

פרק 1

מבוא

כיצד בונים מערכת תוכנה גדולה ומורכבת, כגון מערכת לניהול חשבונות בנק או מערכת לניהול של בית ספר? מתכנת מומחה יתכן ויבנה מערכת המורכבת מכמה יחידות, בלתי תלויות זו בזו עד כמה שאפשר, שכל אחת אחראית על משימה משלה. היחידות ישתפו פעולה כדי לממש את מטרות המערכת.

גישה זו לתכנון ובנייה של מערכות תוכנה עומדת ביסודה של הגישה הנקראת: **תכנות מונחה עצמים** (ובקיצור תמ"ע או OOP). הנחת היסוד של גישה זו היא שאת ביצוע המשימה הגדולה שתוכנית צריכה לבצע ניתן לחלק בין ישויות קטנות יותר. ישות שכזו מיוצגת על ידי יחידת תוכנה המכונה **עצם** (object). עצם הוא יחידת תוכנה סגורה ועצמאית היכולה לבצע פעולות מסוימות. מערכת התוכנה השלמה היא אוסף של עצמים המתקשרים ביניהם ופועלים אלה עם אלה, בדרך כלל בעזרת תוכנית מנחה.

בעשורים האחרונים התפתח רעיון חשוב שעוזר בבניית מערכות תוכנה, הנקרא **שימוש חוזר** (code reuse) ביחידות תוכנה, שנכונותן ושימושיותן הוכחו ביישומים שונים. כיוון שיש צרכים המופיעים כמעט בכל מערכת, אין סיבה לחזור ולפתח פתרונות לצרכים אלה בכל פעם מחדש, אם אלה כבר פותחו בעבר ונבחנו בשימוש בשטח. שפות מונחות עצמים מודרניות מכילות ספריות תוכנה עשירות העונות על צרכים רבים כאלה.

ביחידת לימוד זו נלמד מעט על העקרונות המנחים בניית מערכת תוכנה. כמו כן, נעסוק בסוג מסוים של עצמים – יחידות תוכנה – המאפשרים לטפל באוספי נתונים גדולים. זהו צורך בסיסי כמעט בכל מערכת תוכנה גדולה. נשתמש בשפה מונחת עצמים כדי לממש עצמים כאלה, ונממש גם מערכות קטנות המשתמשות בהם.

א. מערכות תוכנה

מערכות תוכנה (software systems) הן לעתים קרובות תוכניות גדולות מאוד, הכוללות מאות אלפי שורות קוד. הן מטפלות במגוון רחב של נושאים, בכל התחומים שבהם החברה המודרנית מתעסקת. נזכיר לדוגמה מעבדי תמלילים, מערכות בנקאיות, מערכות תקשורת (כגון מערכת האינטרנט) ומשחקי מחשב. ראוי להזכיר גם מערכות תוכנה המשובצות במערכות אחרות: מערכת הטסה של מטוס קרב, מערכת לשינוע מזוודות בנמל תעופה, מערכת להפעלת מכשיר כספומט, מערכת האחראית על הפעלת בלמים ברכב וכן הלאה. אפילו מכשירים פשוטים כביכול, כגון טלפון סלולרי, מכילים בתוכם מערכת תוכנה האחראית על ביצוע הפעולות הנדרשות על ידי המפעיל.

בשל חשיבותן של מערכות תוכנה בכל תחומי החיים, ובשל גודלן ומורכבותן, מושקע בפיתוחן מאמץ רב. אולם, השקעת מאמץ כשלעצמה אינה ערובה לאיכות. מאמץ הפיתוח מונחה על ידי אוסף של דרישות מתהליך התכנון ובניית המערכת ועל ידי קיום דרישות אלה במהלך הפיתוח.

