקритריונים האקדמיים והמחקריים הנוכחיים ומבטיחות הערכה מקיפה של איכות הקוד ברמה הטכנית.

1.1 פרקים חדשים שנוספו

פרק 13: ארגון הפרויקט כחבילת — פרק זה מספק רשימת בדיקה מפורטת לארגון הפרויקט כחבילת (Python package) מקצועית. הפרק כולל קритריונים לקבצי הגדרה (`setup.py` או `pyproject.toml`), מבנה תיקיות מאורגן, קבצי `init.py`, שימוש בתיבות ייחסויים, ניהול תלויות. ארגון נכון כחבילת מאפשר שימוש חוזר בקוד, ניהול תלויות ברור, והפצה והתקנה פשוטה.

פרק 14: עיבוד מקבילי וביצועים — פרק זה עוסק בשימוש נכון במעבדים מרובים (-`multiprocessing`) ובחוטי ביצוע מרובים (multithreading) לשיפור ביצועי המערכת. הפרק מסביר את ההבדלים בין פעולות תובעניות מבחינת מעבד (CPU-bound) לבין פעולות תובעניות מבחינת קלט/פלט (I/O-bound), ומספק רשימות בדיקה מפורטות לכל אחת מהגישות, כולל ניהול משאבים ובתיות חוטים.

פרק 15: עיצוב מודולרי ואבני בניין — פרק זה מציג גישה מודולרית לארכיטקטורת תוכנה המבוססת על אבני בניין (building blocks). כל אבן בנייה היא יחידה עצמאית עם נתוני קלט, פלט והגדרה מוגדרים היטב. הפרק כולל רשימות בדיקה מפורטות לכל סוג של נתונים ומדגיש עקרונות עיצוב חיוניים כגון אחידות, הפרדת דאגות, וקלות בדיקה.

1.2 שיפורים נוספים

נוסף על שלושת הפרקים החדשים, גרסה 2.0 כוללת:

- הרחבת רשימת הבדיקה הסופית (פרק 11) לכלול התיקחות לkritriyonim הטכניים החדשים
- שילוב חלק של הערכה עצמית טכנית לצד הערכה האקדמית
- המלצות לשקלול ציוניים: 60% קרייטריונים אקדמיים ו-40% קרייטריונים טכניים
- הדגשת חשיבות הבדיקה הטכנית המעמיקה כחלק בלתי נפרד מהערכת איכות התוכנה

1.3 מטרת התוספות

התוספות בגרסה 2.0 נועדו להבטיח שפרוייקטי התוכנה שנבדקים עומדים לא רק בסטנדרטים אקדמיים ומחקריים גבוהים, אלא גם בתكني קוד ועיצוב טכניים מקצועיים. השילוב בין הערכה אקדמית להערכת טכנית מעמיקה מייצר תמונה מלאה ומקיפה של

aicohot hafero'iket v'mebtich shehstodnetim mapachim miyamnoiyot hnadesht tovuna meusiyot la'z
iculot mchakr tiaorot.

2 סקירה כללית

מסמך זה מגדר את הקритריונים להגשת פרויקט תוכנה ברמת מצוינות אקדמית המתאימה לסטודנטים מצטיינים במיוחד לתואר שני במדעי המחשב [1], [2]. הדרישות מתמקדות בפיתוח אינטראקטיבי, תיעוד מكيف, ובדגימות יכולות מחקר ופיתוח ברמה גבוהה. בכל מקום שכותב "פרויקט" הכוונה למטרה, משימה או פרויקט שנייהים במסגרת הקורס להגשת.

3 מסמכים פרויקט ותכנון

כל פרויקט תוכנהMK צוועי מתחילה בתיעוד ברור ומקיף של הדרישות והתכנון. מסמכים אלה משמשים כבסיס לפיתוח, מבטיחים הבנה משותפת בין כל הגורמים המעורבים, ומאפשרים מעקב אחר התקדמות הפרויקט לאורך זמן.

3.1 מסמך דרישות המוצר

מסמך דרישות המוצר, או Product Requirements Document (PRD), הוא המסמך המרכזי המגדיר את מטרת הפרויקט ואת הדרישות ממנו [3], [4], [5]. המסמך מתחילה בסקירה כללית של הפרויקט והקשר בו הוא פועל, כולל תיאור ברור של עיתת השימוש שהפרויקט נועד לפטור, ניתוח של השוק התחרותי והצבה אסטרטגית של הפרויקט, ויזיהו של קהל היעד והצדדים המעניינים (stakeholders). לאחר מכן, המסמך מגדר את יעד הפרויקט ומדדי ההצלחה שלו, תוך הגדרת יעדים מדדיים וברורים, מדדי KPI לכימות ההשפעה וההצלחה, וקריטריוני קבלה (acceptance criteria) המאפשרים להערכץ האם הפרויקט עומד בדרישות. חלק מרכזי נוסף במסמך הוא תיאור הדרישות הפונקציונליות והלא-פונקציונליות. זה כולל רשימת תכונות מפורטת עם עדיפויות ברורות, סיפורי משתמש (user stories) ותרחישי שימוש (use cases) המתארים כיצד המשתמשים יפעלו עם המערכת, ודרישות ביצועים, אבטחה, זמינות וscalability [6], [7], [8] המבטיחות שהמערכת תוכל לפעול בצורה יעילה ואמינה. המסמך גם מזהה הנחות, תלויות ומגבליות, כולל מערכות חיצונית ותלויות טכנולוגיות, מגבלות טכניות וארגוניות, ופריטים שנמצאים מחוץ לתחום הפרויקט (out-of-scope items). לבסוף, המסמך כולל ציר זמן מפורט ואבני דרך (timeline and milestones) עם לוח זמנים המציין נקודות ביקורת ומשולחים צפויים (deliverables) בכל שלב.

3.2 מסמך ארכיטקטורה

מסמך הארכיטקטורה (Architecture Documentation) מספק תיאור טכני מكيف של מבנה המערכת ואופן פעולתה. המסמך כולל תרשימים ויזואליים המסבירים את הארכיטקטורה ברמות שונות של פירוט, כגון תרשימי C4 Model המציגים את המערכת ברמות Context, Component, Code-and-tainer, Component, deployment וtrsimaliyim המציגים את התשתיות הטכנית והארכיטקטורה התפעולית (operational architecture).

נוסף על התרשימים, המסמך כולל תיעוד החלטות ארכיטקטוניות (Architecture Decisions - ADRs) המסביר את הרצינול להחלטות ארכיטקטוניות מרכזיות, ניתוח של trade-offs וآلטרנטיבות שנשקלו בתהליך התכנון. המסמך גם מספק תיעוד מפורט של ממשקי התכונות (API) וממשקים אחרים, כולל תיאור מפורט של כל ממשק ציבורי, סכימות נתונים (data schemas) וקונטראקטים המגדירים את האינטראקציות בין רכיבי המערכת.

