Code Complete

מדריך מעשי לפיתוח תוכנה

מהדורה שנייה

Steve McConnell

תרגום ועריכה מקצועית: אשר ברק

Microsoft Press

הוצאת הוד-עמי ספרי מחשבים

Code Complete 2nd Ed By Steve McConnell

עריכה לשונית ועיצוב: **יצחק עמיהוד, שרה עמיהוד**

תרגום ועריכה מקצועית: **אשר ברק**

עיצוב עטיפה: **נלי מגיט**

שמות מסחריים

הודעה

www.hod-ami.co.il info@hod-ami.co.il

מהדורה ראשונה בעברית 9/2013

מסת"ב ISBN 978-965-361-410-9

תוכן עניינים מקוצר

תוכן העניינים

הקדמה

אם הספר הזה הגיע לידיך, סביר להניח שאתה עוסק בכתיבה ובפיתוח של תוכנה. גם אין חשיבות לעובדה שאתה בעל תואר אקדמי או בוגר קורס, סטודנט, או שאתה לומד בעצמך מספרים ומרשת האינטרנט. להבדלים ברקע ובהשכלה יש חשיבות בנושאים אחרים, שהם מחוץ לתחום עיסוקו של ספר זה.

אם אתה עוסק בתוכנה, סביר להניח שאתה מכיר וקרוב לנושאים שונים שנדונים בספר זה. אתה גם יודע ויכול לכתוב ולהבין קוד בשפת תכנות אחת או יותר. יש להניח שאתה מכיר את המשמעות של השימוש בכלי תכנות שונים כמו משתנים, לולאות, פונקציות, מחלקות, הורשות, רקורסיה ועוד. אם הספקת ללמוד נושאים מסוימים בצורה מעמיקה יותר, בוודאי נחשפת לדרך הבינארית שבה פועל עולם המחשוב, מכיר את המחסנית ואת הערימה ויכול לדון בהקצאות זיכרון. אם למדת במסגרת אקדמית, יש להניח שתוכל להעריך את המורכבות של אלגוריתמים ולהתאים את מבנה הנתונים למשימה שעל הפרק, כמו גם עניינים רבים אחרים, ולעשות בהם שימוש טוב.

עם זאת, סביר להניח שהמרצים בכל מסגרת לימוד לא הקדישו זמן רב, או שלא הקדישו כלל, ולא הרחיבו את ההסברים על **הדרכים להפיק קוד טוב**. כמו בכל תחום, על תשתית הבסיס של הידע האקדמי והכלים, מצטברת עם השנים שכבה עבה של ידע מעשי שמאפשרת להשתמש בכלים השונים בצורה אפקטיבית. בעולם התוכנה, שימוש אפקטיבי בכלים, פירושו כתיבת תוכנה איכותית בתהליך יעיל. הנה כמה מאפיינים של כתיבה כזו:

- הקוד שלך קריא ונוח לתחזוקה
- הקוד שלך עושה את מה שהוא צריך לעשות
- בתהליך התכנון והכתיבה, הזמן שלך מושקע בדברים הנכונים
- אתה מתקשר ומשתף פעולה בצוות בתכנון ובכתיבת הקוד
- אתה קשוב לתהליך שעובר עליך בזמן הפיתוח ולומד ממנו לעתיד

כתיבת תוכנה איכותית מתבססת גם על הבנה של הדרך שבה פועלים הכלים, אבל גם על הבנה של דברים נוספים רבים שלא זוכים בדרך כלל לתשומת הלב הראויה. בכלל זה אפשר לכלול את ההבנה של תהליך הכתיבה, הבנה של תהליך התכנון, הבנה של הדרך שבה הקוד מייצג את המציאות, הבנה של מחזור החיים של הקוד, הבנה של האופן שבו כתיבת הקוד משפיעה על האפשרות לתחזק אותו ולשנות אותו, ועוד.

ספר זה מכיל אוצר של "חוכמה" כזו. חלקה התפרסם במגזינים מקצועיים, חלקה הוא תוצאה של המחקר האקדמי במדעי המחשב, וחלקה מבוסס על ניסיון מקצועי מצטבר.

ספר זה אינו עוסק בשפת תכנות מסוימת. לכל שפת תכנות (ויש רבות כאלו) אפשר למצוא שפע של תיעוד, כמו ספרי לימוד, מדריכים, שיעורים מוקלטים ומצולמים ועוד. הספר כולל דוגמאות בכמה שפות תכנות, אבל הרעיונות שהוא מציג הם רעיונות כלליים של מקצוע פיתוח התוכנה ותקפים באופן שווה בכל השפות. זו גם הסיבה **שלא צירפנו את קטעי הקוד**.

למי מיועד הספר הזה?

ספר זה מיועד לאנשים שלמדו תכנות בעצמם, לתלמידים העוסקים בתוכנה, לסטודנטים למדעי המחשב, למתכנתים ואנשי מקצוע אחרים בתחום, לראשי צוותי תוכנה, למנהלי פיתוח, ליועצים טכניים, לגורו של תוכנה ולאוהבי תוכנה ותכנות.

הספר מיועד לכל מי שעשוי לעסוק בכתיבת קוד באופן מקצועי וכל מי שעשוי לעסוק בכתיבה באופן חובבני, אבל אוהב לחשוב ולהבין את מה שהוא עושה, ורוצה לעשות זאת בדרך איכותית.

איך לקרוא את הספר הזה

מומלץ מאוד לקרוא את הספר מהתחלה. הרעיונות המרכזיים יחלחלו במהלך הקריאה אליך, הקורא. אנו מקווים שהדברים גם ישנו את הדרך שבה תכתוב קוד, תקרא ותעריך את האיכות שלו. הספר כתוב בצורה ידידותית וכולל הסברים ודוגמאות קוד רבות. רוב הרעיונות והדוגמאות מובנים בקלות בקריאה ראשונה.

במשך הזמן תמצא את עצמך חוזר אל הספר, כדי להבהיר לעצמך נושאים שחשובים בעיניך, או שנתקלת בהם במהלך העבודה השוטפת. תוכן העניינים והאינדקס יאפשרו לך למצוא את הנושא הרלוונטי בקלות.

הערת המתרגם

הספר מתורגם מתוך הספר

Code Complete 2nd Ed.

A practical handbook of software construction

Steve McConnel

Microsoft Press © 2004

היה לי המזל להיתקל בספר הזה בתחילת הקריירה שלי כמתכנת מקצועי. עולם התוכנה עבר מאז תהפוכות בקצב שאופייני רק לו כנראה. פלטפורמות התחלפו, מערכות הפעלה באו והלכו, טכנולוגיות פרצו, שלטו, והתיישנו בן לילה. ועדיין, כמעט כל מילה בספר הזה נשארה רלוונטית. המהדורה הראשונה של הספר יצאה לשוק בשנת 1993 וזכתה ב- Jolt Award, מהדורה שנייה יצאה בשנת 2004 והיום, עשר שנים אחרי יציאתה לאור, **הספר הזה נחשב עדיין לאחד מספרי החובה למתכנת**. למעשה, הוא מופיע בכל רשימה של ספרי חובה למתכנת שמעלה החיפוש בגוגל, וכמעט תמיד הוא נמצא במקום הראשון.

במהלך השנים הזדמן לי להציג את הספר הזה לכמה עמיתים והופתעתי לא אחת מהעובדה שגם אנשים ותיקים ומנוסים שאני מעריך את כישוריהם כמתכנתים אינם מכירים אותו. כשהתחלתי לעסוק בתפקידים ניהוליים שכללו חניכה והכשרה, מצאתי את עצמי מלמד את העקרונות המופיעים בו שוב ושוב. לצערי, התברר לי שהמשימה של קריאת שמונה מאות עמודים באנגלית מרתיעה רבים וטובים ועוברת למקום לא ריאלי בסדר היום העמוס שלנו.