נפרט כמה מדרישות אלה :

מפרט מדויק: שלב ראשון בפיתוח מערכת הוא כתיבת מפרט מפורט המפרט מהם תפקידי המערכת, כיצד היא אמורה לפעול וכן הלאה. במהלך הפיתוח, מפרט זה מתעדכן בהתאם לצורך, ובסיום הפיתוח נוצר ממנו תיעוד מפורט, מודפס או אינטראקטיבי, של המערכת, של תפקידיה ושל דרכי הפעלתה, וכן תיאור של מגבלותיה. בדרך כלל כולל התיעוד גם הסבר על דרישות ההתקנה וההפעלה מבחינת חומרה ותוכנה. המפרט, כפי שהוא מובע בתיעוד, משמש כ"חוזה" הפעלה: אם משתמשים במערכת בהתאם להוראות שבמפרט, היא תבצע את תפקידיה כפי שהוא מוגדר בו.

נכונות: אין ערך לחוזה אם אחד הצדדים אינו מקיים את הכתוב בו. המפרט מועיל לשימוש במערכת כאשר היא "עובדת נכון", כלומר מתנהגת בהתאם למתואר בו. נכונות של מערכת תוכנה היא כמובן דרישה מרכזית, ומערכת שאינה מקיימת אותה אינה קבילה. אולם, הבטחת נכונות של מערכת תוכנה ידועה כמשימה קשה, בייחוד במערכות גדולות ומורכבות. חברות תוכנה משקיעות מאמצים רבים ומעסיקות עובדים רבים בטיפול בנכונות, ולמרות זאת, מעטות הן מערכות התוכנה שבהן לא חוזרות ומתגלות תקלות, אפילו לאחר זמן רב של שימוש.

כמובן שככל שהמערכת רגישה יותר, כך משקיעים יותר בשיפור נכונותה: לא יעלה על הדעת כי מערכת תוכנה האחראית על הטסת מטוס קרב תפעל נכון **כמעט** תמיד – היא חייבת לפעול נכון בכל מצב אפשרי. לעומת זאת, תוכנות רבות הנמצאות בשימוש במחשבים אישיים פועלות נכון בדרך כלל, אך לא תמיד.

הגישה המונחית עצמים מאפשרת גישה מובנית לטיפול בנכונות תוכנה. כיוון שעצם הוא יחידה כמעט עצמאית, ניתן בדרך כלל לבדוק אותו היטב, לפני שמשלבים אותו במערכת. גישה אחרת לנכונות, שאותה הזכרנו, היא **שימוש חוזר** בקוד שכבר נעשה בו שימוש בכמה מערכות. ככל שמספר המערכות המשתמשות ביחידת קוד מסוימת גדל, כך גדל ביטחוננו בנכונותה של היחידה. שילוב יחידות קוד קיימות במערכת חדשה מקל על הבטחת הנכונות, וגם חוסך זמן פיתוח רב.

עמידות במצבים חריגים: מערכת תוכנה נדרשת לעתים להתמודד עם מצבים חריגים. עמידות מתייחסת ליכולת המערכת להתמודד גם עם מצבים אלה. לדוגמה, כיצד תתנהג מרכזית טלפונים, המיועדת לטפל ב-1,000 שיחות בו-זמנית, אם בנקודת זמן מסוימת יחליטו דווקא 1001 מנויים לשוחח בטלפון? האם המרכזייה תקרוס ותפסיק כליל את השירות לכל הלקוחות, או שתספק שירות סביר לרובם?

ידידותיות למשתמש: כאשר מערכת היא פשוטה וקלה להפעלה, אנו מכנים אותה "ידידותית למשתמש". לחלק המערכת המטפל בקשר עם המשתמש קוראים **ממשק המשתמש (user interface)**. ככל שהוא קל ונוח להפעלה, כך צוברת מערכת התוכנה נקודות זכות בקרב קהל היעד שלה – המשתמשים. עבודה עם תוכנית מחשב שאינה ידידותית, כמו תוכנית המצפה לקלט מבלי להסביר מהו, היא ללא ספק חוויה מתסכלת.