4. **תיעוד קוד ומבנה פרויקט**

תיעוד קוד נכון ומבנה פרויקט מסודר הם יסודות חיוניים לפיתוח תוכנה מקצועית. תיעוד טוב מאפשר למפתחים אחרים להבין את הקוד במהירות, להשתמש בו בצורה נכונה, ולתרום לפרויקט ביעילות.

4.1. **קובץ README מקייף**

קובץ README הוא המסמך המרכזי המלאה כל פרויקט תוכנה ומשמש כמדריך למשתמש ברמת user manual מלא[9], [10]. הקובץ מתייחס בהוראות התקנה (Installation Instructions) המפרטות אתדרישות המערכת (system requirements), מספקות הוראות התקנה שלב-אחר-שלב לשביבות שונות, מסבירות כיצד משתני סביבה (environment variables), וכוללות מדריך לפתרון בעיות נפוצות (troubleshooting).

לאחר ההתקנה, הקובץ מספק הוראות הפעלה (Usage Instructions) המתארות כיצד להריץ את התוכנה במצבים שונים, מסבירות דגלים ואפשרויות ב-CLI או GUI, ומציגות workflow טיפוסי למשתמש. הקובץ גם כולל דוגמאות והדגמות (Examples & Demonstrations) עם דוגמאות קוד מעשיות להרצאה, צילומי מסך של ממשק המשתמש, תרחישי שימוש נפוצים, וקישורים לסרטוני הדוגמה אם רלוונטי.

נוסף על כך, README כולל מדריך תצורה (Configuration Guide) המסביר את קבצי הקונפיגורציה והפרמטרים הנחוצים לכיוול והשפעתם על פועלות המערכת, הנחיות לתרומות קוד (Contribution Guidelines) הכוללות תקני קוד וסגנון, ומידע על רישוי השימוש וייחוס לספריות צד שלישי ותורמים (License & Credits).

4.2. **מבנה פרויקט מודולרי**

ארגון נכון של מבנה הפרויקט הוא מפתח לתחזקה עיליה ולהתפתחות עתידית של הקוד. עקרונות הארגון כוללים חלוקה לוגית של הפרויקט לתיקות לפי תפקיד, כגון קוד מקור, בדיקות, תיעוד, נתונים, תוכאות, קונפיגורציה ומשאבים. הארגון יכול להיות מבוסס-תכונות (feature-based) או בארכיטקטורה שכבותית (layered architecture), תוך הפרדה ברורה בין קוד, נתונים, תוכאות ותיעוד.

גודל הקבצים הוא שיקול חשוב בארגון הפרויקט. קבצי קוד לא צריכים לעלות על כ-150 שורות, מה שבטיחת שכל קובץ ממוקד באחריות אחת וקל להבנה. כאשר קובץ הופך גדול מדי, יש לפרק אותו לפונקציות ומודולים קטנים יותר תוך שמירה על הפרדת

אחריות (separation of concerns). חשוב גם לשמור על עקביות בשמות תיקיות וקבצים, תוך שימוש באותו סגנון naming בכל הפרויקט.
דוגמה למבנה פרויקט מומלץ:

```
project-root/
└── src/                  # Source code
    ├── agents/            # Agent modules
    ├── utils/              # Helper functions
    └── config/             # Configuration code
    └── tests/              # Unit and integration tests
    └── data/                # Databases and input files
    └── results/             # Experiment results
    └── docs/                # Additional documentation
    └── config/              # Configuration files
    └── assets/              # Images, graphs, resources
    └── notebooks/           # Analysis notebooks
    └── README.md
    └── requirements.txt
    └── .gitignore
```

4.3 איקות קוד והערות

איקות הקוד נמדדת לא רק בפונקציונליות שלו אלא גם בקלות הקריאה והתחזוקה שלו. תקני הערות בקוד (Code Comments Standards) [11], [12], [13] דורשים שהערות יסבירו את ה-"למה" ולא רק את ה-"מה", כלומר יתמקדו בהחלטות עיצוב ורציונל ולא רק בתיאור הפעולות. כל פונקציה, מחלקה (class) ומודול (module) צריכים לכלול Docstrings המסבירים את התכליית, הפרמטרים וערכי החזרה. הערות צרכיות להסביר החלטות עיצוב מורכבות, לטעד הנחות ותנאים מוקדמים, ולהתעדכו יחד עם שינויי הקוד.

עקרונות כתיבת קוד איקוטי כוללים שימוש בשמות משתנים ופונקציות תיאוריים ומדויקים, כתיבת פונקציות קצורות ומוקדמות בעיקרון האחראיות היחידה (-sin, -cos), הימנעות מקוד כפול לפי עיקרונו (DRY - Don't Repeat Yourself), ושמירה על עקביות בסגנון הקוד לאורך כל הפרויקט.

5 ניהול קונפיגורציה ואבטחת מידע

ניהול נכון של קונפיגורציה ואבטחת מידע הם קריטיים להפעלה בטוחה של מערכות תוכנה, במיוחד כאשר מדובר בסביבות ייצור או עבודה עם מידע רגיש.

5.1 קבצי קונפיגורציה

הפרדת הגדרות הקונפיגורציה מהקוד היא עיקרון יסודי בפיתוח תוכנה מודרני. ניהול הגדרות נעשה באמצעות קבצי קונפיגורציה נפרדים בפורמטים סטנדרטיים כגון `son`, `yaml` או `env`. תוך הימנעות מקובעים (hardcoded values) בתוך הקוד עצמו. חשוב לספק קבצי דוגמה כגון `example.env`. עם ברירות מחדל שומרות על אבטחה, ולתעד כל פרמטר קונפיגורציה כדי להקל על המשתמשים להבין את ההשפעות של כל הגדרה. בעבודה עם מערכת בקרת גרסאות Git, חשוב מאוד להשתמש בקובץ `gitignore`. כדי למנוע העלאה בטעות של קבצי קונפיגורציה רגילים למאגר הקוד. כמו כן, מומלץ ליצור קבצי `template` לקונפיגורציה עבור סביבות שונות כגון פיתוח (`dev`), ביניים (`staging`) ויצור (`production`), כדי להקל על המעבר בין סביבות תוך שמירה על הפרדה נכונה.

5.2 אבטחת מידע

הגנה על מפתחות API וסודות אחרים היא קריטית למניעת דליפת מידע ושימוש לא מורשה [14], [15]. הכלל המרכזי הוא שאסור בהחלה לשמר מפתחות API בקוד המקורי (environment variables) בלבד. במקרה זה, יש להשתמש אך ורק במשתני סביבה (environment variables) כפי ש�示ה בקוד לדוגמה: `("API_KEY")`.env.get("API_KEY"). יש להסתיר את קבצי secrets מה `gitignore`., ובנסיבות יוצר להשתמש בכלים ניהול סודות (-man). secrets man. בנוסף `gitignore` מתקדמים (agement tools)()

נוסף על הגנה בסיסית, חשוב לישם החלפה תקופתית של מפתחות (rotation), לנטר השימוש במפתחות ה-API כדי לאוות חריגות, ולהגביל הרשות למינימום הנדרש (least privilege) כדי להקטין את הנזק האפשרי במקרה של פריצה.