לקחתי על עצמי את תרגום הספר כתרומה לקהילת הפיתוח בארץ. אני אסיר תודה לסטיב מקונל, להוצאת Microsoft Press ולהוצאת O'Reilly בעלי הזכויות בספר, שאיפשרו בנדיבותם את המבצע הזה. אני מודה להוצאת הוד-עמי שלקחה על עצמה את ההוצאה לאור בעברית, בייחוד ליצחק עמיהוד, אחד מאבות התחום בארץ.

מכיוון שרצינו לשמור על נפח נגיש של המהדורה העברית עבור המתכנת הסביר, הושמטו מהספר המקורי, בהסכמה ובתיאום עם ההוצאה באנגלית, מספר פרקים העוסקים בתכנון ברמות גבוהות יותר ובטכניקות שונות שמשיקות לתהליך הפיתוח עצמו. בפרקים שתורגמו השתדלתי לשמור על רוחו ולשונו העשירה והציורית של המחבר, כפי שבאו לידי ביטוי במקור באנגלית. עם זאת, במקומות מסוימים נעשו התאמות ותמצות לפי הצורך.

אם מצאת את הספר הזה מועיל, רכוש גם את הנוסח המקורי המלא והרחב את ידיעותיך.

בהצלחה.

פרק 2

מחלקות (Classes)

בראשית התכנות, המתכנתים חשבו על תכנות במונחים של **פקודות** (statements). במהלך שנות ה-70 וה-80 של המאה הקודמת, המתכנתים התחילו לחשוב במונחים של **פונקציות** (functions). במאה ה-21, המתכנתים חושבים במונחים של **מחלקות** (classes). הערה: בספרות באנגלית מקובל עבור **פונקציות** גם השימוש במונח **שגרות** (routines).

מחלקה היא אוסף של נתונים ושל פונקציות שחולקות אחריות לנושא מוגדר היטב. מחלקה יכולה להיות אוסף של פונקציות בנושא מוגדר היטב, גם אם אין להן נתונים משותפים. מתכנת טוב יותר, הוא זה שמצליח להגדיל ככל האפשר את חלקי התוכנה שהוא יכול להתעלם מהם בביטחה, בזמן שהוא עובד על חלקים אחרים של הקוד. מחלקות הן הכלי המרכזי בהשגת המטרה הזו.

הבסיס למחלקות: סוגי נתונים מופשטים (ADT)

המונח ADT (Abstract Data Types), או סוגי נתונים מופשטים, מייצג אוסף של נתונים ופעולות על הנתונים הללו. הפעולות האמורות גם מתארות את הנתונים לעולם החיצוני וגם מאפשרות לעולם החיצוני לשנות אותם. ADT יכול להיות חלון במערכת גרפית עם כל הפעולות שמשפיעות עליו, קובץ ופעולות על קובץ, טבלת שיעורי ביטוח ופעולות עליה, או צירופים אחרים דומים של נתונים ופעולות.

הבנת ADT חיונית להבנה של **תכנות מונחה עצמים** (object-oriented programming). בלי להבין את המשמעות של ADT, מתכנתים עלולים ליצור מחלקות שהן מחלקות רק בשם ולא במהות. בפועל, מחלקות כאלו הן בסך הכל אריזה נוחה לאוסף של נתונים ופונקציות בעלי קשר רופף ביניהם. הבנת ADT מאפשרת למתכנתים ליצור מחלקות קלות יותר לפיתוח ראשוני וקלות יותר לשינוי על פני הזמן.

בעבר, ספרי תכנות פנו לעולם המתמטי בלבד, כשהגיעו לנושא ADT. הם התבטאו בנוסח כזה, למשל: "אפשר לחשוב על ADT כעל מודלים מתמטיים עם אוסף הפעולות שמוגדרות עליו". הגדרות כאלו גורמות לכך שיובן כאילו השימוש היחיד שאפשר למצוא ל-ADT הוא כחומר הרדמה...

הסברים יבשושיים כאלה של ADT מחמיצים את הנקודה העיקרית לחלוטין. ADT, כלומר סוגי נתונים מופשטים, הם עצמים מרגשים, משום שאפשר להשתמש בהם כדי לעבוד עם ישויות בעולם האמיתי, במקום להתעסק עם ביטויי תכנות מפורטים. במקום להוסיף איבר לרשימה מקושרת, אפשר לומר שמוסיפים תא לגיליון אלקטרוני, או שמוסיפים סוג חלון חדש לרשימת סוגי חלונות, או לומר שמוסיפים קרון נוסעים להדמיה של רכבת. על ידי ADT אפשר להשתמש במונחים של עולם הבעיה במקום במונחים של עולם התכנות!

דוגמה לצורך ב-ADT

כדי להתחיל בהסבר, ניקח לדוגמה מקרה שבו **סוג נתונים מופשט** (ADT) יכול להיות שימושי. ניכנס לפרטים אחרי שתהיה לנו דוגמה לדבר עליה.

נניח שאתה כותב תוכנה ששולטת בהצגת טקסט על המסך ומשתמשת לשם כך בגופנים (fonts) שונים, בגדלים שונים, ובתכונות נוספות של הגופן (כמו נטוי או מודגש). חלק מהתוכנה משנה את הגופן של הטקסט. אם תשתמש ב-ADT, תהיה לך קבוצה של פונקציות שמתייחסות לגופן, יחד עם הנתונים של הגופן כמו השם שלו, הגודל והתכונות הנוספות. האוסף הזה של פעולות על גופן ונתונים שלו הוא ADT.

עכשיו נניח שאינך משתמש ב-ADT, כלומר, אתה מייצג את אוסף הנתונים של הגופן, אבל אינך כולל בחבילה גם את הפונקציות שקשורות לגופן. עכשיו נניח שברצונך לשנות גודל של גופן מסוים ל-12 נקודות, שזה למעשה 16 פיקסלים. לצורך זה תכתוב משהו כזה:

```
currentFont.size=16
```

אם בנית ספרייה של פונקציות שירות, הקוד עשוי להיות קצת קריא יותר:

```
currentFont.size=PointsToPixels(12)
```

תוכל גם לתת לתכונה "גודל" שם קצת יותר ספציפי, כמו למשל:

```
currentFont.sizeInPixels=PointsToPixels(12)
```

כעת, אינך יכול לכתוב גם את הביטוי *currentFont.sizeInPixels* וגם את הביטוי *currentFont.sizeInPoints* מכיוון שאם תציין את שניהם, *currentFont* לא ידע באיזה מהם להשתמש.

אם אתה צריך להדגיש את הגופן, אתה עשוי להשתמש באחד מהקודים לעיל שתבחר, ולהוסיף or עם קבוע הקסדצימלי *0x02*:

```
currentFont.attribute= currentFont.attribute or 0x02
```

תוכל לכתוב קוד יותר ברור ומתומצת, כמו לדוגמה:

```
currentFont.attribute= currentFont.attribute or BOLD
```

או אולי:

```
currentFont.bold=True
```

כמו בסוגיה של גודל הגופן שהוצגה קודם, אם הקוד צריך לעבוד על הנתונים באופן ישיר, הדבר מגביל את אופן השימוש ב- *currentFont*.

אם אתה מתכנת באופן כזה, יש להניח שתשתמש פעמים רבות בשורות קוד אלו, וכשתצטרך לשנות משהו, תצטרך לעשות את זה בהרבה מקומות.