מהירות תגובה: פרק הזמן מזימון פעולה ועד לסיום ביצועה הוא המדד למהירות תגובה. ממערכת תוכנה דורשים מהירות תגובה שתתאים להגדרת תפקידיה. לדוגמה, מערכת ניטור בחדר טיפול נמרץ אמורה להציג את המידע שהיא מעבדת ללא שהיות (או לכל היותר בשהות של

שברירי שנייה). ממערכת לביצוע פעולות בנקאיות (כספומט) נצפה לזמן תגובה של שניות בודדות עד עשרות שניות. בשימוש במערכת לעיבוד סטטיסטי של שאלוני סקר נסתפק בהזנת הנתונים, וקבלת תוצאות לאחר שעה, או אפילו לאחר לילה, כלומר למחרת היום.

תמיכה בתחזוקה ובעדכון: מערכת תוכנה, כמו כל מוצר מורכב, דורשת תחזוקה (maintenance). הצורך בתחזוקה נובע הן מגילוי פגמים בתוכנה במהלך השימוש בה, והן מדרישות של משתמשים לשיפור יכולותיה. תחזוקת מערכת תוכנה פירושה ביצוע שינויים בקוד. כאשר מדובר בקוד בהיקף גדול, זו משימה קשה, לעתים אף קשה יותר מהמשימה המקורית של פיתוח הקוד, שכן יש להבין היטב את הקוד לפני ביצוע שינויים בו. ידוע שרוב ההוצאות הכספיות על מערכות תוכנה קשורות לתחזוקה, ולא לפיתוח המקורי. הקפדה על כללי תכנון ותכנות נכונים ועל תיעוד מדויק בעת כתיבת התוכנית עוזרת מאוד לתחזק אותה בקלות ולשנותה לפי הצורך.

מה נוכל להסיק מתיאור כל הדרישות הללו ממערכות תוכנה? ביחידה זו לא נבנה מערכות גדולות, ואף לא נזדקק לתחזוקת מערכות. עבורנו משמעותית מסקנה אחת העולה מתיאור הדרישות, והיא החשיבות של רכישת הרגלים נכונים של תכנון תוכנה, של תכנות שיוצר תוכנה בעלת מבנה ברור וקל להבנה, ושל תיעוד מלא ומדויק. מסקנה נוספת היא שחשוב להבין את המבנים בשפת התכנות וכן את הגישות לתכנות המאפשרים כולם יחד בניית תוכנה העונה על מכלול הדרישות שהוצגו. ביחידה זו נעשה צעדים ראשוניים בכיוונים אלה.

ב. מה לפנינו?

נסקור עתה בקיצור אחדים מהנושאים החשובים שנעסוק בהם ביחידה זו.

1.1. תכנות מודולרי ויישומי בתכנות מונחה עצמים

בסעיף הקודם אמרנו כי מערכות תוכנה הן בדרך כלל גדולות ומסובכות. כדי לעמוד בדרישות שמנינו, התגבשה בקהילת מהנדסי התוכנה גישה להתמודדות עם האתגר של בניית מערכות כאלה, הנקראת "הפרד ומשול". הכוונה היא לחלוקת המשימה של תכנות המערכת הגדולה לתת-משימות, וכתובת תת-מערכת עבור כל תת-משימה. כאשר חוזרים על תהליך זה, מתפרקת המשימה הראשונה באופן היררכי לתת-משימות. כך ניתן להתמודד ישירות עם כל תת-משימה. יחידת תוכנה עבור תת-משימה קטנה כזו נקראת לעתים **מודול (module)**, וארגון תוכנה כאוסף מודולים נקרא **תכנות מודולרי**.

לגישה זו יתרונות רבים. היא מאפשרת חלוקת העבודה על מערכת גדולה בין כמה צוותים, כך שכל צוות מקבל תת-משימה מוגדרת היטב. הקטנת המטלה המוטלת על כל צוות מאפשרת בנייה מהירה יותר של פתרון מתאים. גישה זו טובה מאוד גם למתכנת הבודד, מאחר שהיא מחלקת את משימת התכנות לתת-משימות מוגדרות היטב. ארגון מודולרי של תוכנה מקל גם על בדיקות נכונות ועל תחזוקה. כאשר מתגלה שגיאה, קל בדרך כלל לשייכה לאחד מהמודולים. צמצום התחום שבו נגרמת השגיאה מאפשר זיהוי מהיר וקל יותר של סיבתה. תיקון השגיאה לעתים קרובות ייעשה על ידי שינוי מודול יחיד.