6 בדיקות ואיכות תוכנה

בדיקות תוכנה מקיפות הן אבן הפינה של איכות הקוד ואמינות המערכת. מערכת בדיקות טוביה מגלה באגים מוקדם, מבטיחה שהקוד עומד בדרישות, ומאפשרת ביטחון בשינויים עתידיים.

6.1 בדיקות יחידה

בדיקות יחידה (Unit Tests) בודקות רכיבים בודדים של הקוד באופן מבודד. דרישות כיסוי הבדיקות (Test Coverage) [17], [18] מגדירות שיש לשאוף לכיסוי מינימלי של 70-80% ל코드 חדש, עם דגש על כיסוי מוגבר לקוד קריטי ולוגיקה עסקית. הבדיקות צריכות לכיסות לא רק את המסלולים הרגילים אלא גם מקרים קיצוניים (edge cases) ותנאי גבול.

סוגים הקיימים הנדרשים כוללים כיסוי החלטות (Statement coverage) המבטיח שככל שורת קוד הורצת לפחות פעם אחת, כיסוי ענפים (Branch coverage) המבטיח שככל החלטה נבדקה עם כל האפשרויות, וכיסוי מסלולים (Path coverage) למסלולים קריטיים המבטיח שצירופים שונים של החלטות נבדקו. לביצוע הבדיקות יש להשתמש במסגרות בדיקה סטנדרטיות

כגון unittest או pytest, לבצע אוטומציה של הבדיקות ב-pipeline CI/CD, וליצור דוחות כיסוי (coverage reports) המאפשרים מעקב אחר איכות הבדיקות.

6.2 טיפול במרקרי קיצון ותקלות

זיהוי ותיעוד מקרים קיצון (Edge Cases) הוא חלק חיוני מפיתוח תוכנה איכותית. יש לאזהות באופן שיטתי תנאי גבול ומרקרי קיצון, לטעד כל מקרה עם תיאור מפורט של הקלט הצפוי והתגובה הנדרשת, ולכלול צילומי מסך של תקלות כאשר זה רלוונטי. מנגנון טיפול בשגיאות (Error Handling) נדרש לכלול תכנות הגנתי (defensive programming) עם בדיקות קלט מקיפות, הודעות שגיאה ברורות וموעילות למשתמש, רישום לוגים מפורט לצורך ניפוי שגיאות (debugging), והידרדרות חלקה (graceful degradation) במקרים כשל כך שהמערכת ממשיכה לפעול במידת האפשר.

תיעוד התקלות נדרש לכלול תיאור מדויק של התקלה והסיבה לה, תיאור של תגובת המערכת וכי怎ד היא מטפלת בשגיאה, והערכה של ההשפעה על המשתמש או על המערכת כולה. תיעוד זה משמש גם למניעת תקלות דומות בעתיד ולשיפור מתמיד של המערכת.

6.3 תוצאות בדיקה צפויות

תיעוד תוצאות בדיקה צפויות מאפשר השוואת מהירה בין התנагות בפועל לבין התנагות המוצפפת. יש לטעד את תוצאות הרצאה הצפויות לכל בדיקה, ליצור דוחות automated testing עם המציגים את שיעור ההצלחה, ולשמור לוגים של הרצאות מוצלחות וכושלות לניתוח עתידי ולמידה מטוענית.

7 מחקר וניתוח תוצאות

מחקר עמוק וניתוח תוצאות הם שבדיל בין פרויקט תוכנה רגיל לבין עבודה אקדמית ברמת מצוינות. החלק המחקרי כולל ניסויים שיטתיים, ניתוח כמותי ו איכותי, והצגה ויזואלית של התוצאות.

7.1 חקר פרמטרים

ניתוח רגישות פרמטרים (Sensitivity Analysis) הוא תהליך שיטתי של בדיקת השפעת פרמטרים שונים על ביצוע המערכת. התהליך כולל ביצוע ניסויים שיטתיים עם שינוי מבוקר של פרמטרים, תיעוד מדויק של השפעת כל פרמטר על התוצאות, ושימוש בשיטות ניתוח מתוקדמות כגון נגזרות חלקיות (partial derivatives), ניתוח מבוסס שונות (variance-based analysis), או גישת "פרמטר-אחד-בכל-פעם" (one-at-a-time, OAT). המטרה היא לאזהות את הפרמטרים הקritisטים המשפיעים ביותר על הביצועים ולהבין את הקשרים ביניהם.

תיעוד הניסויים כולל יצירת טבלה מסודרת של כל הניסויים עם ערכי הפרמטרים והنتائج המתאימות, הפקת גרפים ממוחשיים כגון line charts וheatmaps לרגישות

בין-פרמטרית, ו-plots sensitivity להערכת השפעות, וביצוע ניתוח סטטיסטי של התוצאות לקבעת מובהקות ורמת ביטחון.

7.2 מחברת ניתוח תוצאות

מחברת ניתוח תוצאות Results Analysis Notebook היא כלי מרכזי להצגת המחקר בצורה אינטראקטיבית ומפורטת. עומק הניתוח מושג באמצעות שימוש ב-Jupyter Notebook או כלים דומים המאפשרים שילוב של קוד, טקסט ותוצאות, ביצוע ניתוח מתודדי ושיטתי של תוצאות הניסויים, השוואת בין אלגוריתמים שונים, תוצאות או גישות מתודולוגיות, והכללת הוכחות מתמטיות או ניתוחים תיאורתיים כאשר זה רלוונטי.

הכללת נוסחאות ונתוניים מפורטים לכתיבת המשוואות ונוסחאות מקצועיות, הסברים מתמטיים מפורטים למודלים ואלגוריתמים הכלולים את העקרונות התיאורתיים, ואסמכתאות בספרות אקדמית ומחקרים קודמים המקנים אמינות ומקצועיות לעובדה.

7.3 הציג ויזואלית של תוצאות

ויזואליזציה איקוונית של נתונים היא חיונית להעברת המסר המחקרי בצורה ברורה ומשמעות. סוגי הוויזואלייזציות כוללים Bar charts להשוואות קטגוריות המאפשרות השוואת מהירה בין אפשרויות שונות, Line charts ל\Migrations לאורך זמן המראות שינויים והתפתחויות, Scatter plots לזיהוי מתאימים וקשרים בין משתנים, Heatmaps להציג רגישות פרמטרים בשתי ממדים, Box plots להציג התפלגות ומדדים סטטיסטיים, ו-Waterfall charts ניתוח שינויים רציפים ותרומות יחסיות.

aicיות הגרפים נמדדת בבהירות ובדיקה התוויות, בשימושocabularies עקבאים ונגישים שמתאימים גם לאנשים עם לקויות ראייה, בכלל כתובים captions מפורטים ומקרא legends) ברווח, וברזולציה גבוהה המתאימה לפרטומים אקדמיים או מקצועיים. ככל שהגרפים מקצועיים וברורים יותר, כך ההשפעה של המחקר גדולה יותר.