היתרונות של שימוש ב-ADT

הבעיה אינה בכך שהגישה שמשנה את הנתונים באופן ישיר היא שיטת תכנות לא טובה. הבעיה היא בכך שאפשר להחליף אותה בשיטה טובה יותר, שמציעה את היתרונות הבאים:

אתה יכול להסתיר פרטי מימוש. הסתרת מידע על הגופן פירושה שאם הגופן משתנה, אפשר לשנות אותו במקום אחד מבלי להשפיע על כל התוכנה. לדוגמה, אם לא היית מסתיר את פרטי המימוש בתוך ADT, שינוי הייצוג של התכונה bold מהשיטה הראשונה לשיטה השנייה היה גורר שינוי של התוכנה שלך בכל מקום שמשתמשים בו, מכיוון שלא קבעת את ההדגשה, או אי-ההדגשה, של הגופן במקום אחד בלבד. הסתרת המידע על הגופן מגינה על שאר חלקי התוכנה, אם החלטת, למשל, לאחסן נתונים באחסון חיצוני במקום בזיכרון, או לכתוב מחדש את כל הפונקציות שמטפלות בגופנים, ברור שטיפול חד-פעמי טוב יותר משיטוט בכל התוכנית לשם שינוי.

שינויים אינם משפיעים על כל התוכנה. אם גופנים צריכים להשתכלל כדי לתמוך בפונקציות נוספות (כמו למשל תמיכה באותיות רישיות קטנות [small caps], אותיות רגילות וכד'), תוכל לשנות את התוכנה במקום אחד. השינוי לא ישפיע על שאר חלקי התוכנה.

תוכל לעשות את הממשק החיצוני שלך ברור יותר. קוד כמו למשל *Font.size=16* אינו ברור במידה מספקת משום ש-*16* יכול להיות גודל בנקודות או בפיקסלים. אי אפשר להסיק מההקשר במה מדובר. אם אוספים את כל הפונקציות הרלוונטיות לתוך ADT, אפשר לקבוע ממשק אחיד במונחי נקודות, או במונחי פיקסלים, או במונחים שמבחינים בין השניים באופן ברור, וכך לתמוך במניעת בלבול.

קל יותר לשפר ביצועים. אם צריך לשפר ביצועים של גופן, אפשר לכתוב מחדש כמה פונקציות מוגדרות-היטב, במקום להתחיל לעשות זאת במקומות רבים בגוף התוכנית.

קל יותר לראות שהתוכנה נכונה. אפשר להחליף את העבודה המעייפת של בדיקת נכונות של ביטויים כמו *currentFont.attribute = currentFont.attribute or 0x02* במשימה הקלה יותר של בדיקת נכונות של ביטויים כמו *currentFont.SetBoldOn()*. בביטוי הראשון אפשר לטעות בשם התכונה שמשנים, בפעולה (*and* במקום *or*), או בערך (*0x20* במקום *0x02*). במקרה השני, הטעות היחידה בביטוי *currentFont.SetBoldOn()* יכולה להיות קריאה לפונקציה בשם הלא נכון, ולכן קל יותר לזהות זאת.

התוכנה הופכת למתועדת מתוך עצמה (self-documenting). אפשר להחליף משפטים כמו *currentFont.attribute= currentFont.attribute or 0x02* על ידי החלפה של הערך 0x02 בקבוע BOLD או במה שהערך הזה מייצג, אבל זה בוודאי לא משתווה לבהירות ופשטות הקריאה של *currentFont.SetBoldOn()*.

במחקר שנערך על ידי Woodfield, Desmond ו-Shen נתבקשו סטודנטים להשיב על מבחן שהתייחס לשתי תוכנות: אחת חולקה לשמונה פונקציות לפי פונקציונלית, והשנייה חולקה לשמונה יחידות של ADT (1981). הסטודנטים שנבחנו על ADT קיבלו ציונים גבוהים יותר בכ-30 אחוזים משל חבריהם שנבחנו על הגרסה הפונקציונלית.

אינך צריך להעביר נתונים ממקום למקום בתוכנית. בדוגמה שהוצגה בפיסקה קודמת, עליך לשנות את העצם *currentFont* ישירות או להעביר אותו לכל פונקציה שפועלת עם גופנים. אם אתה משתמש ב-ADT, אינך צריך להעביר את *currentFont* לכל מקום, ואינך צריך להפוך אותו למשתנה גלובלי. הנתונים של ADT נמצאים בתוכו, ורק הפונקציות שלו יכולות לגשת אליהם. פונקציות שאינן חלק מה ADT לא צריכות לעסוק בנתונים של הגופן.

אתה יכול להשתמש בישויות מהעולם האמיתי במקום במבני נתונים ברמת המימוש. אתה יכול להגדיר פעולות שמתייחסות לגופנים באופן ששאר חלקי התוכנית יוכלו לפעול במונחי גופנים, ולא במונחים של גישה למערכים, הגדרות של מבני נתונים, מצבי *True* או *False* וכד'. במקרה שלנו, אם מגדירים ADT, מגדירים כמה פונקציות ששולטות בגופן, אולי כך:

```
currentFornt.SetSizeInPoints(sizeInPoints)
currentFornt.SetSizeInPixels(sizeInPixels)
currentFornt.SetBoldOn()
currentFornt.SetBoldOff()
currentFornt.SetItalicOn()
currentFornt.SetItalicOff()
currentFornt.SetTypeface(faceName)
```

הקוד שנמצא בפונקציות אלו יהיה כנראה קצר ודומה מאוד לקוד שראינו קודם לכן. ההבדל הוא בזה שבודדנו את הגישה לנתוני הגופן באמצעות אוסף של פונקציות. זה מְסַפֵּק רמה טובה יותר של הפשטה עבור שאר חלקי התוכנית בכל הקשור לעבודה עם גופנים, ונותן שכבה של הגנה בפני שינויים בגופנים.

דוגמאות נוספות ל-ADT

נניח שאתה כותב תוכנה ששולטת במערכת הקירור של כור גרעיני. אתה יכול להתייחס למערכת הקירור כאל ADT, על ידי הגדרת הפעולות הבאות עבורה:

```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate(rate)
coolingSystem.OpenValve(valveNumber)
coolingSystem.CloseValve(valveNumber)
```

הסביבה המיוחדת שבה תפעל תקבע את הקוד שייכתב כדי לממש כל אחת מפעולות אלו. שאר התוכנה יכולה להתייחס את מערכת הקירור באמצעות פונקציות אלו ולא תצטרך לפרט לכל אחד מהם את המידע של מבנה הנתונים, מגבלות של מבנה הנתונים, שינויים וכד'.

הנה כמה דוגמאות ל**סוגי נתונים מופשטים** (ADTs) ופעולות אפשריות עליהם:

Cruise Control
Set speed
Get current settings
Resume former speed
Deactivate

List
Initialize list
Insert item in list
Remove item from list
Read next item from list

Set of Help Screens
Add help topic
Remove help topic
Set current help topic
Display help screen
Remove help display
Display help index
Back up to previous screen

Pointer
Get pointer to new memory
Dispose of memory from existing pointer
Change amount of memory allocated

Blender
Turn on
Turn off
Set speed
Start "Insta-Pulverize"
Stop "Insta-Pulverize"

Light
Turn on
Turn off

Menu
Start new menu
Delete menu
Add menu item
Remove menu item
Activate menu item
Deactivate menu item
Display menu
Hide menu
Get menu choice

Fuel Tank
Fill tank
Drain tank
Get tank capacity
Get tank status

Stack
Initialize stack
Push item onto stack
Pop item from stack
Read top of stack

File
Open file
Read file
Write file
Set current file location
Close file

Elevator
Move up one floor
Move down one floor
Move to specific floor
Report current floor
Return to home floor

אפשר להסיק מהדוגמאות הללו כמה עקורונות מנחים, אשר מפורטים להלן:

בנה והשתמש בסוגי נתונים אופייניים בצורה של ADT, ולא בצורה של סוגי מידע בסיסיים. מרבית הדיון סביב ADT מתייחס לייצוג של **סוגי נתונים** (data types) אופייניים בתור ADT. כמו שאתה יכול לראות מהדוגמאות, אפשר ליצג **ערימה** (stack), **רשימה** (list) **ותור** (queue) או כל מבנה נתונים אחר בתור ADT.