לאחר שהכירו בחשיבות של גישה זו לתכנות, הצעד הבא היה פיתוח שפות תכנות בהן קיימת תמיכה ברעיון המודול. אלה הן השפות המונחות עצמים. תוכנית בשפה כזו מורכבת ממחלקות. **מחלקה (class)** מגדירה מבנה וכן אוסף פעולות של עצמים. ניתן לייצר ממחלקה עצם אחד, חמישה או מאה עצמים – לכולם יהיה אותו המבנה ואותן הפעולות. בריצת תוכנית טיפוסית של שפה מונחית עצמים משתתפים עצמים שנוצרו ממחלקה אחת או יותר. העצמים מבצעים פעולות, מתקשרים זה לזה ומממשים יחד את משימת התוכנית. כל עצם אחראי למשימה קטנה ומוגדרת היטב.

תכנות מודולרי בשפה מונחית עצמים פירושו הגדרת מחלקות ומשימות של העצמים שייוצרו מהמחלקות. בפרקים הראשונים (פרק 2, 3 ו-6) נדון שוב, כהמשך לדיון בנושא זה ביחידות קודמות, במחלקות, בעצמים ובקשר ביניהם.

ב.2. טיפול באוספי נתונים

הניסיון הנצבר בתכנות מערכות גדולות מלמד לא רק על התועלת שבתכנות מודולרי, אלא גם מספק הנחיה כיצד לבצע פירוק מודולרי. הנחיה זו מדגישה את חשיבות הנתונים והטיפול בהם (זאת, בניגוד לצעדים הראשונים בתכנות המדגישים את רעיון האלגוריתמים ומימושם בקוד). מערכות תוכנה גדולות מטפלות בנתונים רבים, מסוגים רבים ושונים, ופועלות עליהם שוב ושוב. כאשר מרכזים יחד, בעצמים מתאימים, נתונים ופעולות שימושיות עליהם, מקבלים בסיס טוב לארגון המערכת כולה. משום כך, עצם טיפוסי במערכת הבנויה בשפה מונחית עצמים מורכב מנתונים ומאוסף פעולות שימושיות עליהם. מתכנת, הלומד להגדיר מחלקות, לומד למעשה כיצד להגדיר את הנתונים ואת הפעולות עליהם. האריזה יחד של נתונים ופעולות מכונה **הכמסה (encapsulation)**, והיא עיקרון חשוב בתכנות מונחה עצמים.

כאשר עוסקים בנתונים, יש מקרה הראוי לתשומת לב מיוחדת, והוא הטיפול באוספים דינמיים של נתונים. הדוגמאות לאוספים שאנו רוצים לייצגם בתוכניות שלנו הן רבות: אוסף הלקוחות של בנק, אוסף התנועות בחשבון, אוסף התלמידים בכיתה או בבית ספר, רשימת הספרים בספרייה, רשימת הכתובות והטלפונים בספר כתובות ממוחשב, ועוד ועוד. אוספים אלה הם דינמיים כיוון שהם יכולים לגדול, ולפעמים אף לקטון. ברוב המערכות הגדולות יש תת-מערכות העוסקות בטיפול באוספים אלה.

מספר סוגי האוספים משתמשים בהם במערכות אינו רב, ויש פעולות על אוספים שבהם משתמשים שוב ושוב: הכנסת פריט נוסף לאוסף, חיפוש פריט בעל תכונות מסוימות, סריקת כל האוסף, וכיוצא באלה. חייהם של בוני המערכות היו בוודאי קלים יותר אילו עמדה לרשותם ספרייה של יחידות תוכנה מוכנות המאפשרות לבנות אוספים כאלה ולטפל בהם. ביחידות אלה אפשר להשתמש שימוש חוזר בכל מערכת הזקוקה לטיפול באוספי נתונים. ואכן, בכל שפת תכנות מונחית עצמים מודרנית יש ספרייה כזו.