8 ממוק משמש וחווית משתמש

ממוק משמש (UI - User Interface) וחווית משתמש (User Experience - UX) טובים הם קריטיים להצלחת כל מערכת תוכנה. אפילו מערכת עם פונקציונליות מעולה עלולה להימשך אם המשתמשים מתकשים להשתמש בה.

8.1 קритריוני איקות

kritериוני השימוש (Usability Criteria) כוללים מספר מימדים חשובים. קלות למידה (Learnability) מודדת עד כמה קל למשתמשים חדשים ללמידה לשימוש המערכת,יעילות (Efficiency) בודקת עד כמה מהר משתמשים מנוסים יכולים לבצע משימות, זכירות (Memorability) מעריכה עד כמה קל למשתמשים לחזור למערכת לאחר הפסקה ולזכור

כיצד להשתמש בה, מניעת שגיאות (Error Prevention) בודקת עד כמה המערכת מגינה על המשתמש מפני טעויות, ושביעות רצון (Satisfaction) מודדת עד כמה משתמשים נהנים מהעבודה עם המערכת.

עשרה ההיוריסטיות של נילסן (Nielsen's 10 Heuristics) [21] הן קבוצה מוכרת של עקרונות לעיצוב ממשקם. הן כוללות נראות סטטוס המערכת (Visibility of system status), התאמה בין המערכת והעולם האמיתי (Match between system and real world), שליטה וחופש למשתמש (Consistency and standards), עקביות ותקינה (User control and freedom), מניעת שגיאות (Recognition over recall), זיהוי במקום זיכרון (Error prevention), גמישות ויעילות שימוש (Flexibility and efficiency of use), עיצוב אסתטי ומינימליסטי (Aesthetic and minimalist design), עזרה למשתמשים לזהות ולהתואושש משגיאות (Help users recognize and recover from errors), ועזרה ותיעוד (Help and documentation).

8.2 תיעוד ממشك

תיעוד מקיף של הממשק כולל צילומי מסך של כל מסך ומצב אפשרי במערכת, תיאור מפורט של workflow טיפוסי של משתמש המראה את המסלול שלהם מתחילה השימוש ועד להשגת המטרה, הסברים על אינטראקציות ופידבק שהמערכת נותנת למשתמש בתגובה לפעולות שונות, ושיקולי נגישות (accessibility considerations) המבטיחים שהמערכת שמיישת גם לאנשים עם מוגבלות.

9 ניהול גרסאות ותיעוד פיתוח

ניהול גרסאות נכון הוא חיוני לעובדות צוות, למעקב אחר שינויים, ולאפשרות לחזור לגרסאות קודמות במקרה הצורך. בנוסף, תיעוד תהליכי הפיתוח עוזר להבין את החלטות שנתקבלו לאורך הדרכ.

9.1 שיטות עבודה מומלצות עם Git

שיטות עבודה מומלצות (Git Best Practices) כוללות שמירה על היסטוריה commits ברורה עם הודעות משמעותיות המתארות מה השתנה ולמה, שימוש ב-branches נפרדים לפיתוח תוכנות חדשות כדי לשמר על יציבות הענף הראשי, ביצוע סקירות קוד (code reviews) באמצעות Pull Requests לפני מיזוג שינויים, ושימוש ב-tagging לסימון גרסאות מרכזיות ומשמעותיות של המערכת.

9.2 ספר הפרומפטים

תיעוד תהליכי הפיתוח עם בינה מלאכותית (Prompt Engineering Log) הוא חלק חדש וחשוב בפיתוח תוכנה מודרני. התיעוד כולל רשימה של כל הפרומפטים המשמעותיים ששימושו לבנייתuproject, תיאור של ההקשר והמטרה של כל פרומפט, דוגמאות לפלאטים שהתקבלו ואיך הם שולבו בפרויקט, תיעוד של שיפורים איטרטיביים של פרומפטים לאורץ זמן, ושיטות

עובדת מומלצות (best practices) שהופקו מהניסיון. מבנה מומלץ לティקית הפרומפטים כולל תיקיות נפרדות לפרומפטים של תכנון ארכיטקטורה, ייצור קוד, בדיקות, וтиיעוד, עם קובץ סקירה כללי.

10 עלויות ותមhor

הבנת עלויות הפיתוח והתפעול היא חיונית לתכנון נכון של הפרויקט ולקבלת החלטות מושכלות לגבי משאבם וטכנולוגיות.

10.1 ניתוח עלויות

ניתוח עלויות (Cost Breakdown) של שימוש ב-API Tokens כולל ספירה מדוקית של tokens בכניסה והן ביציאה (input/output tokens), חישוב העלות למיליאן tokens (per Mtokens) לפי התעריפים של כל ספק שירות, והערכת העלות הכוללת לפי מודל ושירות. הדוגמה הבאה מציגה ניתוח עלויות טיפוסי:

טבלה 1: ניתוח עלויות API Tokens

Total Cost	הוצאות כוללת / Output Tokens	Input Tokens	מודל / Model
\$45.67	523,000	1,245,000	GPT-4
\$32.11	412,000	890,000	Claude 3
\$77.78	935,000	2,135,000	סה"כ / Total

אסטרטגיית אופטימיזציה כוללת הפחתת שימוש ב-tokens באמצעות סיכום וקיצור של פרומפטים, שימוש ב-batch processing לעיבוד מרובה ויעיל יותר, ובבחירה מודלים לפי יחס עלות-תועלת (cost-effectiveness) כאשר מודלים זולים יכולים לתת תוצאות מספקות.

10.2 ניהול תקציב

ניהול תקציב יעל כולל תחזית עלויות לסקירה עתידית כדי להבין את העלות הצפויות כאשר המערכת גדלה, ניטור (monitoring) של שימוש בזמן אמיתי להזות חריגות מוקדם, והגדלת התראות על חריגת מתќיב כדי למנוע הוצאות לא מתוכנות.

11 הרחבה ותחזוקתיות

תכנן מערכת שנייה להרחבת ולתחזק בקלות הוא השקעה לטווח ארוך שמשתלמת כאשר המערכת צריכה להפתח ולהשתנות.

11.1 נקודות הרחבה

ארQUITECTורת תוספים (Plugins Architecture) מאפשרת הוספה פונקציונליות חדשה ללא שינוי בקוד הליבה. זה מושג באמצעות הגדרת ממשקים (interfaces) ברורים להרחבה

המגדירים את החוצה בין הליבה לתוספים, הוספת נקודות חיבור (lifecycle hooks) כגון beforeCreate, afterUpdate middleware לעיבוד שרשרת של בקשות, ועיצוב מבוסס-API (API-first design) שמבטיח שכל הפונקציונליות נגישה דרך ממשקים מוגדרים היטב.

