

הנחיות להגשת תוכנה ברמת הצטיינות יתרה לתואר שני במדעי המחשב

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

גרסה 2.0

ד"ר יורם סגל

© Dr. Segal Yoram - כל הזכויות שמורות

22-11-2025

תוכן העניינים

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

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

1 חדש בגרסה 2.0

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

1.1 פרקים חדשים שנוספו

פרק 13: ארגון הפרויקט כחבילה – פרק זה מספק רשימת בדיקה מפורטת לארגון הפרויקט כחבילה (Python package) מקצועית. הפרק כולל קריטריונים לקבצי הגדרה (`setup.py`) או (`pyproject.toml`), מבנה תיקיות מאורגן, קבצי `__init__.py`, שימוש בנתיבים יחסיים, וניהול תלויות. ארגון נכון כחבילה מאפשר שימוש חוזר בקוד, ניהול תלויות ברור, והפצה והתקנה פשוטה.

פרק 14: עיבוד מקבילי וביצועים – פרק זה עוסק בשימוש נכון במעבדים מרובים (multi-processing) ובחומי 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 ובטיחות חוטים.

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

1.2 שיפורים נוספים

נוסף על שלושת הפרקים החדשים, גרסה 2.0 כוללת:

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

1.3 מטרת התוספות

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

איכות הפרויקט ומבטיח שהסטודנטים מפתחים מיומנויות הנדסת תוכנה מעשיות לצד יכולות מחקר תיאורטיות.

2 סקירה כללית

מסמך זה מגדיר את הקריטריונים להגשת פרויקט תוכנה ברמת מצוינות אקדמית המתאימה לסטודנטים מצטיינים במיוחד לתואר שני במדעי המחשב [1], [2]. הדרישות מתמקדות בפיתוח איכותי, תיעוד מקיף, ובהדגמת יכולות מחקר ופיתוח ברמה גבוהה. בכל מקום שכתוב "פרויקט" הכוונה למטלה, משימה או פרויקט שנתינים במסגרת הקורס להגשה.

3 מסמכי פרויקט ותכנון

כל פרויקט תוכנה מקצועי מתחיל 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 מתחיל בסקירה כללית של הפרויקט והקשר בו הוא פועל, כולל תיאור ברור של בעיית המשתמש שהפרויקט נועד לפתור, ניתוח של השוק התחרותי והצבה אסטרטגית של הפרויקט, וזיהוי של קהל היעד והצדדים המעוניינים (stakeholders). לאחר מכן, המסמך מגדיר את יעדי הפרויקט ומדדי ההצלחה שלו, תוך הגדרת יעדים מדידים וברורים, מדדי KPI לכימות ההשפעה וההצלחה, וקריטריוני קבלה (acceptance criteria) המאפשרים להעריך 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) ותרשימי שימוש (use cases) המתארים כיצד המשתמשים יפעלו עם המערכת, ודרישות ביצועים, אבטחה, זמינות ו-scalability [6], [7], [8] המבטיחות שהמערכת תוכל לפעול בצורה יעילה ואמינה. המסמך גם מזהה הנחות, תלויות ומגבלות, כולל מערכות חיצוניות 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 ומשלוחים צפויים (deliverables) בכל שלב.

3.2 מסמך ארכיטקטורה

מסמך הארכיטקטורה (Architecture Documentation) מספק תיאור טכני מקיף 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, Container, Component ו-Code, תרשימי UML המתארים תהליכים מורכבים ואינטראקציות בין רכיבים, ותרשימי deployment המציגים את התשתית הטכנולוגית והארכיטקטורה התפעולית (operational architecture).

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

4 תיעוד קוד ומבנה פרויקט

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

4.1 קובץ EMDAER מקיף

קובץ ה-README הוא המסמך המרכזי המלווה כל פרויקט תוכנה ומשמש כמדריך למשתמש ברמת user manual מלא [9], [10]. הקובץ מתחיל בהוראות התקנה (Installation Instructions) המפרטות את דרישות המערכת (system requirements), מספקות הוראות התקנה שלב-אחר-שלב לסביבות שונות, מסבירות כיצד להגדיר 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 להרצה, צילומי מסך של ממשק המשתמש, תרחישי שימוש נפוצים, וקישורים לסרטוני הדגמה אם רלוונטי.