השאלה שעליך לשאול את עצמך היא "מה הערימה, הרשימה או התור מייצגים?" אם הערימה מייצגת עובדים, התייחס ל-ADT של עובדים ולא לערימה סתם; אם הרשימה מייצגת רשומות חיוב, התייחס אליה כאל רשומות חיוב ולא כאל רשימה סתמית; אם התור

פרק 3

פונקציות איכותיות

בסעיפים קודמים עסקנו במחלקות. נעבור עכשיו למה שקורה בקרביים, בפונקציות ובמאפיינים שיוצרים את האבחנה בין פונקציה איכותית לפונקציה גרועה. אם אתה מעדיף לקרוא על תכנון פונקציות לפני צלילה לפרטים הקטנים, עיין בפרק 1, "תכנון" ואז חזור לכאן. תכונות חשובות נוספות של תכנון פונקציות מופיעות בפרק 4, "תכנות מתגונן".

לפני שנעבור לפרטים המאפיינים פונקציות איכותיות, כדאי שנגדיר בדיוק שני עניינים בסיסיים.

מה זו "**פונקציה**"? **פונקציה** (function) היא יחידת קוד שניתן להפעיל אותה לצורך מטרה ברורה אחת. דוגמאות לכך עשויות להיות function בשפת C++; method בשפת Java; function או sub procedure בשפה Visual Basic; גם macros ב- C וב- C++ יכולים להיחשב פונקציות.

מהי "**פונקציה איכותית**"? תשובה לשאלה זו מורכבת יותר. הדרך הפשוטה ביותר להסביר זאת היא להראות מהי פונקציה לא איכותית. הנה דוגמה לפונקציה באיכות נמוכה:

```
C++ Example of a Low-Quality Routine
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
  double & estimRevenue, double ytdRevenue, int screenX, int screenY,
  COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
  int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
  inputRec.revenue[i] = 0;
  inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
  }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
    for ( i = 0; i < 12; i++ )
        profit[i] = revenue[i] - expense.type1[i];
    }
```

```
else if ( expenseType == 2 ) {
        profit[i] = revenue[i] - expense.type2[i];
        }
else if ( expenseType == 3 )
        profit[i] = revenue[i] - expense.type3[i];
        }
```

מה לא בסדר בפונקציה זו? הנה רמז: אתה אמור למצוא לפחות 10 בעיות שונות בפונקציה זו, ולאחר שתגבש את הרשימה שלך, אנא עיין ברשימה זו:

- לפונקציה נקבע שם גרוע. השם *HandleStuff()* אינו אומר דבר על תפקיד הפונקציה.
- הפונקציה אינה מתועדת.
- לפונקציה יש פריסה גרועה. הארגון הפיזי של הקוד שפרוס על הדף אינו נותן הכוונה על הארגון הלוגי שלה.
- הערך של משתנה הקלט *inputRec* משתנה. אם זה ערך קלט של הפונקציה הוא לא אמור להשתנות (וב- C++ הוא צריך להיות מוגדר *const*). אם הערך של המשתנה אמור להשתנות, הוא לא אמור להיקרא *inputRec*.
- הפונקציה קוראת וכותבת למשתנים גלובליים. היא קוראת מהמשתנה *corpExpense* וכותבת אל *profit*. הפונקציה צריכה לתקשר עם פונקציות אחרות בצורה ישירה יותר, ולא באמצעות קריאה וכתיבה למשתנים גלובליים.
- לפונקציה אין מטרה יחידה. היא מאתחלת כמה משתנים, כותבת כמה נתונים לבסיס הנתונים, עושה כמה חישובים, ונראה שהפעולות אינן קשורות זו לזו. לפונקציה צריכה להיות מטרה אחת ברורה ומוגדרת היטב, ופעולות שמכוונות לאותה מטרה.
- הפונקציה אינה מגינה על עצמה מפני נתונים משובשים. אם *crntQtr* שווה 0, הביטוי *ytdRevenue * 4.0 / (double) crntQtr* יגרום לשגיאה של חלוקה ב-0.
- הפונקציה עושה שימוש בכמה "מספרי קסם" כגון 100, 4.0, 12, 2 ו-3. דיון ב"מספרי קסם" ראה סעיף "מספרים" בפרק 7.
- הפונקציה אינה משתמשת בפרמטרים שלה *screenX* ו-*screenY*.
- אחד מהפרמטרים של הפונקציה מועבר אליה בצורה לא נכונה. *prevColor* מסמן בתור reference parameter (כלומר &), למרות שהפונקציה אינה מציבה בו ערך.
- לפונקציה יש יותר מדי פרמטרים. הגבול העליון של מספר פרמטרים המקובל עבור פונקציה הוא 7; הפונקציה הזו מקבלת 11. הפרמטרים מסודרים בצורה לא נוחה לקריאה ולכן רוב המתכנתים אפילו לא ינסו לבחון אותם מקרוב או לספור אותם.
- הפרמטרים של הפונקציה הזו אינם מסודרים בצורה סבירה ואינם מתועדים.

פרט למחשב עצמו, הפונקציות הן ההמצאה הגדולה ביותר של עולם המחשוב. הפונקציה גורמת לכך שהתוכנית תהיה קלה לקריאה ולהבנה יותר מכל תכונה אחרת של כל שפת תכנות. רע מאוד הדבר שבדוגמה זו השימוש בכלי זה נעשה בדרך כל כך גרועה.

הפונקציה היא גם הטכניקה הטובה ביותר לחיסכון במקום ולשיפור ביצועים. תאר לעצמך כמה ארוך יותר היה הקוד שלך אם היית צריך לחזור על כל הקוד של כל פונקציה בכל מקום בתוכנית במקום לקרוא לה. תאר לעצמך כמה קשה היה לשפר ביצועים אם היה צריך לטפל בכל מופע כזה של הפונקציה. הפונקציה עושה את התכנות המודרני לאפשרי.

"ובכן," אתה אומר בוודאי, "אני יודע שפונקציות הן דבר נפלא ואני משתמש בהן בכל הזדמנות. אז מה אתה רוצה בעצם?"

אני רוצה לומר שיש הרבה סיבות טובות ליצור פונקציות ושקיימות דרכים נכונות ודרכים לא נכונות לעשות זאת. בתחילה נראה שכותבים פונקציות כדי לא לחזור על הקוד פעמים רבות ולהשתמש בקריאות אל אותו קטע קוד. ספר המבוא לתוכנה אומר שפונקציות טובות משום שהם עושות את התוכניות קלות יותר לפיתוח, לאיתור שגיאות ולניפוי (debugging), לתיעוד ולתחזוקה. חוץ מכמה פרטים לגבי התחביר של שימוש בפרמטרים ומשתנים מקומיים זה מה שהיה שם. לא היה למתכנת תיאור טוב ומקיף של התיאוריה והפרקטיקה של יישום פונקציות. כאן תמצא הסבר הרבה יותר טוב.

סיבות טובות ליצור פונקציה

הנה רשימה של סיבות ליצור פונקציה, חלק מהן חופפות במידה זו או אחרת:

הפחתת מורכבות. הסיבה החשובה ביותר ליצור פונקציות היא כדי להפחית את המורכבות של התוכנית. צור פונקציה כדי להסתיר פרטים, כך שלא תצטרך לחשוב ולטפל בהם במהלך התכנות. צריך להתמקד בפרטים בעת כתיבת הפונקציה, אבל אחרי שהפונקציה נכתבה, אתה אמור להיות מסוגל "לשכוח" מהפרטים ולהשתמש בפונקציה בלי לדעת איך היא פועלת בתוכה. סיבות אחרות ליצירת פונקציות - צמצום גודל הקוד, שיפור היכולת לתחזק את המערכת, שיפור הנכונות של המערכת - כולן סיבות טובות, אבל בלי יכולת ההפשטה של הפונקציות, תוכנית מורכבת תהיה בלתי אפשרית לניהול מבחינה אינטלקטואלית.

אינדיקציה אחת לצורך בפונקציות היא **קינון** (nesting) עמוק של פונקציות או של תנאים. הפחת את המורכבות באמצעות העברת החלק הפנימי לפונקציה נפרדת.