תיעוד הרחבה כולל הדרכה מפורטת לפיתוח plugins, דוגמאות לתוספים פשוטים ומורכבים, ותיאור של כללים ונהלים (conventions) להרחבה בטוחה שלא תשבור את המערכת.

11.2 תחזוקתיות

קוד ניתן לתחזקה (Maintainable Code) מאופיין במודולריות (Modularity) והפרדת אחריות (separation of concerns) כך שכל חלק בקוד אחראי על דבר אחד בלבד, שימוש חוזר (Reusability) של קומponentות כך שקוד כתוב פעם אחת ומשמש במקומות רבים, ניתנות לניתוח (Analyzability) כך שקל להבין את הקוד ולזהותו בעיות, וניתנות לבדיקה (Testability) כך שקל לכתוב בדיקות אוטומטיות לקוד.

12 תקני איכות בינהוומיים

תקן ISO/IEC 25010 [22] מגדיר מודל מكيف לאיכות תוכנה המכסה שמונה מאפייני איכות עיקריים. כל מאפיין מחלק למאפייני משנה המאפשרים הערכה מפורטת ואובייקטיבית של איכות המוצר.

12.1 מאפייני איכות מוצר

התאמה פונקציונלית (Functional Suitability) בודקת עד כמה המערכת עומדת בדרישות הפונקציונליות, כולל שלמות (Completeness) של כיסוי כל התכונות הנדרשות, נכונות (-Correctness) של התוצאות, והתאמה (Appropriateness) למשימות שהמערכת אמורה לבצע. יעילות ביצועים (Performance Efficiency) מעריכה את התנהלות הזמן (Time behavior) כולל זמני תגובה, ניצול משאבים (Resource utilization) כולל זיכרון ומעבד, ויכולת (Capacity) להתמודד עם עומסים גדולים.

תאימות (Compatibility) בוחנת יכולת פעולה הדדית (Interoperability) עם מערכות אחרות ודו-קיום (Coexistence) במקביל למערכות אחרות. שימושיות (Usability) כוללת קלות למדיה, יכולת הפעלה, נגישות, הגנה מפני שגיאות משתמש, ואסתטיקה של הממשק. אמינות (Reliability) מודדת בשלות המערכת (Maturity), זמינות (Availability), סובלנות (Recoverability), ויכולת התאוששות (Fault tolerance).

אבטחה (Authenticity) כוללת סודיות (Confidentiality), שלמות (Security), אימיות (Integrity), אחראיות (Accountability), וא-הכחשה (Non-repudiation). תחזוקתיות (Maintainability) מעריצה מודולריות, שימוש雄厚, ניתנות לניתוח, ניתנות לשינוי (Modifiability), וניתנות לבדיקה. לבסוף, ניידות (Portability) בוחנת התאמה (Adaptability) לנסיבות שונות, ניתנות להתקנה (Installability), וניתנות להחלפה (Replaceability).

13 רשות בדיקה סופית

לפני הגשת הפרויקט, חשוב לעבור על רשות בדיקה מקיפה כדי לוודא שכל הדרישות מולאנו. התיעוד צריך לכלול מסמך PRD מפורט עם כל המרכיבים, תיעוד ארכיטקטורה עם תרשימי בלוקים ברורים, קובץ README מקיים בرمת מדריך משתמש מלא, תיעוד API מלא לכל המשקימים הציבוריים, וספר פרומפטים מתועד. הקוד עצמו צריך להיות מאורגן במבנה פרויקט מודולרי ומסודר, עם קבצים שלא עולים על 150 שורות, הערות קוד מקיפות docstrings לכל פונקציה ומחלקה, ועקבות בסגנון הקוד לאורך כל הפרויקט.

הkonfiguracija צריכה להיות נפרדת מהקוד, עם קבצי דוגמה כמו `env.example`, ללא מפתחות API בקוד המקורי, ועם `gitignore`. מעודכן. הבדיקות צריכה לכלול unit tests עם כיסוי של לפחות 70%, תיעוד מקרי קיצון (edge cases), טיפול מקיף בשגיאות (error handling), ודוחות בדיקה אוטומטיים. החלק המחברי צריך לכלול ניסויים עם שינוי פרמטרים, ניתוח רגישות מתועד, מחברת ניתוח עם גרפים ממחישים, ונוסחאות מתמטיות אשר רלוונטי.

הויזואלייזציה צריכה לכלול גרפים אינטuitיביים של תוכאות, צילומי מסך של ממש המשמש, ותרשיimi ארכיטקטורה ברורים. ניתוח העליות צריכה לכלול טבלת שימוש tokens, ניתוח עליות מפורט, וסטרטגיית אופטימיזציה. ההרחבה צריכה לכלול נקודות הרחבה (extension points) מתועדות, דוגמאות לפיתוח תוספים (plugins), וממשקים ברורים להרחבה.

13.1 בדיקה טכנית מפורטת

נוסף על הקriterיוונים לעיל, יש לוודא עמידה בקריטריוניים הטכניים המפורטים בפרקים 13–15:

- ארגון הפרויקט כחילה עם קבצי `pyproject.toml/setup.py` קבצי `__init__.py`, ומבנה תיקיות מאורגן
- שימוש נכון במעבדים מרובבים (multiprocessing) לפעולות CPU-bound
- שימוש נכון בחוטי ביצוע (multithreading) לפעולות O-I, כולל בטיחות חוטים
- עיצוב מבוסס אבני בניין עם הגדרות ברורות של קלט, פלט והגדרה לכל מודול
- ולידציה מקיפה של כל נתוני הקלט
- תיעוד מפורט של כל אבן בנייה והתלויות שלה

לבסוף, יש לוודא היסטוריית Git מסודרת, רישוון מצורף, ייחוס לספריות צד שלישי, והוראות התקנה והפעלה (deployment).

14 מקורות ותקנים נוספים

לצורך הכתת פרויקט ברמתมาตรฐาน, מומלץ להתייחס לתקנים ומקורות בינלאומיים מוכרים. אלה כוללים את תכנית אבטחת אינטלקט ציבורי של MIT [23], מודל אינטלקט התוכנה ISO/IEC 25010 [22], שיטות העבודה ההנדסיות של Google [24], ההנחיות ל-API של Microsoft [25] וההיוריסטיות לשימושות של נילסן [21]. מקורות אלה מספקים בסיס מוצק לעובדה מקצועית ואקדמית בرمאה הגבוהה ביותר.

15 ארגון הפרויקט כחבילת

ארגון הקוד כחבילת (package) הוא עיקרונו יסוד בפיתוח תוכנה מקצועי. חבילה מאורגנת נכון מאפשרת שימוש בחומרה בקוד במספר פרויקטים, ניהול תלויות (dependencies) בצורה ברורה, הפעלה והתקנה פשוטה, ובדיקות (testing) מובנות. פרק זה מספק רשות בדיקה מפורטת לארגון הפרויקט כחבילת Python מקצועי.