נוסף על כך, ה-README כולל מדריך תצורה (Configuration Guide) המסביר את קבצי הקונפיגורציה והפרמטרים הניתנים לכיול והשפעתם על פעולת המערכת, הנחיות לתרומת קוד (Contribution Guidelines) הכוללות תקני קוד וסגנון, ומידע על רישיון השימוש וייחוס לספריות צד שלישי ותורמים (License & Credits).

4.2 מבנה פרויקט מודולרי

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

גודל הקבצים הוא שיקול חשוב בארגון הפרויקט. קבצי קוד לא צריכים לעלות על כ-150 שורות, מה שמבטיח 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 איכות קוד והערות

איכות 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 התכלית, הפרמטרים וערכי ההחזרה. ההערות צריכות להסביר החלטות עיצוב מורכבות, לתעד הנחות ותנאים מוקדמים, ולהתעדכן יחד עם שינויי הקוד. עקרונות כתיבת קוד איכותי כוללים שימוש בשמות משתנים ופונקציות תיאוריים ומדויקים, כתיבת פונקציות קצרות וממוקדות העומדות בעיקרון האחריות היחידה (sin- gle responsibility), הימנעות מקוד כפול לפי עיקרון 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 בעבודה 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 `.json`, `.yaml` או `.env`, תוך הימנעות מקבועים (`hardcoded values`) בתוך הקוד עצמו. חשוב לספק קבצי דוגמה כגון `env.example` עם ברירות מחדל ששומרות 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 רגישים למאגר הקוד. כמו כן, מומלץ ליצור קבצי `template` לקונפיגורציה עבור סביבות שונות כגון פיתוח (`dev`), ביניים (`staging`) וייצור (`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], [16]. הכלל המרכזי הוא שאסור בהחלט לשמור מפתחות API בקוד המקור עצמו. במקום זאת, יש להשתמש אך ורק במשתני סביבה (`environment variables`), כפי שמומחש בקוד לדוגמה: `os.environ.get("API_KEY")`. יש להסתיר את קבצי `.env` באמצעות `.gitignore`, ובסביבות ייצור להשתמש בכלי ניהול סודות (`secrets man-` `agement tools`) מתקדמים. נוסף על הגנה בסיסית, חשוב ליישם החלפה תקופתית של מפתחות (`rotation`), לנטר שימוש במפתחות ה-API כדי לזהות חריגות, ולהגביל הרשאות 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 טובה 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% לקוד חדש, עם דגש על כיסוי מוגבר לקוד קריטי ולוגיקה עסקית מרכזית. הבדיקות צריכות לכסות לא רק את המסלולים הרגילים אלא גם מקרי קיצון (`edge cases`) ותנאי גבול. סוגי הכיסוי הנדרשים כוללים כיסוי הצהרות (`Statement coverage`) המבטיח שכל שורת קוד הורצה לפחות פעם אחת, כיסוי ענפים (`Branch coverage`) המבטיח שכל 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-CI/CD pipeline, ולייצר דוחות כיסוי (coverage reports) המאפשרים מעקב אחר איכות הבדיקות.

6.2 טיפול במקרי קיצון ותקלות

זיהוי ותיעוד מקרי קיצון (Edge Cases) הוא חלק חיוני מפיתוח תוכנה איכותי. יש לזהות באופן שיטתי תנאי גבול ומקרי קיצון, לתעד כל מקרה עם תיאור מפורט של הקלט הצפוי והתגובה הנדרשת, ולכלול צילומי מסך של תקלות כאשר זה רלוונטי. מנגנוני טיפול בשגיאות (Error Handling) צריכים לכלול תכנות הגנתי (defensive programming) עם בדיקות קלט מקיפות, הודעות שגיאה ברורות ומועילות למשתמש, רישום לוגים מפורט לצורך ניפוי שגיאות (debugging), והידרדרות חלקה (graceful degradation) במקרי כשל כך שהמערכת ממשיכה לתפקד במידת האפשר.

תיעוד התקלות צריך לכלול תיאור מדויק של התקלה והסיבה לה, תיאור של תגובת המערכת וכיצד 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 השוואה מהירה בין התנהגות בפועל לבין ההתנהגות המצופה. יש לתעד את תוצאות ההרצה הצפויות לכל בדיקה, לייצר דוחות automated testing עם pass/fail rates המציגים את שיעור ההצלחה, ולשמור לוגים של הרצות מוצלחות וכושלות לניתוח עתידי ולמידה מטעויות.

