# A

## Abstraction - (הפשטה)

זהו, מושג בסיסי בפיתוח תוכנה המאפשר לנו לייצג מערכות מורכבות או רעיונות בצורה פשוטה ומוכללת. זה כרוך בהתמקדות במאפיינים ובהתנהגות החיוניים של אובייקט, תוך הסתרה או הפשטה של ​​הפרטים המיותרים

בתכנות, Abstractionמושגת באמצעות שימוש במחלקות Abstract וממשקים. מחלקה אבסטרקטית היא מחלקה שלא ניתנת לאינטראקציה והיא יכולה לשמש רק כבסיס למחלקות אחרות. הוא מספק ממשק משותף ומגדיר מספר התנהגויות שמחלקות יורשות יכולות לרשת וליישם. מחלקות Abstract מאפשרות לנו ליצור רמת הפשטה על ידי הגדרת המבנה וההתנהגות המשותפים שחולקים מספר מחלקות קשורות.

מצד שני, ממשק מגדיר חוזה של מתודות שעל מחלקה ליישם. זה מציין מה אובייקט יכול לעשות מבלי לציין איך זה נעשה. ממשקים מאפשרים לנו להגדיר קבוצה משותפת של מתודות שמחלקות שונות יכולות ליישם, לקדם צימוד רופף ולספק דרך להחליף יישומים שונים.

על ידי שימוש בהפשטה, אנו יכולים ליצור קוד מודולרי וניתן לתחזוקה. זה מאפשר לנו לעבוד ברמת הפשטה גבוהה יותר, תוך התמקדות בתכונות וההתנהגויות החיוניות של אובייקט או מערכת, במקום להסתבך בפרטי היישום. זה הופך את הקוד שלנו לגמיש יותר, ניתן להרחבה וקל יותר להבנה ולתחזוקה.

הנה דוגמה פשוטה להמחשת הפשטה באמצעות ממשק ב C#

// Define an interface for a Shape

public interface IShape

{

void Draw();

}

// Implement a concrete class Circle that implements the IShape interface

public class Circle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing a circle");

}

}

// Implement another concrete class Rectangle that also implements the IShape interface

public class Rectangle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing a rectangle");

}

}

// Usage of abstraction

public class DrawingApp

{

public void DrawShapes(List<IShape> shapes)

{

foreach (var shape in shapes)

{

shape.Draw();

}

}

}

// Usage

public static void Main()

{

var shapes = new List<IShape>();

shapes.Add(new Circle());

shapes.Add(new Rectangle());

var drawingApp = new DrawingApp();

drawingApp.DrawShapes(shapes);

}

בדוגמה זו, אנו מגדירים ממשק **IShape** המציין שיטה **Draw** . המחלקות **Circle** ו- **Rectangle** מיישמות שתיהן את ממשק **IShape** ומספקות יישום משלהן של שיטת **Draw** .

המחלקה **DrawingApp** משתמשת בהפשטה שמספקת ממשק **IShape** כדי לצייר צורות שונות. זה לא צריך לדעת את הפרטים הספציפיים של כל צורה; הוא רק צריך לדעת שניתן לצייר כל צורה על ידי הפעלת השיטה **Draw**

על ידי שימוש בהפשטה באמצעות ממשקים, אנו יכולים לעבוד עם צורות ברמת הפשטה גבוהה יותר, לפשט את הקוד שלנו ולהפוך אותו לגמיש יותר כדי להכיל צורות שונות בעתיד.

**Abstract class - מחלקה מופשטת**

מחלקה מופשטת היא מחלקה בתכנות מונחה עצמים (OOP) שאינה ניתנת למופע ישירות. היא מוכרזת באמצעות מילת המפתח ' abstract' ומשמשת כשרטוט או תבנית עבור מחלקות אחרות.