15.1 קובץ הגדרת החבילה

כל חבילה מקצועית צריכה כולל קובץ הגדרה המפרט את מאפייני החבילה ותלוויותיה. ניתן להשתמש בקובץ `setup.py` המסורתי או בקובץ `toml` `pyproject.toml` המודרני יותר. קובץ ההגדרה צריך לכלול את שם החבילה, מספר הגרסה, תיאור קצר, שם המחבר, רישיון השימוש, ורשימה מלאה של כל התלוויות החיצוניתות עם מספרי גרסאות ספציפיים או טוחני גרסאות מקובלים.

15.2 קבצי `__init__.py`

קובצי `__init__.py` הם המנגנון המרכזי בו Python מזהה תיקייה כחבילת. קובץ זה צריך להיות בתיקייה הראשית של החבילה ובכל תת-תיקייה שאמורה להיות תת-חבילה. הקובץ יכול להישאר ריק, אך מומלץ להשתמש בו ליצוא (export) של הממשקים הציבוריים של החבילה באמצעות המשתנה `__all__`, ולהגדרת קבועים כגון `__version__` המציין את גרסת החבילה. בקובץ זה ניתן גם לבצע אתחולן חדש לחבילה, אך יש להימנע מלוגיקה מורכבת שעלולה להאט את טיעינת החבילה.

15.3 מבנה תיקיות מאורגן

ארגון התקיות צריך להיות לוגי ועקובי. קוד המקור צריך להיות בתיקייה ייועדת, בדרך כלל/`src` או בתיקייה הנושאת את שם החבילה. הבדיקות צרכות להיות בתיקייה נפרדת בשם/`tests`, והтиיעוד בתיקייה נפרדת בשם/`docs`. הפרדה זו מבטיחה שקוד הייצור, קוד הבדיקות, והтиיעוד לא מעורבים זה זהה, מה שמקל על תחזוקה וניהוט בפרויקט.

15.4 שימוש בנתיבים יחסיים

כל הייבואים (imports) בקוד צריכים להשתמש בנתיבים יחסיים או בשמות חבילות, ולעתום לא בנתיבים מוחלטים. למשל, במקרים לכתוב `import /path/to/module`, יש להשתמש `from mypackage.submodule import function`. גם כאשר מבצעים קריאה או כתיבה של קבצים, יש לחשב את הנתיב באופן יחסי למיקום החבילה ולא למיקום קובץ ההרצה. זה מבטיח שהחבילה תעבור בכל סביבה ללא תלות במבנה הספציפי של מערכת הקבצים.

15.5 רשימת בדיקה: ארגון בחבילה

עbero על הפריטים הבאים ובדקו את הפרויקט שלכם:

1. קובץ הגדרת חבילה:

- האם קיים קובץ `setup.py` או `pyproject.toml`?
- האם הקובץ מכיל את כל המידע הנדרש (שם, גרסה, תלויות)?
- האם התלויות מפורטות עם מספרי גרסאות?

2. קובץ `py.__init__.py`:

- האם קיים קובץ `py.__init__.py` בתיקייה הראשית?
- האם הקובץ מיצא את הממשקים הציבוריים?
- האם `__version__` מוגדר בקובץ זה?

3. מבנה תיקיות:

- האם קוד המקור נמצא בתיקייה "יעודית"?
- האם הבדיקות נמצאות בתיקייה נפרדת `?tests`?
- האם התיעוד נמצא בתיקייה נפרדת `?docs`?

4. נתיבים יחסיים:

- האם כל היבואים משתמשים בנתיבים יחסיים?
- האם הקוד נמנע מנתיבים מוחלטים?
- האם קריאה/כתיבה של קבצים נעשית ביחס לנתיב החבילה?

15.6 דוגמה למבנה נכון

מבנה תיקיות מומלץ לחבילה:

```

my_project/
└── src/
    └── my_package/
        ├── __init__.py
        ├── core.py
        └── utils.py
└── tests/
    ├── __init__.py
    └── test_core.py
└── docs/
└── setup.py
└── README.md
└── requirements.txt
└── .gitignore

```

16. עיבוד מקבילי וביצועים

שימוש במעבדים מרובים (multiprocessing) ובחוטי ביצוע מרובים (multithreading) הוא חיוני לביצועים אופטימליים של תוכנה מודרנית. הבנת השימוש הנכון בכלים אלו היא קריטית לפיתוח מערכותיעילות שמנצלות את מלאה יכולות החומרה המודרנית.

16.1 הבדל בין Multithreading ל-Multiprocessing

ההבדל המרכזי בין שתי הגישות נובע מסוג העומס על המערכת. Multiprocessing מתאים לפעולות תובעניות מבחינת מעבד (CPU-bound) כגון חישובים מתמטיים מורכבים, עיבוד תמונות, או אימון מודלים של למידת מכונה. בפעולות אלו, כל תהליך (process) פועל בזיכרון נפרד ויכול לנצל ליבת מעבד שונה, מה שמאפשר מקביליות אמיתייה.

לעומת זאת, Multithreading מתאים לפעולות תובעניות מבחינת קלט/פלט (I/O-bound) כגון קריאות רשת, גישה למסדי נתונים, או קריאה וכתיבת של קבצים. המערכת ממתינה רוב הזמן לתגובה מרכיבים חיצוניים, וחוטי ביצוע (threads) מאפשרים לבצע פעולות אחרות בזמן ההמתנה.

16.2 שימוש ב-Multiprocessing

בעת שימוש ב-`multiprocessing`, חשוב להזיהות פעולות במערכת שתובעניות מבחינת מעבד ומתקימות למקבול (parallelization). יש להשתמש במודול `multiprocessing` של Python ולהגדיר את מספר התהליכים באופן דינמי על פי מספר הלייבות הזמן במערכת, למשל באמצעות `multiprocessing.cpu_count()`. חשוב לטפל בשיתוף נתונים בין תהליכים

בצורה נכונה באמצעות מנגןונים כגון Queue או Pipe, ולודא שהתהליכים נסגרים כראוי בסיום העבודה כדי למנוע זליגת משאבים.

16.3 שימוש ב-Multithreading

בעת שימוש ב-multithreading, יש לאԶות פעולות שתובעניות מבחינת קלט/פלט וכוללות המתנה לתגובה מרכיבים חיצוניים. יש להשתמש במודול threading של Python ולנהל את חוטי הביצוע בצורה מסודרת. קריטי להבטיח סyncronization בין חוטים באמצעות מנעולים (locks) או סמפורים (semaphores) כדי למנוע מצב תחרות (race conditions). יש להגן על משתנים משותפים באמצעות מנעולים ולהימנע מנעילה הדדית (deadlocks) על ידי תכנון קפדי של סדר נעילה ושחרור.

16.4 בטיחות חוטים

בטיחות חוטים (thread safety) היא היבט קריטי בעבודה עם multithreading. יש להימנע ממצבים תחרות על ידי הגנה על כל גישה למשתנים משותפים באמצעות מנעולים. יש להשתמש במנגנונים בוטחים לחוטים כמו Queue. Queue מאפשר מעבירים מידע בין חוטים. חשוב גם להימנע מנעילה הדדית על ידי שמירה על סדר קבוע של נעילה ושחרור מנעולים, ושימוש ב-`stnemetats with` (context managers) להבטיח שחרור אוטומטי של מנעולים.