7 מחקר וניתוח תוצאות

מחקר מעמיק וניתוח תוצאות הם מה שמבדיל בין פרויקט תוכנה רגיל לבין עבודה אקדמית ברמת מצוינות. החלק המחקרי כולל ניסויים שיטתיים, ניתוח כמותי ואיכותי, והצגה ויזואלית של התוצאות.

7.1 חקר פרמטרים

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

תיעוד הניסויים כולל יצירת טבלה מסודרת של כל הניסויים עם ערכי הפרמטרים והתוצאות המתאימות, הפקת גרפים ממחישים כגון line charts למגמות, heatmaps לרגישות

בין-פרמטרית, ו-sensitivity plots להמחשת השפעות, וביצוע ניתוח סטטיסטי של התוצאות לקביעת 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 מתודי ושיטתי של תוצאות הניסויים, השוואה בין אלגוריתמים שונים, תצורות או גישות מתודולוגיות, והכללת הוכחות מתמטיות או ניתוחים תיאורטיים כאשר זה רלוונטי.

הכללת נוסחאות ומתמטיקה נעשית באמצעות LaTeX לכתובת משוואות ונוסחאות מקצועיות, הסברים מתמטיים מפורטים למודלים ואלגוריתמים הכוללים את העקרונות התיאורטיים, ואסמכתאות לספרות אקדמית ומחקרים קודמים המקנים אמינות ומקצועיות לעבודה.

7.3 הצגה ויזואלית של תוצאות

ויזואליזציה איכותית של נתונים היא חיונית להעברת המסר המחקרי בצורה ברורה ומשכנעת. סוגי הויזואליזציות כוללים Bar charts להשוואות קטגוריות המאפשרות השוואה מהירה בין אפשרויות שונות, Line charts למגמות לאורך זמן המראות שינויים והתפתחויות, Scatter plots לזיהוי מתאמים וקשרים בין משתנים, Heatmaps להצגת רגישות פרמטרים בשתי ממדים, Box plots להצגת התפלגויות ומדדים סטטיסטיים, ו-Waterfall charts לניתוח שינויים רציפים ותרומות יחסיות.

איכות הגרפים נמדדת בבהירות ובדיוק התוויות, בשימוש בצבעים עקביים ונגישים שמתאימים גם לאנשים עם לקויות ראייה, בכללת כיתובים (captions) מפורטים ומקרא (leg-ends) ברור, וברזולוציה גבוהה המתאימה לפרסומים אקדמיים או מקצועיים. ככל שהגרפים מקצועיים וברורים יותר, כך ההשפעה של המחקר גדולה יותר.

8 ממשק משתמש וחווית משתמש

ממשק משתמש (User Interface - UI) וחווית משתמש (User Experience - UX) טובים הם קריטיים להצלחת כל מערכת תוכנה. אפילו מערכת עם פונקציונליות מעולה עלולה להיכשל אם המשתמשים מתקשים להשתמש בה.

8.1 קריטריוני איכות

קריטריוני השימושיות (Usability Criteria) [19], [20] כוללים מספר מימדים חשובים. קלות למידה (Learnability) מודדת עד כמה קל למשתמשים חדשים ללמוד להשתמש במערכת, 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 המשתמשים נהנים מהעבודה עם המערכת.

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

8.2 תיעוד ממשק

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

9 ניהול גרסאות ותיעוד פיתוח

ניהול גרסאות נכון הוא חיוני לעבודת צוות, למעקב אחר שינויים, ולאפשרות לחזור לגרסאות קודמות במקרה הצורך. בנוסף, תיעוד תהליך הפיתוח עוזר להבין את ההחלטות שנתקבלו לאורך הדרך.

9.1 שיטות עבודה מומלצות עם tiG

שיטות עבודה מומלצות (Git Best Practices) כוללות שמירה על היסטוריית commits ברורה עם הודעות משמעותיות המתארות מה השתנה ולמה, שימוש ב-branches נפרדים לפיתוח תכונות חדשות כדי לשמור 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 תהליך הפיתוח עם בינה מלאכותית (Prompt Engineering Log) הוא חלק חדש וחשוב בפיתוח תוכנה מודרני. התיעוד כולל רשימה של כל הפרומפטים המשמעותיים ששימשו לבניית הפרויקט, תיאור של ההקשר והמטרה של כל פרומפט, דוגמאות לפלטים שהתקבלו ואיך הם שולבו בפרויקט, תיעוד של שיפורים איטרטיביים של פרומפטים לאורך זמן, ושיטות

עבודה מומלצות (best practices) שהופקו מהניסיון. מבנה מומלץ לתיקיית הפרומפטים כולל תיקיות נפרדות לפרומפטים של תכנון ארכיטקטורה, ייצור קוד, בדיקות, ותיעוד, עם קובץ סקירה כללי.