להלן כמה מאפיינים ושימושים מרכזיים של מחלקות abstract:

1. **לא ניתן ליצור מופע:** לא ניתן ליצור אובייקטים ישירות ממחלקה מופשטת. ה class קיים אך ורק למטרות ירושה, ומספק מבנה והתנהגות משותפים עבור תת המחלקות שלו.
2. **עשוי להכיל מתודות abstract ו non-abstract:** ל Abstract class יכולות להיות מתודות abstract ו non-abstract. מתודות abstract מוכרזות ללא מימוש ב Abstract class עצמה, בעוד שלמתודות non-abstract יש מימוש מוגדר. תת-מחלקות חייבות ליישם את המתודות abstract .
3. **מספק התנהגות נפוצה:** מחלקות מופשטות לרוב מגדירות התנהגות, מאפיינים או שיטות נפוצות שמחלקות משנה יכולות לרשת. זה מקדם שימוש חוזר בקוד ועוזר לבסס מבנה משותף בין כיתות קשורות.
4. **מאפשר overriding למתודות:** תת-מחלקות של מחלקה מופשטת יכולות לעקוף שיטות לא מופשטות (שיטות קונקרטיות) כדי לספק יישום משלהן. עם זאת, יש ליישם שיטות מופשטות בתת המחלקה.

# C

## Class - (מחלקה)

מחלקה היא תבנית ליצירת אובייקטים (instances) במחלקה, אתה מגדיר את ה Properties, המתודות והתכונות הפרטיות של האובייקטים שנוצרים ממנה.

מחלקה מתארת את המאפיינים וההתנהגות השייכים לאובייקטים של אותו הסוג. זה אומר שכל אובייקט מסוג מחלקה מחלק את המאפיינים והפונקציות המוגדרים במחלקה.

מחלקות נמצאות בתוך מרחבי שמות (namespace) והן ניתנות לירושה, כלומר, ניתן ליצור מחלקות שיורשות ממחלקות אחרות ולהוסיף או לשנות פונקציונליות.

מחלקות משמשות לאורגון קוד, ליצירת אובייקטים ספציפיים ולהגדרת תוכניות.

## CQRS

CQRS הוא ראשי תיבות של "Command Query Responsibility Segregation" בתרגום חופשי - "הפרדת אחריות בין פקודות לשאילתות". זוהי תבנית ארכיטקטורת תוכנה שמפרידה את התחומים של טיפול בפקודות (שינויים בנתונים) ושאילתות (שאילתת מידע) ביישום.

ביישום אוטונומי מסוגנן, נפתחת תצורה שבה לרוב יש לנו מודל נתונים אחד אשר משמש לקריאה וכתיבה. ב-CQRS, המודל הזה מתחלק לשניים:

1. **הצד של פקודות (הצד של הכתיבה):** הצד הזה של היישום אחראי לטיפול בפקודות, שהן פעולות שמשנות את מצב היישום. הפקודות כוללות פעולות כמו יצירה, עדכון או מחיקת נתונים. הצד של הפקודות נבנה בדרך כלל במבנה המאוד מערכתי ומתמקד בשמירה על אי־ תפוגתיות הנתונים. בדרך כלל הוא משתמש במערכת אחסון נפרדת המותאמת לכתיבה.
2. **הצד של שאילתות (הצד של הקריאה):** הצד הזה של היישום אחראי לטיפול בשאילתות, שהן בקשות לאחזור מידע. הצד של השאילתות מותאם לפעולות קריאה ובדרך כלל משתמש במערכת אחסון נפרדת או במבנה נתונים שונה לצורך שאילתות יעילות. הוא גם יכול לכלול תכונות כמו מטמון ונורמליזציה לשיפור ביצועי השאילתות.

CQRS מציבה הבחנה ברורה בין שני הצדדים הללו, וכך ניתן ליהנות ממספר יתרונות:

* **הקשבה להרצת פקודות (כתיבה):** ניתן להריץ את הצד של הפקודות ואת הצד של השאילתות בנפרד, בהתאם לדרישות המשמעותיות של כל צד. לדוגמה, אם ליישום יש תעבורת כתיבה גדולה, ניתן להגדיל את הצד של הפקודות לטובתו מבלי להשפיע על ביצועי השאילתות.
* **ביצוע מהיר של שאילתות:** על ידי אופטימיזציה של הצד של השאילתות לפעולות קריאה, ניתן להשיג ביצועים טובים יותר בשאילתות. זה חשוב ביישומים עם שאילתות מורכבות וכמויות מידע גדולות.
* **גמישות:** ניתן לבחור פתרונות וטכנולוגיות שונות לכל צד, בהתאם לדרישות הספציפיות של כל צד. זה מאפשר להשתמש בכלי הנכונים לכל משימה.
* **אבטחה:** ניתן ליישם הגנות אבטחה שונות לשני הצדדים, וכך לוודא שנתונים רגישים מוגנים בהתאם.