הטיפול באוספים הוא נושא מרכזי ביחידה זו, ובו נעסוק מפרק 7 והלאה. לא נלמד להשתמש בספרייה המוכנה שהשפה מעמידה לרשותנו, אלא נבנה בעצמנו מחלקות עבור סוגי אוספים נפוצים. סיבה אחת לכך היא שאחת ממטרות היחידה היא הרחבה והעמקה בתכנות, כדי לשפר

את היכולת התכנותית. מעבר לכך, יכולת תכנות טובה הכרחית גם לשימוש מושכל ונכון במחלקות שבספרייה המוכנה. רק מי שטרח וניסה לבנות בעצמו מחלקות לטיפול בכמה סוגי אוספים יוכל להעריך את האיכות ואת הכלליות של מחלקות הספרייה, וידע לנצלן כראוי.

3.3. הפרדה בין ממשק למימוש

עיקרון חשוב בתכנות מודולרי הוא הפרדה בין ממשק למימוש. בשפה מונחית עצמים המחלקה היא מימוש רעיון המודול. המפרט של מחלקה – הכולל את הגדרתה, את הגדרת הפעולות הכלולות בה ואת דרך ההפעלה שלהן – נקרא בשם **ממשק (interface)** המחלקה. המידע הכלול בממשק זה הוא כל מה שצריך לדעת המשתמש במחלקה – הלקוח – כדי להפעיל את העצמים שלה. המילה "לקוח" כאן פירושה כפול: כל עצם המשתמש בעצם של המחלקה, וגם כל מתכנת של מחלקה אחרת שממנה נוצרים עצמים כאלה. השימוש במונח "לקוח" רומז גם לכך שהממשק הוא חוזה בין המחלקה למשתמשיה. **המימוש (implementation)** מתייחס הן למבנה הפנימי של העצמים של המחלקה, והן לקוד המממש את הפעולות.

כדוגמה פשוטה, ממשק של פעולה יכול לקבוע כי היא מחזירה שורש שלישי של מספר המועבר לה כפרמטר. המימוש הוא הקוד של הפעולה. ברור שמימוש הפעולה חשוב מאוד – אי אפשר לזמן פעולה שאין לה מימוש. אולם כדי שנוכל להשתמש בפעולה, עלינו לדעת רק מה היא מבצעת וכיצד מפעילים אותה, כלומר מהו הממשק שלה. הדבר דומה לאופן ההפעלה של טלפון סלולרי. כדי להתקשר למספר מסוים מהמכשיר הנייד שברשותנו, עלינו לדעת לאתר מספר טלפון השמור בזיכרון ולהכיר את לחצן ה-send. אין לנו צורך ובדרך כלל אין לנו אף יכולת לדעת כיצד ממומשת פעולת האיתור, ומהו המנגנון שבבסיס פעולת הלחצן.

להפרדה בין ממשק למימוש מתייחסים גם במונח **הסתרת מידע (information hiding)**. המימוש, כלומר המבנה הפנימי של המחלקה וקוד הפעולות, מוסתר מהלקוח והוא ידוע רק לכותב הקוד של המחלקה עצמה.

לדרישת ההפרדה כמה סיבות. הסיבה הראשונה נוגעת להגדרת המשימה של המחלקה. נניח שאתם נדרשים להגדיר בעיה. אם ההגדרה שהגעתם אליה כוללת גם רכיבים של הפתרון, כנראה שהבנת הבעיה והיכולת לנסח כראוי לוקות בחסר. נוחו קצת ונסו שנית. באופן דומה, הגדרת תת-משימה של מערכת צריכה להיות מנוסחת במונחים שאינם תלויים במימוש מסוים של המשימה. רק הגדרה כזו תאפשר לבחון את כל הגישות האפשריות למימוש, ומהן לבחור את זו שנראית לנו הטובה ביותר. מכאן שהגדרת מחלקה צריכה להיות בלתי תלויה במימוש שלה.