10 עלויות ותמחור

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

10.1 ניתוח עלויות

ניתוח עלויות (Cost Breakdown) של שימוש ב-API Tokens כולל ספירה מדויקת של tokens הן בכניסה והן ביציאה (input/output tokens), חישוב העלות למיליון 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

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

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

10.2 ניהול תקציב

ניהול תקציב יעיל כולל תחזית עלויות לסקאלה עתידית כדי להבין את העלויות הצפויות כאשר המערכת גדלה, ניטור (monitoring) של שימוש בזמן אמת כדי לזהות חריגות מוקדם, והגדרת התראות על חריגה מתקציב כדי למנוע הוצאות לא מתוכננות.

11 הרחבה ותחזוקתיות

תכנון מערכת שניתן להרחיב ולתחזק בקלות הוא השקעה לטווח ארוך שמשתלמת כאשר המערכת צריכה להתפתח ולהשתנות.

11.1 נקודות הרחבה

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

המגדירים את החוזה בין הליבה לתוספים, הוספת נקודות חיבור (lifecycle hooks) כגון beforeCreate, afterUpdate וכדומה שמאפשרות לתוספים להגיב לאירועים, שימוש במנגנוני middleware לעיבוד שרשרתי של בקשות, ועיצוב מבוסס-API (API-first design) שמבטיח שכל הפונקציונליות נגישה דרך ממשקים מוגדרים היטב. תיעוד ההרחבה 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 לתחזוקה (Maintainable Code) מאופיין במודולריות (Modularity) והפרדת אחריות (separation of concerns) כך שכל חלק בקוד אחראי על דבר אחד בלבד, שימוש חוזר (Reusability) של קומפוננטות כך שקוד כתוב פעם אחת ומשמש במקומות רבים, ניתנות לניתוח (Analyzability) כך שקל להבין את הקוד ולזהות בעיות, וניתנות לבדיקה (-Testability) כך שקל לכתוב בדיקות אוטומטיות לקוד.

12 תקני איכות בינלאומיים

תקן ISO/IEC 25010 [22] מגדיר מודל מקיף לאיכות תוכנה המכסה שמונה מאפייני איכות עיקריים. כל מאפיין מתחלק 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 התנהגות הזמן (Time behavior) כולל זמני תגובה, ניצול משאבים (Resource utilization) כולל זיכרון ומעבד, ויכולת (Capacity) להתמודד עם עומסים גדלים.

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

אבטחה (Security) כוללת סודיות (Confidentiality), שלמות (Integrity), אימות (Authenticity), אחריותיות (Accountability), ואי-הכחשה (Non-repudiation). תחזוקתיות (Maintainability) מעריכה מודולריות, שימוש חוזר, ניתנות לניתוח, ניתנות לשינוי (Modifiability), וניתנות לבדיקה. לבסוף, ניידות (Portability) בוחנת התאמה (Adaptability) לסביבות שונות, ניתנות להתקנה (Installability), וניתנות להחלפה (Replaceability).

13 רשימת בדיקה סופית

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

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

הויזואליזציה צריכה לכלול גרפים איכותיים של תוצאות, צילומי מסך של ממשק המשתמש, ותרשימי ארכיטקטורה ברורים. ניתוח העלויות צריך לכלול טבלת שימוש ב-tokens, ניתוח עלויות מפורט, ואסטרטגיות אופטימיזציה. ההרחבה צריכה לכלול נקודות הרחבה (extension points) מתועדות, דוגמאות לפיתוח תוספים (plugins), וממשקים ברורים להרחבה.