# E

## Encapsulation

המונח מתייחס ל "חשיפת פתרון לבעיה מבלי לדרוש מהצרכן להבין היטב את תחום הבעיה".

זו ההגדה שניתנה על ידי Mark Seeman

לדוגמה: כאשר מפתחים ב Windows Form וצריך לשים על המסך אלו שהם פקדים, לא צריך באמת לצייר אותם, אלא בסך הכל מניחים פקדים על משטח העיצוב והכל מוכן, זה נראה מאוד פשוט אבל לפקדים האלו יש המון פרטים מסובכים של איך בפועל הפקדים מתפקדים ומופיעים על המסך.

ישנן המון שכבות חבויות מאחורי הפקדים, סביבת הפיתוח מכילה את כל הדברים הקשורים לבניית ממשק המשתמש, לא צריך להבין את הפנימיות של אותן מסגרות כדי להשתמש בהן כדי להשיג את המטרות שיש לך.

Encapsulation מבוססת על שני מושגים עיקריים.

* הסתרת מידע והגנה משינויים.
* המחלקה הכיתה חייבת להסתיר את המידע שהיא מקפלת כדי להגן עליה מפני משתמשים שעלולים בטעות לשנות משהו ובכך לגרות לתקלות.

וכתוצאה מכך הקוד צריך להיות חסין תקלות.

# F

## Fragile - (שביר)

בהקשר של תכנות, "Fragile" מתייחס לקוד או תוכנה המועדים להישבר או להתקלקל בקלות. קוד שביר מאופיין לרוב ברגישותו לשינויים בסביבה, בנתוני קלט או בתלות. הוא עשוי להפגין התנהגות בלתי צפויה או קריסות כאשר הוא נחשף לשינויים קלים. קוד שביר יכול להיות מאתגר לתחזוקה ולניפוי באגים, מכיוון שאפילו לשינויים קטנים יכולים להיות השלכות לא מכוונות. כדי למתן את השבריריות, מפתחים שואפים לכתוב קוד חזק ועמיד שיכול להתמודד עם תרחישים שונים ולהתאושש בקלות משגיאות.

# H

## High abstraction

המונח מתייחס לרמת ההתרחבות או ההסתרה שבקוד מקורי. ברמה זו, קוד עם "High abstraction" מתמקד במהות המשימה או התהליך שהוא צריך לבצע (ברמה עקרונית או לוגית), ומנסה להסתיר את הפרטים הטכניים.

במילים אחרות, קוד עם "High abstraction" פועל על מנת להשיג מטרות עקרוניות גבוהות יותר ואינו מתעסק בפרטים טכניים מיוחדים שברמה נמוכה יותר. זה מאפשר לקוד להיות יותר קריא, גמיש, ומותאם לשימושים שונים.

לדוגמה, בקוד עם "High abstraction," אם יש מחלקה המתמקדת בניהול משימות פרטיות, היא עשויה לספק ממשק או שירות אשר מאפשר לקבוע משימה כלשהי ולא לדאוג לפרטים כמו השימוש באחסון פרטי, תחבורה פיזית וכדומה. כאשר משתמשים בממשק או בשירות כזה, ניתן להשתמש בו מבלי לדאוג לפרטי המימוש.

עבודה עם רמת ההתרחבות הנכונה עשויה להבטיח קוד מותאם יותר למשתמש, גמיש יותר, ופחות מסוכן בשינויים ותחזוקה.