לדרישת ההפרדה יש סיבות נוספות. אם מחלקים משימת פיתוח מערכת בין כמה צוותים, ההסתמכות של כל צוות רק על הממשק של היחידות שכותבים הצוותים האחרים מקלה מאוד על החלוקה. למעשה, ללא הנחה זו, חלוקת פיתוח של מערכת בין כמה צוותים אינה מעשית, כיוון שכל פעם שצוות מסוים משנה את גישתו לפתרון, הצוותים האחרים צריכים להתחיל את עבודתם מחדש. כמו כן, כאשר כותבים קוד למערכת תוכנה, אנו מעוניינים כי שינויים ועדכונים עתידיים, הן לצורך החלפת פתרון בפתרון עדיף, והן לתיקון שגיאות, יהיו מקומיים עד כמה שאפשר. כלומר, אנו מעדיפים שינוי של קטע קוד מסוים לא יגרור אחריו שינויים במקומות רבים אחרים

בקוד. אם תוכנת לקוח המשתמשת במחלקה (ובעצמים הנוצרים ממנה) מסתמכת רק על הממשק, ולא על המימוש, אזי ניתן להחליף מימוש באחר או לתקן שגיאה בקוד, וזאת בלי שהדבר ידרוש שינוי כלשהו בתוכנת הלקוח. יש בזה יתרון גדול מאוד להקלת התחזוקה של מערכות תוכנה.

בנושא זה, של הפרדת ממשק ממימוש והסתרת המימוש נדון באופן שוטף לאורך היחידה. נחدد את הגדרת רעיון ההפרדה ונציג לכך מגוון דוגמאות. בדיונונו במחלקות לניהול אוספי נתונים נדון בפירוט במימוש רעיון זה במחלקות אלה. למרבה ההפתעה, לא בכל המחלקות נצליח להגיע למימוש מלא של הפרדה והסתרה. הרעיון נשמע פשוט, אך יישומו אינו בהכרח כה פשוט. חשוב לציין שבספרייה של שפת התכנות המחלקות לטיפול באוספים כולן מפרידות היטב בין ממשק למימוש, ומסתירות את המימוש באופן מלא. חוסר ההצלחה שלנו בחלק מהמקרים נובע מכך שמנגנונים מתקדמים יותר של השפה, המסייעים להפשטה ולהסתרה של מימושים, אינם כלולים ביחידה זו. בכל אופן, גם העיסוק במקרים שבהם אנו נכשלים מועיל להבנת אותם סוגי אוספים, והוא גם מדגים את הצורך במנגנוני שפה מתקדמים יותר.

ב.4. נושאים נוספים

נזכיר בקיצור כמה נושאים נוספים שנדונים ביחידה זו.

כדי לאפשר הפרדה מלאה בין ממשק למימוש, הכרחי שלמחלקה יהיה **תיעוד (documentation)** ברור ומלא, המתאר את המחלקה ואת פעולותיה. תיעוד כזה הוא תוצר של תכנון ראוי של המחלקה והגדרת מטרתה. כאשר קיים תיעוד כזה, המשתמש במחלקה יכול לכתוב את התוכניות המשתמשות במחלקה תוך הסתמכות על הממשק בלבד. יצירת תיעוד כזה למחלקות היא רכיב חשוב בתכנות מונחה עצמים, ואנו נקדיש לו תשומת לב רבה.

הזכרנו את הדרישה למהירות תגובה סבירה של מערכת תוכנה. זו קשורה קשר הדוק ל**יעילות (efficiency)** האלגוריתמים שהמערכת משתמשת בהם. נעמיק עוד בנושא זה בפרק 5. יתרה מזאת, לאורך כל היחידה נדון ביעילות של אלגוריתמים שבהם נעסוק.