13.1 בדיקה טכנית מפורטת

נוסף על הקריטריונים לעיל, יש לוודא עמידה בקריטריונים הטכניים המפורטים בפרקים 13-15:

- ארגון הפרויקט כחבילה עם קבצי `pyproject.toml/setup.py`, קבצי `__init__.py`, ומבנה תיקיות מאורגן

- שימוש נכון במעבדים מרובים (multiprocessing) לפעולות CPU-bound

- שימוש נכון בחוטי ביצוע (multithreading) לפעולות I/O-bound, 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 בניה והתלויות שלה

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

14 מקורות ותקנים נוספים

לצורך הכנת פרויקט ברמת מצוינות, מומלץ להתייחס לתקנים ומקורות בינלאומיים מוכרים. אלה כוללים את תכנית אבטחת איכות התוכנה של MIT [23], מודל איכות התוכנה ISO/IEC 25010 [22], שיטות העבודה ההנדסיות של Google [24], ההנחיות ל-API של Microsoft [25], וההיוריסטיקות לשימושיות של נילסן [21]. מקורות אלה מספקים בסיס מוצק לעבודה מקצועית ואקדמית ברמה הגבוהה ביותר.

15 ארגון הפרויקט כחבילה

ארגון הקוד כחבילה (package) הוא עיקרון יסוד בפיתוח תוכנה מקצועי. חבילה מאורגנת נכון מאפשרת שימוש חוזר בקוד במספר פרויקטים, ניהול תלויות (dependencies) בצורה ברורה, הפצה והתקנה פשוטה, ובדיקות (testing) מובנות. פרק זה מספק רשימת בדיקה מפורטת לארגון הפרויקט כחבילה Python מקצועית.

15.1 קובץ הגדרת החבילה

כל חבילה מקצועית חייבת לכלול קובץ הגדרה המפרט את מאפייני החבילה ותלויותיה. ניתן להשתמש בקובץ `setup.py` המסורתי או בקובץ `pyproject.toml` המודרני יותר. קובץ ההגדרה צריך לכלול את שם החבילה, מספר הגרסה, תיאור קצר, שם המחבר, רישיון השימוש, ורשימה מלאה של כל התלויות החיצוניות עם מספרי גרסאות ספציפיים או טווחי גרסאות מקובלים.

15.2 קבצי `__init__.py`

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

15.3 מבנה תיקיות מאורגן

ארגון התיקיות צריך להיות לוגי ועקבי. קוד המקור צריך להימצא בתיקייה ייעודית, בדרך כלל `src/` או בתיקייה הנושאת את שם החבילה. הבדיקות צריכות להימצא בתיקייה נפרדת בשם `tests/`, והתיעוד בתיקייה נפרדת בשם `docs/`. הפרדה זו מבטיחה שקוד הייצור, קוד הבדיקות, והתיעוד לא מעורבבים זה בזה, מה שמקל על תחזוקה וניווט בפרויקט.

15.4 שימוש בנתיבים יחסיים

כל הייבואים (imports) בקוד צריכים להשתמש בנתיבים יחסיים או בשמות חבילות, ולעולם לא בנתיבים מוחלטים. למשל, במקום לכתוב `import /path/to/module`, יש להשתמש ב-`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 תעבוד בכל סביבה ללא תלות במבנה הספציפי של מערכת הקבצים.

15.5 רשימת בדיקה: ארגון כחבילה

עברו 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 `__init__.py`:

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

3. מבנה תיקיות:

- האם קוד המקור נמצא בתיקייה ייעודית?
- האם הבדיקות נמצאות בתיקייה נפרדת `tests/`?
- האם התיעוד נמצא בתיקייה נפרדת `docs/`?

4. נתיבים יחסיים:

- האם כל הייבואים 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) ובחטי ביצוע מרובים (multithreading) הוא חיוני לביצועים אופטימליים של תוכנה מודרנית. הבנת השימוש הנכון בכלים אלו היא קריטית לפיתוח מערכות יעילות שמנצלות את מלוא יכולות החומרה המודרנית.

16.1 הבדל בין Multiprocessing ל-Multithreading

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 אמיתית.

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

16.2 שימוש ב-Multiprocessing

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

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

16.3 שימוש ב-Multithreading

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

16.4 בטיחות חוטים

בטיחות חוטים (thread safety) היא היבט קריטי בעבודה עם multithreading. יש להימנע ממצבי תחרות על ידי הגנה על כל גישה למשתנים משותפים באמצעות מנעולים. יש להשתמש במבני נתונים בטוחים לחוטים כמו Queue, queue. כאשר מעבירים מידע בין חוטים. חשוב גם להימנע מנעילה הדדית על ידי שמירה על סדר קבוע של נעילה ושחרור מנעולים, ושימוש ב-context managers (with statement) להבטחת שחרור אוטומטי של מנעולים.

16.5 רשימת בדיקה: עיבוד מקבילי

עברו 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 הערכתם את התועלת הפוטנציאלית מביצוע מקבילי?