הוספת רמת הפשטה. העברת קוד לתוך פונקציה בעלת שם ברור, היא אחת הדרכים הטובות ביותר לתיעוד המטרה שלו. במקום לקרוא סדרה של שורות קוד כאלו:

```
if ( node <> NULL ) then
  while ( node.next <> NULL ) do
    node = node.next
    leafName = node.name
  end while
else
  leafName = ""
end if
```

אתה יכול לקרוא את השורה הברורה הזו:

```
leafName = GetLeafName( node )
```

הפונקציה החדשה כל כך קצרה, ולכן כל התיעוד שהיא צריכה הוא שם טוב. השם נותן רמת הפשטה גבוהה יותר משמונה שורות הקוד שהיו קודם. כך הקוד קריא יותר וקל יותר להבנה, ומפחית את המורכבות של הפונקציה שהכילה את הקוד שנכתב קודם.

מניעת קוד כפול. כנראה שהסיבה המקובלת ביותר ליצירת פונקציות היא ההימנעות מקוד כפול. ואמנם, יצירת קוד כפול בשתי פונקציות אכן מרמזת על טעות בתכנון. שים גרסה גנרית של הקוד המשותף במחלקת בסיס, והעבר את הקוד המתמחה למחלקות יורשות. אפשר גם למשוך את הקוד המשותף אל פונקציה נפרדת וליצור שתי פונקציות שיקראו לה. כאשר הקוד נמצא במקום אחד אתה חוסך את המקום שהיית צריך להקדיש לקוד הכפול, וגם שינויים יהיו קלים יותר כי תצטרך לעשות אותם במקום אחד בלבד. הקוד יהיה אמין יותר משום שתצטרך לבדוק רק במקום אחד כדי לוודא שהוא עובד. שינויים יהיו קלים יותר משום שתוכל להימנע מהמחשבה שאתה עושה את אותו שינוי בכמה מקומות ולתאם ביניהם, בשעה שבפועל אתה עושה שינויים שונים מעט זה מזה.

תמיכה בהורשה. צריך מעט קוד כדי לדרוס פונקציה קצרה במחלקת הבסיס. תוכל להקטין את שיעור הטעויות במחלקות יורשות אם תשמור על פונקציות פשוטות במחלקת הבסיס.

הסתרת הסדר בו מתבצעים דברים. כדאי להסתיר את הסדר בו מתבצעים דברים. למשל, אם תוכנית מקבלת נתונים מהמשתמש ואז מקבלת נתונים נוספים מקובץ, אף אחת מהפונקציות הללו לא צריכה להיות תלויה בזה שפונקציה אחרת כבר התבצעה לפניה. עוד דוגמה לרצף יכולה להיות, כשאתה כותב קוד שקורא את הרשומה הראשונה **במחסנית** (stack) ומפחית מהמשתנה *stackTop*. שים את שתי שורות הקוד הללו בתוך הפונקציה *PopStack()* כדי להסתיר את הסדר שבו מתבצעים הדברים. עדיף להסתיר זאת מאשר לכתוב במקומות שונים בקוד.

הסתרת פעולות על מצביעים (pointers). מצביעים נוטים להיות קשים לקריאה והם עלולים להיות עתירי שגיאות. על ידי בידוד פעולות על מצביעים לפונקציות, ביכולתך להתרכז בפעולה שאתה מנסה לבצע ולא בניהול של המצביעים. אם אתה מבצע פעולות על מצביעים במקום אחד בלבד, אתה יכול להיות רגוע יותר לגבי הנכונות שלהן. אם תמצא סוג נתונים אחר שיהיה טוב יותר ממצביעים, תוכל לשנות את התוכנית מבלי לגרום לנזקים גדולים בקוד שמשתמש בהם.

שיפור ניידות. השתמש בפונקציות כדי לבודד פונקציונליות שקשה להעביר אותה. פונקציונליות כזו יכולה להיות, תלות בחומרה, תלות במערכת ההפעלה וכד'.

פישוט תנאים בוליאניים מורכבים. בדרך כלל לא צריך להבין תנאים בוליאנים מורכבים כדי להבין את התוכנה. העברת תנאים כאלה לפונקציות עושה את הקוד קריא

יותר משום ש-(1) הפרטים של התנאי לא נמצאים מול העיניים ו-(2) יש לתנאי שם קריא שמתאר אותו.

כשקובעים לתנאי שם ברור אפשר גם להדגיש את החשיבות שלו. זה מעודד מאמץ נוסף לעשות את התנאי קריא יותר בתוך הפונקציה שלו. התוצאה היא גם קוד קריא יותר בפונקציה הראשית וגם קוד ברור יותר בתנאי עצמו. פישוט תנאים בוליאניים הוא דוגמה להפחתת מורכבות.

שיפור ביצועים. תוכל לשפר את הביצועים של הקוד במקום אחד ללא צורך לחפש את אותו הקוד במקומות שונים כדי לשפר אותו. קוד שכתוב במקום אחד מקל על profiling ועל מציאת מוקדים של חוסר יעילות. ריכוז הקוד בפונקציה אחת תורם לכך ששיפור בפונקציה ישפר את כל הקוד שעושה בה שימוש, באופן ישיר או בעקיפין. כתיבת הקוד במקום אחד מאפשרת למעשה לכתוב מחדש את האלגוריתם בצורה יעילה יותר או בשפה מוצלחת יותר.

להזכירך, **Profiling** היא שיטה למציאת בעיות ביצועים בקוד באמצעות הרצה שלו, באופן שמתעד את הפונקציות שנקראו ואת הזמן שהוקדש לכל פונקציה.

לוודא שכל הפונקציות קצרות? לא צריך לעשות זאת בהכרח, עם כל כך הרבה סיבות טובות ליצור פונקציות, דרישה זאת מיותרת במידת מה. למעשה, יש משימות שמבוצעות טוב יותר בפונקציה אחת ארוכה. דיון על האורך הרצוי לפונקציות, ראה בסעיף "מהו האורך הרצוי לפונקציה?" שבהמשך בפרק 3.

פעולות שנראות פשוטות מדי להיות פונקציה

אחד ממחסומי המחשבה שמונעים יצירת פונקציות טובות, היא הרתיעה מיצירת פונקציות פשוטות. בניית פונקציה שתכיל שתיים או שלוש שורות קוד נראית מוגזמת. עם זאת, הניסיון מלמד שפונקציות קטנות יכולות להיות שימושיות מאוד.

לפונקציות קטנות יש כמה יתרונות. אחד מהן הוא שיפור הקריאות של הקוד. נניח שבקוד מופיעה השורה הזו פעמים אחדות:

```
Pseudocode Example of a Calculation
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

זו אינה שורת הקוד המסובכת ביותר בעולם. מרבית האנשים יבינו שהיא ממירה גודל מיחידות של המכשיר (device units) לגודל בנקודות. הם יראו שבכל הפעמים שבהם מופיעה השורה היא מבצעת את אותה המשימה. היה יותר ברור אם יוצרים פונקציה אחת עם שם ברור שתבצע את ההמרה במקום אחד:

```
Pseudocode Example of a Calculation Converted to a Function
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
  DeviceUnitsToPoints = deviceUnits *
    ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

כשהפונקציה הזו תחליף את הקוד, עשרות השורות שמופיעות בקוד לביצוע המשימה יהיו כך:

```
Pseudocode Example of a Function Call to a Calculation Function
points = DeviceUnitsToPoints( deviceUnits )
```

השורה הזו ברורה יותר, וכמעט מתועדת מתוך עצמה.