לאורך היחידה נעסוק באופן שוטף בלימוד כתיבה נכונה של קוד, כדי שזה יהיה קריא ומאורגן היטב. כמו כן, נעסוק בהבטחת **נכונות (correctness)** של קטעי התוכנה שנכתוב. הבטחת נכונות של מחלקה קשורה קודם כול בהבנה טובה של המבנה ובמימוש שלה. בנוסף, לאחר כתיבת המחלקה, יש לבדוק כל פעולה שלה על פרמטרים במגוון ערכים. בבדיקת נכונות של מחלקות ופעולותיהן יש להקדיש תשומת לב מיוחדת למצבים הקרויים **מקרי קצה**. לדוגמה, אם כתבנו קטע קוד המקבל רשימת ציוני תלמידים ומחשב את הממוצע, ראוי לבדוק, הן על ידי ניתוח האלגוריתם, והן בעזרת הרצה של התוכנית, אם הפעולה נכונה גם עבור רשימה המכילה ציון יחיד, ואפילו עבור רשימה ריקה, שאינה מכילה אף ציון. אלה דוגמאות של מקרי קצה.

לסיים, סוף מעשה במחשבה תחילה הוא החוט שיהיה שזור לאורך תהליך הלמידה של תכנות מונחה עצמים, החל בפרק מבוא זה ולאורך כל היחידה. מחשבה ותכנון לפני כתיבת תוכנית בהיקף כלשהו אינה גורעת מאיכות התוצאות. ההפך הוא הנכון – איכות התוצאות משתפרת ככל שמוקדש זמן רב יותר לשלב של תכנון המערכת.

ג. סיכום

- מערכות תוכנה הן תוכניות גדולות מאוד המבצעות תפקידים רבים ומורכבים ומטפלות במגוון רחב של קלטים.
- דרישות האיכות ממערכות תוכנה הן רבות, ומתמקדות בתחומים האלה:
 - מפרט מדויק
 - נכונות
 - עמידות במצבים חריגים
 - מהירות תגובה
 - ידידותיות למשתמש
 - תמיכה בתחזוקה ובעדכון
 - שימוש חוזר בקוד (code reuse), שהוכח כנכון ויעיל, חוסך בעלויות פיתוח ומשפר את איכות התוכנה.
- תכנות מודולרי הוא הגישה המקובלת לבניית מערכות תוכנה העומדות בדרישות האיכות. שפות מונחות עצמים מספקות מנגנוני שפה לתכנות מודולרי – מחלקות ועצמים. בתכנות מונחה עצמים, תוכנית מורכבת ממחלקות, המגדירות עצמים שלהם מבנה ופעולות. בריצת תוכנית בשפה כזו, עצמים משתפים פעולה כדי לממש את מטרת התוכנית.
- **הכמסה (encapsulation)** – חלוקת תוכנית למודולים המתבססת על הרעיון של אריזה יחד של נתונים ושל פעולות הנעשות עליהם.
- עקרונות הפרדת ממשק ממימוש והסתרת המידע קובעים כי יש להבדיל באופן ברור בין ממשק של מחלקה לבין מימושה. על הממשק להכיל רק את המידע הדרוש להבנת מהות המחלקה ולהפעלת הפעולות המוגדרות בה. המימוש הוא הגדרת מבנה העצמים הנוצרים מהמחלקה ומהקוד של הפעולות, והוא צריך להיות מוסתר מהמשתמשים.
- תיעוד מלא ומדויק של ממשק של מחלקה חיוני לשימוש בה.
- מימושים יעילים של פעולות חשובים בתכנות מונחה עצמים, כמו בכל גישה אחרת לתכנות.
- פיתוח תוכנה בהיקף כלשהו כולל בהכרח הבטחת נכונותה. זו מושגת על ידי ניתוח הקוד ועל ידי בדיקת התוכנה במצבים שונים. במהלך בדיקת נכונות המימוש של פעולה יש חשיבות רבה לבדיקת מקרי קצה.

מושגים

encapsulation	הכמסה / אנקפסולציה
software engineering	הנדסת תוכנה
information hiding	הסתרת מידע
module	מודול
modularity	מודולריות
implementation	מימוש
interface	ממשק
user interface	ממשק למשתמש
software system	מערכת תוכנה
correctness	נכונות
object	עצם
code reuse	שימוש חוזר בקוד
maintenance	תחזוקה
documentation	תיעוד
Object Oriented Programming (OOP)	תכנות מונחה עצמים (תמי"ע)