# I

## Instance - (מופע)

זהו, אובייקט ספציפי שנוצר ממחלקה (class) כאשר נוצרת מחלקה (class) אתה מגדיר בעצם את התבנית ליצירת אובייקטים מסוג זה. כאשר אתה יוצר אובייקט ממחלקה כלשהי, התוכנית יוצרת מופע (instance) של אותה מחלקה. כל מופע הוא יחיד וכולל את המאפיינים והפונקציות שהוגדרו במחלקה.

לדוגמה, אם יש לך מחלקה בשם "Person" שמכילה מאפיינים כמו שם פרטי ושם משפחה, ולה מוגדרת פונקציה בשם "SayHello", כאשר אתה יוצר מופע של "Person", אתה יוצר instance של המחלקה "Person". הנה דוגמה:

Person person1 = new Person(); // יצירת מופע ריק של Person

person1.FirstName = "John";

person1.LastName = "Doe";

person1.SayHello(); // השימוש בפונקציה באמצעות המופע

Person person2 = new Person();

person2.FirstName = "Jane";

person2.LastName = "Smith";

person2.SayHello();

בדוגמה זו, person1 ו-person2 הם שני מופעים שנוצרו ממחלקה "Person". כל מופע מכיל את המאפיינים הפרטיים שלו (שם פרטי ושם משפחה) ויכול להשתמש בפונקציה "SayHello" שמוגדרת במחלקה "Person".

# L

## Loose coupling - (צימוד רופף)

זהו, עיקרון עיצובי בהנדסת תוכנה שמטרתו למזער את התלות בין רכיבים או מודולים של מערכת תוכנה. מקדם עצמאות, מודולריות וממשקים מוגדרים היטב בין רכיבים.

במערכת משולבת באופן רופף, רכיבים עצמאיים מקיימים אינטראקציה באמצעות ממשקים מופשטים, ללא צורך בידע מפורט על פעולתו הפנימית של אחד של השני.

גישת עיצוב זו מציעה מספר יתרונות:

1. **שימוש חוזר בקוד:** ניתן לעשות שימוש חוזר בקלות ברכיבים מחוברים באופן רופף בפרויקטים או בהקשרים שונים, לקדם שיתוף קוד ולהפחית את מאמצי הפיתוח.
2. **יכולת בדיקה:** בדיקת רכיבים בודדים הופכת קלה יותר מכיוון שניתן לבדוק אותם בבודדים ללא תלות נרחבת. ומשפר את האפקטיביות והיעילות של הבדיקה.
3. **מדרגיות:** מערכות משולבות רופפות ניתנות להרחבה יותר, מה שמאפשר הוספה או הסרה של רכיבים מבלי להשפיע על המערכת כולה. משפר את יכולת המערכת להתמודד עם עומס עבודה או ביקוש מוגבר.
4. **תחזוקה:** צימוד רופף מפשט תחזוקה ושינויים של המערכת. שינויים שנעשו ברכיב אחד נוטים פחות לגרום לבעיות לא מכוונות או שגיאות ברכיבים אחרים, מה שהופך את המערכת לקלה יותר לתחזוקה.

באופן כללי, צימוד רופף עוזר ליצור מערכות תוכנה מודולריות, גמישות וניתנות לתחזוקה בקלות, ומאפשר פיתוח, בדיקה והתפתחות קלה יותר של התוכנה.

## Low abstraction

מונח זה, מתייחס לרמת ההתרחבות או ההסתרה שבקוד מקורי. ברמה זו, קוד עם "Low abstraction" יודע הרבה פרטים טכניים ופרטי רמה נמוכה בהיררכיה הטכנית. זה אומר שהקוד מתמקד בפרטים טכניים ולא מסתיר אותם ברמה גבוהה יותר.

במילים אחרות, קוד עם "Low abstraction" יתמקד באיך משהו נעשה, ולא במהו נעשה. מה שעשוי להיות פחות קריא ופחות גמיש בטיפול בשינויים ובתחזוקה, כיוון שכל הפרטים הטכניים מוצגים ברמה נמוכה יותר ויכולים להראות כמועילים רק למפתחי הקוד או לטכנאים טכניים. במקום זה,

ואילו קוד המתבצע עם "High abstraction" ינסה להסתיר את הפרטים הטכניים ויתמקד במהות המשימה או התהליך שהוא צריך לבצע (ברמה עקרונית או לוגית).