דוגמה זו מרמזת גם על סיבה נוספת ליצור פונקציות קטנות. פונקציות קטנות נוטות להתרחב. בדרך כלל לא יודעים זאת עד שכותבים את הפונקציה. בנסיבות מסוימות הקריאה לפונקציה *DeviceUnitsPerInch()* מחזירה *0*, והדבר מצביע על כך שצריך לטפל גם במקרה של חילוק באפס, ולצורך זה צריך לכתוב עוד 3 שורות קוד:

```
Pseudocode Example of a Calculation That Expands Under Maintenance
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;
  if ( DeviceUnitsPerInch() <> 0 )
    DeviceUnitsToPoints = deviceUnits *
      ( POINTS_PER_INCH / DeviceUnitsPerInch() )
  else
    DeviceUnitsToPoints = 0
  end if
End Function
```

אם הקוד המקורי היה כמו שהוא, היה צורך לעשות את השינוי הזה בעשרות מקומות בתוכנית(!) וזו כבר בעיה בסדר גודל אחר.

תכנון ברמת הפונקציה

הרעיון **עקביות** (cohesion) בהקשר לתכנות הוצג במאמר של Wayne Stevens, Glenford Myers, ו- Larry Constantine (1974). רעיונות מודרניים יותר, ביניהם **הפשטה** (abstraction) ו**כימוס** (encapsulation), נותנים תובנות טובות יותר ברמת המחלקה (ואכן, הם החליפו את תפיסת העקביות ברמת המחלקה). עם זאת, עקביות היא עדיין כלי מרכזי בעבודה על פונקציות בודדות.

בהקשר של פונקציות, העקביות מתבטאת בעוצמת הקשר שבין הפעולות שמבצעת הפונקציה. הפונקציה *Cosine()*, למשל, היא בעלת עקביות גבוהה משום שהיא מבצעת פעולה אחת בלבד. הפונקציה *CosineAndTan()* הינה בעלת עקביות נמוכה יותר משום שהיא מבצעת שתי פעולות. המטרה היא לתת לכל פונקציה לבצע פעולה אחת בלבד, ולא לבצע שום דבר אחר.

הדיונים הרבים בנושא מתייחסים בעיקר לרמות השונות של "עקביות". הבנת הרעיון חשובה יותר מהכרת המונחים המיוחדים לו. השתמש ברעיון כדי ליצור פונקציות עקביות ככל האפשר.

עקביות פונקציונלית היא הסוג החזק והטוב ביותר של עקביות, והיא מתרחשת כאשר הפונקציה מבצעת פעולה אחת בלבד. דוגמאות לפונקציות עם עקביות כזו: *sin()*, *GetCustomerName()*, *EraseFile()*, *CalculateLoanPayment()* ו- *AgeFromBirthdate()*. כמובן שכל שנאמר הוא בהנחה שהפונקציות הללו מבצעות מה שהשם שלהן אומר שהן מבצעות. אם זה לא המצב, יש לפונקציות הללו עקביות נמוכה וגם שם גרוע.

סוגים אחרים של עקביות נחשבים פחות אידיאליים.

שמות טובים לפונקציות

שם טוב לפונקציה אמור לתאר בצורה ברורה את כל מה שהפונקציה עושה. הנה כמה קווים מנחים לבחירת שמות טובים עבור פונקציות:

תאר את כל מה שהפונקציה עושה. בשם הפונקציה השתדל לתאר את כל התוצאות (outputs) ואת כל תופעות הלוואי (side effects). אם פונקציה מחשבת את הסה"כ של דוח ופותחת קובץ פלט, הרי *ComputeReportTotals()* אינו שם מתאים עבורה. השם *ComputeReportTotalsAndOpenOutputFile()* מתאים יותר, אבל נראה ארוך וטפשי. אם יש פונקציות רבות עם תופעות לוואי, יהיו לך שמות רבים ארוכים ואולי גם טיפשיים. הפתרון איננו קביעת שמות שיתארו פחות טוב את מה שעושות הפונקציות, אלא תכנות כזה שיפיק פונקציות שיעשו את הדברים באופן ישיר, ולא כתופעות לוואי של פונקציות אחרות.

הימנע מפעלים חסרי משמעות, מעורפלים, או מבולבלים. יש פעלים כאלה שאפשר "למתוח" אותם כדי לכסות עניינים שונים. שמות פונקציות כמו *HandleCalculation()*, *PerformSevices()*, *OutputUser()*, *ProcessInput()* ו- *DealWithOutput()* אינם אומרים מה בדיוק עושה הפונקציה, כי "חישוב", "שירות" או "קלט" אינם חד-משמעיים. לכל היותר, השמות הללו מלמדים שיש לפונקציה קשר לפעולות calculations, services, users, input או output. היוצא מן הכלל לזה הוא כאשר המונח Handle משמש במובן הטכני של טיפול באירוע.

לפעמים הפונקציה בסדר גמור מבחינה פונקציונלית, והבעיה היחידה היא בשם שלה. אם תחליף את השם *HandleOutput()* בשם *FormatAndPrintOutput()*, תדע טוב יותר מה עושה הפונקציה.

במקרים אחרים, הפועל מעורפל משום שהפעולות שמבוצעות על ידי הפונקציה מעורפלות. הפונקציה סובלת מחולשה בהצגת המטרה שלה. במקרה כזה, הטיפול הוא כנראה שינוי של מבנה הפונקציה ושל דרך פעולתה באופן שתהיה לה מטרה ברורה ושם שמבטא בצורה בהירה את המטרה הזו.

אל תבחין בין שמות פונקציות באמצעות מספר בלבד. מתכנת יכול לכתוב פעולות רבות ושונות תחת המטריה של פונקציה אחת. אחר כך הוא יכול לגזור קטעי קוד שונים

פרק 4

תכנות מתגונן (Defensive Programming)

תכנות מתגונן הוא לא הגנה על יכולות התכנות שלך, כי "זה עובד ועוד איך!". תפיסת התכנות המתגונן מבוססת על תפיסת **הנהיגה המתגוננת** (defensive driving), שבה אתה חושב כל הזמן שאינך יכול לדעת מה עומד לעשות הנהג שלידך. כך אתה מבטיח שאם הוא יעשה משהו מסוכן, אתה לא תיפגע. כך אתה מגן על עצמך גם מפני מקרים שבהם הפגיעה בך הינה אשמת הנהג האחר. בתכנות מתגונן, הכוונה היא שהפונקציה שלך לא תיפגע, אפילו אם היא מקבלת נתונים שאינם נכונים, ואפילו אם הנתונים הלא נכונים הללו הם באשמת הפונקציה האחרת. אפשר לומר זאת בצורה כללית יותר: התכנות המתגונן הוא הכרה בכך שייתכן שבפונקציה יהיו טעויות ויהיו בה שינויים, ולכן צריך לבנות את הקוד בהתאם.

הגנה על התוכנית שלך מפני קלט שגוי

בבית הספר ייתכן ששמעת את הביטוי "Garbage In – Garbage Out" (בראשי תיבות **GIGO**, זבל נכנס - זבל יוצא). כדי להפיק תוכנה ברמה מקצועית אין זה מספיק להכיר את הכלל החשוב הזה. תוכנה מקצועית לעולם אינה מוציאה זבל, ללא תלות בקלט שלה. מתכנת טוב משתמש בעיקרון "זבל נכנס - שום דבר לא יוצא", בעיקרון "זבל נכנס - הודעת שגיאה יוצאת", או "אין כניסה לזבל". בתכנות של ימינו, אם יש מצב של "garbage in – garbage out" זוהי הוכחה לתכנות רשלני ולא בטוח.

הנה דוגמה לקבלת תוצאות מעוותות כאשר הקלט אינו נכון בתוכנו. לדוגמה, מזינים תאריך לידה שגוי של עולה חדש, ואז הוא עלול לקבל צו גיוס אוטומטי בגיל גן הילדים, בשעה שהמחשב "סבור" שהוא כבר בוגר.