2. יישום נכון:

- האם מספר התהליכים/חוטים מוגדר דינמית?
- האם שיתוף נתונים מבוצע בצורה בטוחה?
- האם קיים סנכרון נכון בין יחידות הביצוע?

3. ניהול משאבים:

- האם התהליכים/חוטים נסגרים כראוי?
- האם קיים טיפול בחריגות (exceptions)?
- האם נמנעים מזליגת זיכרון?

4. בטיחות (ל-multithreading):

- האם משתנים משותפים מוגנים באמצעות מנעולים?
- האם נמנעים ממצבי תחרות?
- האם נמנעים מנעילה 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 ממשק מוגדר היטב. גישה זו מקלה על תחזוקה, בדיקה, ושימוש חוזר בקוד, ומאפשרת פיתוח מערכות גדולות ומורכבות בצורה מסודרת ומובנית.

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 החיצוניות. כל נתוני הקלט חייבים לעבור ולידציה מקיפה לפני השימוש.

נתוני פלט (Output Data) – התוצרים שהאבן מייצרת. יש להגדיר את סוגי הנתונים, את הפורמט של הפלט, ואת התנהגות הפלט במקרי קצה ושגיאות. הפלט צריך להיות עקבי ומוגדר היטב.

נתוני הגדרה (Setup Data) – פרמטרים וקונפיגורציה לאבן הבניה. אלו כוללים פרמטרים קונפיגורציוניים 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, תחזוקה, ובדיקה.

הפרדת 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. התלויות מסופקות דרך הזרקת תלות (dependency injection).

17.3 ולידציה והגנה

כל אבן בניה צריכה לכלול ולידציה מקיפה של נתוני הקלט. יש לבדוק את סוגי הנתונים, את התחום התקף, ואת התנאים המוקדמים. במקרה של קלט לא תקין, יש להחזיר הודעת שגיאה ברורה ומועילה שמסבירה מה השתבש ומה צריך לתקן. הבדיקות צריכות להתבצע מוקדם ככל האפשר (fail fast) כדי למנוע התפשטות שגיאות למערכת.

17.4 רשימת בדיקה: עיצוב אבני בניין

עברו על הפריטים הבאים ובדקו את הפרויקט שלכם:

1. זיהוי אבני בניין:

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

2. נתוני קלט:

- האם כל נתוני הקלט מתועדים ברורה?
- האם קיימת ולידציה לכל נתוני הקלט?
- האם התלויות מסופקות דרך הזרקת תלות?

3. נתוני פלט:

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

4. נתוני הגדרה:

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

5. עקרונות עיצוב:

- האם כל אבן בניה עומדת בעיקרון האחריות היחידה?
- האם קיימת הפרדת דאגות ברורה?
- האם אבני הבניה ניתנות לשימוש חוזר ולבדיקה?

17.5 דוגמה לאבן בניה טובה

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

מבנה האבן:

- **נתוני קלט** – רשימת מילונים (raw_data) וקריטריוני סינון (filter_criteria)

- **נתוני פלט** – רשימת מילונים 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. **הפרדת דאגות** – קונפיגורציה נפרדת מלוגיקת 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

    Input Data:
    - raw_data: List[Dict] - list of dictionaries
    - filter_criteria: Dict - filtering criteria

    Output Data:
    - processed_data: List[Dict] - processed dictionaries

    Setup Data:
    - 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())
```

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 הערה חשובה

מסמך זה מציג רמת מצוינות גבוהה במיוחד. לא כל סעיף הוא מחויב במלואו, אך ככל שיותר קריטריונים מתקיימים, כך הציון והערכת האיכות יהיו גבוהים יותר. התמקדו בעומק, במקצועיות ובהדגמת יכולות מחקר ברמה אקדמית גבוהה. מומלץ להשתמש בכלי LLM לעזרה בהשלמת הפרויקט. מובהר כי כחלק מהבדיקה יתכן ויעשה שימוש בסוכני AI לביצוע הבדיקה.

18.1 הערכה משולבת

הערכת הפרויקט צריכה לשלב בין הקריטריונים האקדמיים (פרקים 1-12) לבין הקריטריונים הטכניים (פרקים 13-15). שקלול מומלץ הוא 60% קריטריונים אקדמיים ו-40% קריטריונים טכניים. גישה משולבת זו מבטיחה שהפרויקט עומד לא רק בסטנדרטים מחקריים אלא גם בתקני הנדסת 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.