עבודה עם רמת ההתרחבות הנכונה עשויה להבטיח קוד יותר קריא, גמיש יותר, ופחות מסוכן כאשר מבצעים שינויים ותחזוקה.

# M

## Module - (מודול)

מודול הוא יחידת קוד עצמאית שמכילה קוד, טיפוסים, פונקציות או משתנים.

מודולים משמשים לארגון ולסידור הקוד בפרויקט. הם מאפשרים את החלוקה של הקוד לפי נושאים או פונקציות.

מודולים אינם מייצרים אובייקטים כמו מחלקות. הם פשוט מכילים קוד ומשמשים לצורך ארגון קוד והבנתו.

מודולים לא יורשים מאחרים, ואינם משמשים ליצירת אובייקטים. הם נגישים לשימוש מקובץ הקוד הראשי של הפרויקט ומכל מקום אחר שבו נרצה להשתמש בהם.

בסיכום, הוא קובץ עצמאי שמכיל קוד ונתונים ומשמש לארגון קוד והבנתו.

בוא נניח שיש לך פרויקט ב-C# בו אתה רוצה ליצור מודולים לארגון קוד. הנה דוגמה לשני מודולים:

**מודול 1: MathUtils.cs**

using System;

namespace MyProject.Utilities

{

public static class MathUtils

{

// This function performs addition of two numbers

public static int Add(int a, int b)

{

return a + b;

}

// This function performs subtraction of two numbers

public static int Subtract(int a, int b)

{

return a - b;

}

}

}

**מודול 2 StringUtils.cs :**

using System;

namespace MyProject.Utilities

{

public static class StringUtils

{

// This function reverses the input string

public static string ReverseString(string input)

{

char[] charArray = input.ToCharArray();

Array.Reverse(charArray);

return new string(charArray);

}

// This function counts the number of words in the input string

public static int CountWords(string input)

{

string[] words = input.Split(new char[] { ' ', '\t', '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);

return words.Length;

}

}

}

בדוגמאות אלה, יש לנו שני מודולים בשם MathUtils ו-StringUtils. כל מודול מכיל פונקציות פשוטות עם הערות קוד באנגלית, המסבירות את מטרת כל פונקציה. הערות קוד אלו משמשות תיעוד טכני כדי להבהיר את המשמעות והשימוש של הפונקציות.

# O

## Overriding / Override - (דריסה או שיכתוב (כתיבה התוכן מחדש))

Override, בתכנות מונחה עצמים ,(OOP) מתייחס ליכולת של תת-מחלקה לספק מימוש שונה למתודה שכבר מוגדרת ב- superclass שלה.

בשפת התכנות ,C# שיכתוב של מתודה מאפשרת לתת-מחלקה לספק יישום משלה עבור מתודה שכבר מוגדרת ב superclass שלה. להלן הסבר על שיכתוב של מתודה בהקשר של C#

כאשר למתודה בתת-מחלקה יש את אותו שם, סוג החזרה ופרמטרים כמו למתודה במחלקה העל שלה, נאמר שהיא עוקפת את שיטת ה superclass השיטה הנדחקת בתת-המחלקה מספקת יישום מיוחד או שונה של השיטה שעברה בירושה ממחלקת העל.

נקודות מפתח לגבי **Overriding** מתודה:

1. **ירושה:** C# תומך בירושה, ומאפשר למחלקות לרשת ממחלקה בסיסית. כאשר תת-מחלקה יורשת ממחלקת -על, היא כוללת אוטומטית את השיטות המוגדרות במחלקת העל.
2. **מילות מפתח וירטואליות ועקיפות:** כדי לאפשר עקיפת מתודה , יש להצהיר על זה במתודה ה superclass עם מילת המפתח virtual. זה מצביע על כך שניתן לשכתב מחדש את המתודה על ידי תת מחלקה. בתת-המחלקה, המתודה שנכתבת מחדש מסומנת במילת המפתח override.
3. **חתימת מתודה:** המתודה העוקפת בתת-מחלקה חייבת להיות בעלת חתימת מתודה זהה (שם, סוג החזרה ופרמטרים) כמו המתודה הווירטואלית ב- superclass זה מבטיח שהמתודה המוחלטת מזוהה בצורה נכונה בזמן ריצה כאשר קוראים לאובייקט של תת-המחלקה.
4. **מגבילי גישה:** משנה הגישה של המתודה המשוכתבת בתת-המחלקה חייב להיות זהה או פחות מגביל מאשר משנה הגישה של המתודה הווירטואלית ב- superclass. לדוגמה, אם מתודת העל מוצהרת כ protected, המתודה העוקפת יכולה להיות protected או public בתת המחלקה.

להלן דוגמה הממחישה שיכתוב מתודה:

class Animal

{

public virtual void MakeSound()

{

Console.WriteLine("The animal makes a sound.");

}

}

class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("The dog barks.");

}

}

בדוגמה זו, למחלקה **Animal** יש מתודה וירטואלית בשם **MakeSound()** המספקת מימוש ברירת מחדל. המחלקה **Dog** יורשת מ- **Animal** ומשכתבת את המתודה **MakeSound()** כדי לספק יישום משלה.

על ידי שימוש בשיכתוב של מתודה, כאשר אנו יוצרים אובייקט של המחלקה **Dog** וקוראים למתודה **MakeSound()** הוא יבצע את המימוש המוגדר במחלקה **Dog** ולא את מימוש ברירת המחדל ב- **Animal** .

Animal animal = new Animal();

animal.MakeSound(); // Output: The animal makes a sound.

Animal dog = new Dog();

dog.MakeSound(); // Output: The dog barks.

בקוד שלמעלה, הקריאה הראשונה ל **MakeSound()** מבצעת את המימוש מהמחלקה **Animal** , בעוד שהקריאה השנייה, למרות שסוג האובייקט הוא **Animal**  מבצעת את המימוש העוקף מהמחלקה **Dog** ומתבצע פה פולימורפיזם.

וכך זה מדגים כיצד שיכתוב מתודה מאפשרת לתת-מחלקות לספק התנהגות מיוחדת משלהן תוך שמירה על ממשק משותף המוגדר ב- superclass.

# S

## Scalability - (מדגריות)

מושג זה, מתייחס ליכולת של מערכת, יישום תוכנה או תשתית להתמודד עם עומס עבודה הולך וגובר או התאמה לדרישות הולכות וגדלות מבלי לוותר על ביצועים, אמינות או חווית משתמש. זוהי היכולת של המערכת לטפל ביעילות בכמות מוגברת של עבודה או תעבורה ככל שהמערכת גדלה בגודלה או במורכבותה.

מדרגיות היא חיונית בתחומים שונים, כולל פיתוח תוכנה, מחשוב ענן, תשתית רשת ותפעול עסק. זה מבטיח שמערכת יכולה לטפל במשתמשים נוספים, נתונים, עסקאות או בקשות מבלי לחוות ירידה משמעותית בביצועים או בפונקציונליות.

ניתן להשיג מדרגיות בדרכים שונות:

1. **מדרגיות אנכית:** ידוע גם כ - scaling up, כרוך בהגדלת המשאבים (כגון CPU, זיכרון או אחסון) של מכונה או שרת בודדים. לדוגמה, הוספת זיכרון RAM נוסף או שדרוג למעבד מהיר יותר. למדרגיות אנכית יש לרוב מגבלות מכיוון שיש מגבלה מקסימלית לקיבולת החומרה.
2. **מדרגיות אופקית:** ידוע גם כ -scaling out, כולל הוספת עוד מכונות או שרתים בכדי לפזר את עומס העבודה על פני מספר מערכות. גישה זו מאפשרת ניצול טוב יותר של משאבים ויכולה להתאים לביקוש מוגבר על ידי הוספת כוח מחשוב נוסף. ניתן להשיג מדרגיות אופקית באמצעות טכניקות איזון עומסים ואשכולות.
3. **מדרגיות אלסטית:** גישה זו משלבת מדרגיות אנכית ואופקית כאחד, ומאפשרת למערכת להקצות משאבים באופן דינמי על סמך הביקוש הנוכחי. כרוך בהגדלה או הקטנה אוטומטית של משאבים בהתבסס על כללים מוגדרים מראש, כגון קנה מידה אוטומטי בסביבות מחשוב ענן.