יש שלושה עקרונות לטיפול בקלט "זבל":

בדוק את כל הנתונים ממקורות חיצוניים. כשאתה מקבל נתונים מקובץ, ממשתמש, מרשת, או ממקור חיצוני אחר, ודא שהם בטווח המותר. ודא שערכים מספריים נמצאים בטווח הנכון ושמחרוזות תווים אינן ארוכות מדי. אם מחרוזת אמורה לייצג ערך מתוך טווח מסוים של ערכים (למשל מזהה של עסקה פיננסית, לדוגמה), ודא שהערך מתאים

למה שהוא אמור לייצג; אחרת, דחה אותו. אם אתה עובד על יישום מאובטח, שים לב לנתונים שעשויים לתקוף את היישום שלך: נסיונות לגרום לגלישה (buffer overflow), הזרקת פקודות SQL, הזרקת פקודות HTML או XML, גלישת ערכים מספריים (integer overflow), נתונים שמועברים אל פונקציות קריאה של מערכת ההפעלה וכד'.

בדוק את הערכים של כל הפרמטרים שנכנסים לפונקציה. בדיקת הערכים שנכנסים לפונקציה דומה לבדיקת הערכים שנקלטים ממקורות חיצוניים, ההבדל היחיד הוא שהערכים הללו מתקבלים מפונקציות אחרות ולא מחוץ לתוכנה. הדיון בסעיף "מחסומים להגבלת נזק שנגרם משגיאות" שבהמשך בפרק 4, מציג דרכים מעשיות לקבוע אילו פונקציות צריכות לבדוק את הקלט שלהן.

החלט איך להתייחס לקלט שגוי. מצאת פרמטר שגוי, מה עושים איתו? לפי המקרה שלפניך, אתה עשוי לבחור באחת מכמה דרכים שמתוארות בפירוט בסעיף "טכניקות לניהול שגיאות" שבהמשך בפרק 4.

תכנות מתגונן שימושי בשילוב עם טכניקות אחרות של אבטחת איכות שמתוארות בספר זה. הדרך הטובה ביותר לתכנות מתגונן היא להימנע משגיאות. תכנון איטרטיבי, כתיבת פסידו-קוד לפני הקוד, כתיבת הליכי בדיקה וניפוי לפני כתיבת הקוד עצמו, הן פעילויות שעוזרות במניעת שגיאות ולכן הן צריכות לקבל עדיפות על פני תכנות מתגונן. למרבה השמחה, אפשר לשלב אותן עם תכנות מתגונן.

בדיקות תקינות (Assertions)

בדיקות תקינות הן למעשה יחידות קוד שפועלות במהלך הפיתוח ומאפשרות לתוכנה לבדוק את עצמה כשהיא רצה (run). כשבדיקה מחזירה את הערך true (טוב), פירוש הדבר שהכל פועל כצפוי. כשבדיקה מחזירה את הערך false (שגוי) - פירוש הדבר שנמצאה שגיאה לא צפויה בקוד. לדוגמה, אם המערכת מניחה שקובץ לקוח לא יכיל לעולם יותר מ-50,000 רשומות, התוכנה יכולה לכלול בדיקת תקינות שמוודאת שגודל הקובץ קטן או שווה ל-50,000 רשומות. כל עוד מספר הרשומות בקובץ קטן מערך זה - הבדיקה תימשך, אבל כאשר תוכנית הבדיקה תזהה מספר רשומות גדול יותר - היא תתריע שיש בעיה.

בדיקות תקינות שימושיות במיוחד במערכות תוכנה גדולות ומורכבות ובתוכנות שמחייבות רמת אמינות גבוהה. הן מאפשרות למתכנת לחשוף הנחות שאינן נכונות, טעויות שנכנסות לקוד כשהוא משתנה וכו'.

בדיקת תקינות בדרך כלל מקבלת שני פרמטרים: ביטוי בוליאני שמתאר את ההנחה שאמורה להתקיים, והודעה שאמורה להיות מוצגת אם ההנחה אינה מתקיימת. כך נראית בדיקת תקינות בשפת Java כאשר המשתנה *denominator* לא אמור להיות אפס. לפניך דוגמה בקוד Java לבדיקת תקינות פשוטה:

```
Java Example of an Assertion
assert denominator != 0 : "denominator is unexpectedly equal to 0.";
```

בדיקת התקינות הזו בודקת שהמשתנה *denominator* אינו שווה ל-*0*. הפרמטר הראשון, *denominator !=0*, הוא ביטוי בוליאני שעשוי להיות *true* או *false*. הפרמטר השני הוא הודעה שיש להציג אם הערך של המשתנה הראשון הוא *false*, כלומר אם הבדיקה נכשלת.

השתמש בבדיקות תקינות כדי לתעד הנחות וכדי לאתר תנאים לא צפויים. בדיקות תקינות יכולות לשמש לבדיקת הנחות שונות, כמו אלו:

- פרמטר שקיבלה פונקציה (או פרמטר *out* שהיא עומדת להחזיר) נמצא בטווח המותר.
- קובץ מובנה או רציף (stream) הינו פתוח (או סגור) כאשר הפונקציה מתחילה (או מסתיימת).
- קובץ מובנה או רציף נמצא בתחילתו (או בסופו) כשפונקציה מתחילה (או מסתיימת).
- קובץ מובנה או רציף פתוח לקריאה בלבד, לכתיבה בלבד, או לשתי פעולות אלו.
- ערך פרמטר מסוג input only אינו משתנה על ידי הפונקציה.
- מצביע אינו null.
- מערך או רצף אחר של אלמנטים מכיל לפחות מספר מסוים של ערכים.
- טבלה אותחלה באופן שהיא מכילה ערכים אמיתיים.
- רשימת אלמנטים ריקה (או null) כאשר פונקציה מתחילה (או מסתיימת).
- גרסה מתוחכמת במיוחד של פונקציה מחזירה את אותו ערך כמו המימוש הפשוט יותר שכתוב בצורה בהירה יותר.

כמובן שאלו הן דוגמאות של דברים בסיסיים בלבד בשעה שהפונקציות שלך עשויות להכיל הנחות רבות אחרות שתוכל לתעד באמצעות בדיקות תקינות.

בדרך כלל, אינך רוצה שהמשתמש יראה הודעות של בדיקות תקינות, אשר מיועדות בעיקר לפיתוח הקוד ולתחזוקתו. את בדיקות התקינות מהדרים בדרך כלל כחלק מהקוד בזמן פיתוח, והן אינן כלולות בקוד המשוחרר לייצור (גרסת production). בדיקות תקינות חושפות הנחות סותרות, מצבים לא צפויים, ערכים לא נכונים שהועברו לפונקציות וכד'. בגרסאות הייצור של הקוד לא כוללים אותן כדי לא לפגוע בביצועים של המערכת (זמני תגובה).

בניית מנגנון משלך לביצוע בדיקות תקינות

שפות תכנות רבות כוללות מנגנונים מובנים לבדיקת תקינות. אם השפה שאתה משתמש בה אינה תומכת ישירות בבדיקות תקינות, קל מאוד לכתוב אותן. המימוש הסטנדרטי של שפת C++ למשל אינו תומך בהודעות שגיאה. הנה לדוגמה מימוש משופר של ASSERT עבור C++ במבנה של מאקרו:

```
C++ Example of an Assertion Macro
#define ASSERT( condition, message ) {         \
  if ( !(condition) ) {                        \
    LogError( "Assertion failed: ",            \
      #condition, message );                   \
    exit( EXIT_FAILURE );                      \
  }                                            \
}
```

עקרונות מנחים לשימוש בבדיקות תקינות

הנה כמה עקרונות מנחים לשימוש בבדיקות תקינות:

השתמש בקוד לטיפול בשגיאות עבור שגיאות שאתה מצפה שיקרו; השתמש בבדיקות תקינות לדברים שאתה מצפה שלא יקרו לעולם. בדיקות תקינות (assertions) בודקות תנאים שאינם אמורים להתרחש *לעולם*. קוד שמטפל בשגיאות מיועד למצבים לא רגילים שלא עשויים להתרחש לעתים קרובות, אבל המתכנת צופה אותם ולכן הקוד הסופי אמור לטפל בהם. טיפול בשגיאות מתייחס בדרך כלל לזיהוי נתונים שגויים; בדיקות תקינות מטפלות בדרך כלל בשגיאות (באגים) בקוד התוכנית.