16.5 רישימת בדיקה: עיבוד מקבילי

עbero על הפריטים הבאים ובדקו את הפרויקט שלכם:

1. זיהוי פעולות מתאימות:

- האם זיהיתם פעולות תובעניות מבחינת מעבד או קלט/פלט?
- האם בחרתם את הכלי הנכון (multiprocessing או multithreading)?
- האם העריכתם את התועלת הפוטנציאלית מביצוע מקבילי?

2. יישום נכון:

- האם מספר התהליכים/חוטים מוגדר דינמית?
- האם שיתוף נתונים מבוצע בצורה בטוחה?
- האם קיים סyncronization בין ייחדות הביצוע?

3. ניהול משאבי:

- האם התהליכים/חוטים נסגרים כראוי?
- האם קיים טיפול בחיריגות (exceptions)?
- האם נמנעים מזילגת זיכרון?

4. בטיחות (ל-*multithreading*):

- האם משתנים משותפים מוגנים באמצעות מנעולים?
- האם נמנעים ממצבי תחרות?
- האם נמנעים מנעליה הדדית?

17. עיצוב מודולרי ואבני בניין

עיצוב מבוסס אבני בניין (building blocks design) הוא גישה מודולרית לארכיטקטורת תוכנה שבה כל רכיב במערכת הוא יחידה עצמאית עם משקל מוגדר היטב. גישה זו מקלה על תחזקה, בדיקה, ושימוש חוזר בקוד, ומאפשרת פיתוח מערכות גדולות וモרכבות בצורה מסודרת ו邏輯ית.

17.1 מבנה אבן בניה

כל אבן בניה במערכת מוגדרת על ידי שלושה סוגים של נתונים:

נתוני קלט (Input Data) – המידע הנדרש לביצוע הפעולה. יש להגדיר בצורה ברורה את סוגי הנתונים, התחום התקף לכל פרמטר, ואת התלויות החיצונית. כל נתוני הקלט חייבים לעבור ולידציה מקיפה לפני השימוש.

נתוני פלט (Output Data) – התוצריים שהאבן מייצרת. יש להגדיר את סוגי הנתונים, את הפורמט של הפלט, ואת התנונות הפלט במקרי קצה וشنויות. הפלט צריך להיות עיקבי ומוגדר היטב.

נתוני הגדרה (Setup Data) – פרמטרים וקונפיגורציה לאבן הבניה. אלו כוללים פרמטרים קונפיגורטיביים עם ערכי ברירת מחדל סבירים, הגדרות שנטענות מקבצי קונפיגורציה או משתני סביבה, ופרמטרי אתחול הנדרשים לפני שימוש באבן הבניה.

17.2 עקרונות עיצוב

עיצוב טוב של אבני בניין צריך לעמוד בעקרונות הבאים:

אחריות יחידה (Single Responsibility) – כל אבן בניה אחראית למשימה אחת מוגדרת. זה מקל על הבנת הקוד, תחזקה, ובדיקה.

הפרזת דאגות (Separation of Concerns) – כל אבן בניה עוסקת בהיבט אחד של המערכת ואינה מעורבת בהיבטים אחרים. לדוגמה, אבן בניה שמבצעת חישובים לא תהיה אחראית גם על שמירת התוצאות לדיסק.

קלות שימוש חוזר (Reusability) – אבני הבניה ניתנות לשימוש חוזר בהקשרים שונים. הן לא תלויות בקוד ספציפי למערכת ויכולות לפעול באופן עצמאי.

יכולת בדיקה (Testability) – כל אבן בניה ניתנת לבדיקה באופן עצמאי ללא תלות ביתר המערכת. התלוויות מסוימות דרך הזרקת תלות (dependency injection).

17.3 ולידציה והגנה

כל אבן בניתה צריכה לכלול ולידציה מקיפה של נתוני הקלט. יש לבדוק את סוגי הנתונים, את התחומים התקף, ואת התנאים המוקדמים. במקרה של קלט לא תקין, יש להחזיר הודעה שגיאה ברורה ומוסילת שמשבירה מה השتبש ומה צריך לתקן. הבדיקותricesות להתבצע מוקדם ככל האפשר (fail fast) כדי למנוע התפשטות שגיאות למערכת.

17.4 רשימת בדיקה: עיצוב אבני בניין

עbero על הפריטים הבאים ובדקו את הפרויקט שלכם:

1. זיהוי אבני בניין:

- האם יצרתם תרשימים של המערכת וזיהיתם אבני בנייה?
- האם כל אבן בניתה מוגדרת כמחלקה או פונקציה נפרדת?
- האם לכל אבן בניתה יש שם תיאורי ותיעוד מפורט?

2. נתוני קלט:

- האם כל נתוני הקלט מתועדים ברורה?
- האם קיימת ולידציה לכל נתוני הקלט?
- האם התלוויות מסוימות דרך הזרקת תלות?

3. נתוני פלט:

- האם כל נתוני הפלט מתועדים?
- האם הפלט עקובי בכל מצב?
- האם השגיאות מתועדות ומוחזרות בצורה ברורה?

4. נתוני הגדרה:

- האם כל הפרמטרים הקונפיגורטיביים זוהו?
- האם קיימים ערכי ברירת מחדל סבירים?
- האם הקונפיגורציה מופרדת מהקוד?

5. עקרונות עיצוב:

- האם כל אבן בניתה עומדת בעיקרונו האחוריות היחידה?
- האם קיימת הפרדת דאגות ברורה?
- האם אבני הבנית ניתנות לשימוש חוזר ולבדיקה?

17.5 דוגמה לאבן בניה טובה

להלן דוגמה קונספטואלית לאבן בניה המעבדת נתונים (DataProcessor). אבן הבניה האז ממחישה את העקרונות המרכזיים:
מבנה האבן:

- **נתוני קלט** – רשימת מילוניים (raw_data) וקריטריוני סינון (filter_criteria)
- **נתוני פלט** – רשימת מילוניים מעובדים (processed_data)
- **נתוני הגדרה** – מצב עיבוד (processing_mode) ('fast' או 'accurate') וגודל אצווה (batch_size)

התכונות המרכזיות של הדוגמה:

1. **אחריות יחידה** – האבן אחראית רק על עיבוד נתונים, לא על שמירתם או טעינתם
2. **הפרצת דאגות** – קונפיגורציה נפרדת לוגיקת העיבוד
3. **ולידציה מקיפה** – בדיקת קלט לפני עיבוד
4. **ניתנות לבדיקה** – כל פונקציה ניתנת לבדיקה באופן עצמאי
5. **שימוש חוזר** – ניתן להשתמש באותה אבן בהקשרים שונים

יישום הדוגמה:

Building Block Example - Part 1

```
class DataProcessor:  
    """  
        Building block for processing data  
  
        InputData:  
            - raw_data: List[Dict] - list of dictionaries  
            - filter_criteria: Dict - filtering criteria  
  
        OutputData:  
            - processed_data: List[Dict] - processed dictionaries  
  
        SetupData:  
            - processing_mode: str ('fast'/'accurate')  
            - batch_size: int (default: 100)  
    """  
  
    def __init__(self, processing_mode='fast',  
                 batch_size=100):  
        # Setup configuration  
        self.processing_mode = processing_mode  
        self.batch_size = batch_size  
        self._validate_config()  
  
    def process(self, raw_data, filter_criteria):  
        """Process data"""  
        # Input validation  
        self._validate_input(raw_data,  
                             filter_criteria)  
  
        # Processing logic  
        result = self._do_processing(raw_data,  
                                     filter_criteria)  
  
        # Return output  
        return result
```

Building Block Example - Part 2

```
def _validate_config(self):
    """Validate configuration"""
    valid_modes = ['fast', 'accurate']
    if self.processing_mode not in valid_modes:
        raise ValueError(
            f"Invalid mode: {self.processing_mode}"
        )
    if self.batch_size <= 0:
        raise ValueError(
            "Batch size must be positive"
        )

def _validate_input(self, raw_data, criteria):
    """Validate input data"""
    if not isinstance(raw_data, list):
        raise TypeError("raw_data must be list")
    if not isinstance(criteria, dict):
        raise TypeError("criteria must be dict")

def _do_processing(self, data, criteria):
    """Perform actual processing"""
    # Main logic here
    filtered = [item for item in data
                if self._matches(item, criteria)]
    return filtered

def _matches(self, item, criteria):
    """Check if item matches criteria"""
    return all(item.get(k) == v
              for k, v in criteria.items())
```

שימוש בדוגמה:

Usage Example

```
# Create building block with configuration
processor = DataProcessor(
    processing_mode='accurate',
    batch_size=50
)

# Input data
data = [
    {'name': 'Alice', 'age': 30, 'city': 'TLV'},
    {'name': 'Bob', 'age': 25, 'city': 'JLM'},
    {'name': 'Charlie', 'age': 30, 'city': 'TLV'}
]

# Filter criteria
criteria = {'age': 30, 'city': 'TLV'}

# Process
result = processor.process(data, criteria)
# Result: [{name': 'Alice', ...},
#           {'name': 'Charlie', ...}]
```

18 הערכה חשובה

מסמך זה מציג רמת מצוינות גבוהה במיוחד. לא כל סעיף הוא מחויב במלואו, אך ככל שיותר קритריונים מתקיים, כך הציון והערכת האיכות יהיו גבוהים יותר. התמקדו בעומק, במקרים ובהדגמת יכולות מחקר ברמה אקדמית גבוהה. מומלץ להשתמש בכלים LLM לעזרה בהשלמת הפרויקט. מובהר כי חלק מהבדיקה יתבצע וייעשה שימוש בסוכני AI לביצוע הבדיקה.

18.1 הערכה משולבת

הערכת הפרויקט צריכה לשלב בין הקритריונים האקדמיים (פרק 1–12) לבין הקритריונים הטכניים (פרק 13–15). שכלל מומלץ הוא 60% קритריונים אקדמיים ו-40% קритריונים טכניים. גישה משולבת זו מבטיחה שהפרויקט עומד לא רק בסטנדרטים מחקריים אלא גם בתקנים הנדרת תוכנה מקצועיים.

19 English References

- 1 MIT ACIS, *Mit software quality assurance plan*, <https://acisweb.mit.edu/acis/sqap/sqap.r1.html>, 2022.
- 2 A. Downey, *Software engineering practices for scientists*, <http://allendowney.blogspot.com/2013/05/software-engineering-practices-for.html>, 2013.
- 3 Monday.com, *Prd template - product requirement document*, <https://monday.com/blog/rnd/prd-template-product-requirement-document/>, 2024.
- 4 Miro, *Modular prd template*, <https://miro.com/templates/modular-prd/>, 2024.
- 5 Aha! *What is a good product requirements document template*, <https://www.aha.io/roadmapping/guide/requirements-management>, 2024.
- 6 Pacific Certification, *Iso 25010 software product quality model*, <https://blog.pacificcert.com/iso-25010-software-product-quality-model/>, 2024.
- 7 ISO 25000, *Iso 25010 standards overview*, <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>, 2024.
- 8 ISO, *Iso/iec 25010:2011 systems and software quality requirements and evaluation*, <https://www.iso.org/standard/35733.html>, 2011.
- 9 Archbee, *Readme files guide*, <https://www.archbee.com/blog/readme-files-guide>, 2024.
- 10 GitHub, *About readmes*, <https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes>, 2024.
- 11 Daily.dev, *10 code commenting best practices for developers*, <https://daily.dev/blog/10-code-commenting-best-practices-for-developers>, 2024.
- 12 Stack Overflow, *Best practices for writing code comments*, <https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments/>, 2021.
- 13 Codacy, *Code documentation best practices*, <https://blog.codacy.com/code-documentation>, 2024.
- 14 Hoop.dev, *Api security best practices: Protecting secrets with environment variables*, <https://hoop.dev/blog/api-security-best-practices-protecting-secrets-with-environment-variables/>, 2024.

- 15 Claude Support, *Api key best practices: Keeping your keys safe and secure*, <https://support.claude.com/en/articles/9767949-api-key-best-practices>, 2024.
- 16 OpenAI, *Best practices for api key safety*, <https://help.openai.com/en/articles/5112595-best-practices-for-api-key-safety>, 2024.
- 17 PractiTest, *Test coverage metrics*, <https://www.practitest.com/resource-center/blog/test-coverage-metrics/>, 2024.
- 18 Google Testing Blog, *Code coverage best practices*, <https://testing.googleblog.com/2020/08/code-coverage-best-practices.html>, 2020.
- 19 UX Design, *Measuring design quality with heuristics*, <https://uxdesign.cc/measuring-design-quality-with-heuristics-44857efa514>, 2024.
- 20 J. Nielsen, *10 usability heuristics for user interface design*, <https://www.nngroup.com/articles/ten-usability-heuristics/>, 1994.
- 21 J. Nielsen, *10 usability heuristics for user interface design*, <https://www.nngroup.com/articles/ten-usability-heuristics/>, 1994.
- 22 ISO, *Iso/iec 25010:2011 systems and software quality requirements and evaluation*, <https://www.iso.org/standard/35733.html>, 2011.
- 23 MIT ACIS, *Mit software quality assurance plan*, <https://acisweb.mit.edu/acis/sqap/sqap.r1.html>, 2022.
- 24 Google, *Google engineering practices documentation*, <https://google.github.io/eng-practices/>, 2023.
- 25 Microsoft, *Microsoft rest api guidelines*, <https://github.com/microsoft/api-guidelines>, 2023.