מדרגיות חיונית לעסקים וארגונים מכיוון שהיא מבטיחה שהמערכות שלהם יכולות להתמודד עם צמיחה, פעילות מוגברת של משתמשים ודרישות משתנות. ומאפשר למערכת להישאר אמינה וביצועית גם תחת עומסים גבוהים, מספקת חווית משתמש חלקה ומונעת השבתה או צווארי בקבוק.

# T

## Tight coupling - (צימוד הדוק)

מוסג זה, מתייחס לרמה גבוהה של תלות הדדית והסתמכות בין רכיבים או מודולים בתוך מערכת תוכנה. במערכת מחוברת הדוקה, לרכיבים יש ידע ישיר על פעולתו הפנימית זה של זה והם קשורים זה בזה.

במערכת צמודה:

1. **תלות ישירה:** רכיבים מסתמכים במידה רבה על ממשקים, שיטות או מבני נתונים ספציפיים המסופקים על ידי רכיבים אחרים. הם מתייחסים ישירות וקוראים זה לפונקציות של זה או ניגשים למשתנים זה של זה.
2. **תלות הדדית גבוהה:** שינויים שנעשו ברכיב אחד דורשים לרוב שינויים מתאימים ברכיבים התלויים. שינוי או החלפה של רכיב אחד יכולים להיות בעלי אפקט מדורג על רכיבים אחרים, מה שיוביל למערכת מורכבת ושברירית.
3. **חוסר הפשטה:** רכיבים מחוברים בחוזקה חושפים את פרטי היישום הפנימיים שלהם זה לזה, ומגבילים את ה Encapsulationוה Abstraction. זה מקטין את האופי המודולרי של המערכת ומקשה על ההבנה והשינוי.
4. **הסתבכות קוד:** לשינויים שנעשו ברכיב אחד עלולות להיות השלכות לא מכוונות או שגיאות ברכיבים אחרים בשל הקשרים ההדדיים החזקים ביניהם. וזה הופך את בסיס הקוד למורכב ומאתגר יותר לתחזוקה.

Tight coupling יכול להוביל למספר חסרונות:

1. **גמישות מופחתת:** מערכות מחוברות הדוק הן פחות גמישות וניתנות להתאמה לשינויים. ביצוע שינויים או הצגת תכונות חדשות מצריך לעתים קרובות שינוי של מספר רכיבים, מה שהופך את המערכת לקשיחה וקשה לשינוי או הרחבה.
2. **שימוש חוזר מוגבל:** רכיבים מחוברים בצורה הדוקה(Tight coupling) הם מאתגרים לשימוש חוזר בהקשרים או פרויקטים שונים. שימוש חוזר ברכיב בודד עשוי לדרוש הכללת הרכיבים התלויים בו, הגברת המורכבות והפרעת שימוש חוזר בקוד.
3. **בדיקה קשה:** בדיקת רכיבים בודדים הופכת למאתגרת במערכת מחוברת הדוקה. התלות מקשה על בידוד ובדיקת יחידות בודדות, מה שעלול להוביל לבדיקה לא מלאה או לא יעילה.
4. **בעיות מדרגיות:** מערכות ב Tight coupling יכולות להתמודד עם אתגרים בעת הוספת תכונות או מודולים חדשים ועשויה לדרוש שינוי של מספר רכיבים, מה שהופך את המערכת פחות ניתנת להרחבה וקשה יותר לתחזוקה.

לעומת זאת, מערכת משולבת באופן רופף (Loose coupling), כפי שהוזכר קודם לכן, מקדמת עצמאות, מודולריות וממשקים מוגדרים היטב בין רכיבים, ומציעה יתרונות כמו שימוש חוזר בקוד, מדרגיות ותחזוקה.