אם ניהול שגיאות עוסק במצב לא רגיל, הוא יאפשר לתוכנה להגיב לשגיאה בצורה מסודרת. אם בדיקת תקינות נכשלת במצב לא רגיל, הטיפול הנכון איננו טיפול בשגיאה בלבד, אלא כתיבה מחדש של הקוד ושחרור גרסה חדשה של התוכנה.

אפשר לחשוב על בדיקות תקינות כעל תיעוד שיכול לרוץ - אי אפשר לסמוך על זה שהן יגרמו לקוד לפעול, אבל הן יכולות לתעד את ההנחות בצורה אקטיבית יותר מאשר הערות בקוד.

שים לב לא להכניס הרצת קוד לתוך בדיקת תקינות. הכנסת קוד לבדיקת תקינות מעלה את האפשרות שהמהדר לא יכלול את קוד הבדיקה כאשר תכבה את בדיקות התקינות. בחן את הדוגמה הבאה:

```
Visual Basic Example of a Dangerous Use of an Assertion
Debug.Assert( PerformAction() ) ' Couldn't perform action
```

הבעיה עם הקוד הזה היא שאם אינך מהדר את בדיקות התקינות, אינך מהדר את הקוד שמבצע את הפעולה. כתוב את הפעולה בשורה נפרדת, הצב את התוצאות למשתנה ובדוק אותו. הנה דוגמה לשימוש בטוח בבדיקת תקינות:

```
Visual Basic Example of a Safe Use of an Assertion
actionPerformed = PerformAction()
Debug.Assert( actionPerformed ) ' Couldn't perform action
```

השתמש בבדיקות תקינות כדי לוודא ולתעד preconditions ו- postconditions. preconditions (**תנאי פתיחה**) ו-postconditions (**תנאי סיום**) הם חלק משיטת תכנות שנקראת "design by contract" ("**תכנון לפי חוזה/הסכם**") (Meyer 1997). כאשר נעשה

שימוש בסוגי תנאים אלה, כל פונקציה או מחלקה יוצרת כביכול "חוזה" עם שאר חלקי הקוד.

Preconditions - תנאי פתיחה שמבטיח קוד התוכנית שקוראת לפונקציה או יוצרת אובייקט, והם ההתחייבויות כביכול של קוד התוכנית כלפי הקוד שהיא קוראת לו.

Postconditions - תנאי הסיום שייתקיימו בסיום הריצה של הקוד, והם כביכול ההתחייבות של קוד התוכנית כלפי הקוד שהיא קראה לו.

בדיקות תקינות הן כלי טוב לתיעוד של תנאי פתיחה ושל תנאי סיום. אפשר לתעד זאת גם באמצעות הערות, אבל הערות אינן יכולות לבדוק באופן דינמי בזמן ריצה את הנכונות של תנאי הפתיחה ושל תנאי הסיום שנדרשו.

בדוגמה הבאה, בדיקות תקינות משמשות לתיעוד של תנאי פתיחה ושל תנאי סיום של הפונקציה *Velocity*. בדיקות התקינות שלהלן משמשות גם לתיעוד של תנאים שלפני ואחרי...

```
Visual Basic Example of Using Assertions to Document Preconditions and
Postconditions
Private Function Velocity ( _
  ByVal latitude As Single, _
  ByVal longitude As Single, _
  ByVal elevation As Single _
  ) As Single

  ' Preconditions
  Debug.Assert ( -90 <= latitude And latitude <= 90 )
  Debug.Assert ( 0 <= longitude And longitude < 360 )
  Debug.Assert ( -500 <= elevation And elevation <= 75000 )
  ...

  ' Postconditions
  Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )

  ' return value
  Velocity = returnVelocity
End Function
```

אם המשתנים *latitude*, *longtitude*, *elevation* היו מגיעים ממקור חיצוני, הרי שערכים לא תקינים היו צריכים להיות מטופלים בקוד של ניהול שגיאות ולא באמצעות בדיקות תקינות. אם הנתונים מגיעים ממקור פנימי שאנחנו בוטחים בו, והתכנון של הפונקציה מסתמך על כך שהערכים שלהם נכונים - הרי שבדיקות תקינות מתאימות יותר.

כדי לקבל קוד איכותי - בדוק תקינות ואז טפל בשגיאות. בכל מצב שגיאה צריכה להיות פונקציה לבדיקת תקינות או מנגנון לניהול שגיאות, אבל אין צורך בשניהם. יש מומחים שטוענים שרק סוג אחד נחוץ (Meyer 1997).

עם זאת, מערכות תוכנה בעולם האמיתי ופרויקטים אמיתיים נוטים להיות מבולגנים מכדי להסתמך על בדיקות תקינות בלבד. בפרויקט גדול שנמשך זמן רב, חלקים שונים בו עשויים להיות מתוכננים על ידי מתכננים שונים על פני תקופה של שנים אחדות ואפילו רבות. המתכננים לעתים מופרדים זה מזה על פני זמן ומרחב, ועל פני גרסאות שונות. התכנון שלהם עלול להתמקד בטכנולוגיות שונות שמופעלות בחלקים שונים של הפרויקט. הדברים חמורים יותר כאשר חלקי פרויקט נרכשים מגורם חיצוני. מתכנתים עשויים לפעול על פי נהלי עבודה וסטנדרטים שונים בחלקים שונים של המערכת. בצוות פיתוח גדול, באופן בלתי נמנע, חלק מהמפתחים יהיו קפדניים יותר מהאחרים, וחלקים שונים של הקוד יעברו בחינה מעמיקה יותר מחלקים אחרים. מתכנתים מסוימים יבנו בדיקות ניפוי מקיפות יותר מאחרים ולהיות נתונים למעקב של צוותי בדיקות שפועלים באזורים גיאוגרפיים שונים ואשר נתונים ללחצים עסקיים. לכל הדברים האלה יש השלכות על איכות התוצר ועל **היכולת לשחזר דברים** (regression testing).

בנסיבות כאלו, ניתן להשתמש גם בבדיקות תקינות וגם בטיפול בשגיאות כדי לבחון כל מצב אפשרי. בקוד של Microsoft Word למשל, תנאים שתמיד צריכים להיות נכונים נבדקים באמצעות בדיקות תקינות, אבל גם מטופלים באמצעות קוד ניהול שגיאות למקרה שהבדיקה נכשלת. בפרויקטים גדולים מורכבים וארוכים כמו Word, בדיקות תקינות הן כלי חשוב, משום שהן עוזרות להציף כמה שיותר שגיאות בתהליך הפיתוח. עם זאת, יש לדעת שהיישום מורכב מאוד (מיליוני שורות קוד) ועבר דורות של שינויים, ולכן אי אפשר לצפות לכך שכל טעות אפשרית תאותר ותתוקן לפני שגרסה חדשה שלו תצא לשוק. במקרה כזה חייב להיות בגרסת הייצור הליך של טיפול בשגיאות.

הנה דוגמה איך גישה כזו עשויה להיראות בפונקציה *Velocity*:

```
Visual Basic Example of Using Assertions to Document Preconditions and
Postconditions
Private Function Velocity ( _
  ByRef latitude As Single, _
  ByRef longitude As Single, _
  ByRef elevation As Single _
  ) As Single

  ' Preconditions
  Debug.Assert ( -90 <= latitude And latitude <= 90 )
  Debug.Assert ( 0 <= longitude And longitude < 360 )
  Debug.Assert ( -500 <= elevation And elevation <= 75000 )
  ...

  ' Sanitize input data. Values should be within the ranges asserted above,
  ' but if a value is not within its valid range, it will be changed to the
  ' closest legal value
```

קוד בדיקות תקינות