תוכן עניינים

[מבוא 2](#_Toc531281310)

[Creational Patterns 3](#_Toc531281311)

[Builder 3](#_Toc531281312)

[Factory 6](#_Toc531281313)

[Simple Factory 6](#_Toc531281314)

[Factory Method 7](#_Toc531281315)

[Abstract Factory 9](#_Toc531281316)

[Singleton 13](#_Toc531281317)

[Structural Patterns 13](#_Toc531281318)

[Adapter 13](#_Toc531281319)

[Composite 15](#_Toc531281320)

[Decorator 17](#_Toc531281321)

[Facade 21](#_Toc531281322)

[Behavioural Patterns 22](#_Toc531281323)

[Iterator 22](#_Toc531281324)

[Observer 24](#_Toc531281325)

[State 25](#_Toc531281326)

[Template Method 28](#_Toc531281327)

[Other Patterns 30](#_Toc531281328)

[Dependency Injection 30](#_Toc531281329)

[Lazy Initialization 31](#_Toc531281330)

[Method Chaining 32](#_Toc531281331)

[Null Object 33](#_Toc531281332)

*בעז רוזובסקי*, 2017-2018

# מבוא

[*Design Patterns*](https://en.wikipedia.org/wiki/Software_design_pattern) נהיו מפורסמים אחרי שהספר  [*Design Patterns: Elements of Reusable Object-Oriented Software*](https://en.wikipedia.org/wiki/Design_Patterns_(book)) פורסם בשנת 1994 ע"י החבורה שמכונה “Gang of Four” או בקיצור ([חן](https://www.youtube.com/watch?v=0MxOfeB7Ykk)) “GoF”. בספר, ה-Design Pattern-ים מחולקים ל-3 חלקים: *Creational Patterns*, *Structural Patterns* ו-*Behavioral Patterns*.

Design Patterns*,* מה זה? Design Patterns הם פתרונות כללים שניתן לעשות בהם שימוש חוזר לבעיות נפוצות שחוזרות על עצמן. הן אינן קוד בפני עצמו שניתן להדר (compile), אלא הם מייצגים תבנית קוד לפתרון בעיה שניתן ליישם בכל מיני מצבים. הם נועדו כדי לעזור לכתוב קוד קריא יותר, שקל לתחזק ולשנות אותו (Maintainable, Readable, Scalable או Maintainability, Readability, Scalability).  
בנוסף, הם מקנים שפה משותפת בין מתכנתים המכירים בהם, לדוגמה כאשר מתכנת קורא קוד חדש, ונאמר לו כי השתמשו ב- Design Pattern(s)האלה והאלה, יהיה לו קל יותר להבין את הקוד. כמו כן, הם יכולים למנוע בעיות עתידיות.

אולם, יש לציין שרובם מנפחים את הפרוייקט מפני שהם צורכים יצירת מחלקות רבות יותר, והם אינם בהכרח גורמים לקוד להיות יעיל יותר. יש מקרים בהם ההפך הוא הנכון, הקוד נהיה קצת פחות יעיל מכיוון שיש לו עלויות נלוות (overhead) (לרוב ניתן להזניח זאת), אולם חשוב לציין כי יש Design Patterns מסוימים שתפקידם הוא לעייל את הביצועים.  
בנוסף, יש הטוענים כי Design Patterns מרמזים על יכולות חסרות בשפות התכנות, הרי שבשפות מסוימות, חלק מה- Design Patternsנהפכים לפשוטים יותר ליישום בקצת או בהרבה, ובחלק מהמקרים בנויים בשפה עצמה, ואין צורך ליישם אותם בעצמנו עד כדי שליישם אותם בעצמנו רק יגרום לקוד מסובך ופחות יעיל בהרבה.  
יתר על כן, יש מצבים ששניים או יותר Design Patterns מחמיאים אחד לשני בצורה נפלאה, ויש מצבים שהם מנוגדים אחד לשני לחלוטין, אפילו אותם Design Patterns שבמצב אחר יוצרים שילוב מעולה. כמו כן, שימוש מופרז או לא חכם ב- Design Patternsיכול לגרום להפך מהתוצאה הרצויה: קוד מסובך, בלתי קריא ושקשה לתחזוקה ושינוי.  
שימו לב, שיכול להיות שבמצבים פשוטים מסוימים, שימוש ב-Design Pattern מסוים שבדרך כלל מתאים למצב כזה, יהפוך את הקוד למסורבל ומסובך יתר על המידה מרוב הפשטות של המקרה.

**\* הערה:**יכול להיות שכאשר מתחילים לקרוא Design Pattern מסוים, יהיה יותר קל לקרוא את ההסבר הכללי ואת אופן המימוש באופן כללי, לאחר מכן לדלג לדוגמות ורק אחר כך לחזור לקרוא את ההסבר באופן מפורט יותר.

**\* הערה:**כל הדוגמות שרשומים במסמך זה הם לפי הבנתי ומייצגים את איך שאני רואה את הדברים. גם ב-Design Patterns אין דברים מוחלטים ויש אזורים אפורים, ויכול להיות שמחלקה תיישם מספר Design Patterns בו זמנית. ניתן למצוא את רוב דוגמות הקוד בפתרון המצורף.

**\* הערה:**לא רשמתי פה את כל ה-Design Patterns שקיימים, ובפרט לא את כל אלה שמופעים בעמוד ה-Wikipedia משום שאינני למדתי אותם בקורס או שלא הבנתי חלק מהם עד הסוף ואיני רוצה להטעות. אולם, אני כן אציין כאלה שנראים לי חשובים, לא בהכרח לפי סדר חשיבותם (יש כאלה שאיני יודע עליהם כלום (אולם יודע על קיומם בשם בלבד), ולכן אני לא יודע את משמעותם או גודל ערכם, ומשום כך רק אציין אותם ברשימה זו:

* Creational Patterns
* Structural Patterns
  + [Flyweight](https://en.wikipedia.org/wiki/Flyweight_pattern)
  + [Bridge](https://en.wikipedia.org/wiki/Bridge_pattern)
  + [Proxy](https://en.wikipedia.org/wiki/Proxy_pattern)
* Behavioral Patterns
  + [Chain of Responsibility](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern)
  + [Command](https://en.wikipedia.org/wiki/Command_pattern)
  + [Mediator](https://en.wikipedia.org/wiki/Mediator_pattern)
  + [Memento](https://en.wikipedia.org/wiki/Memento_pattern)
  + [Strategy](https://en.wikipedia.org/wiki/Strategy_pattern)
  + [Visitor](https://en.wikipedia.org/wiki/Visitor_pattern)

**\* הערה:**רשימה של Design Patterns של GoF שאני לא מבין אותם כמעט או נראים לי פחות חשובים:

* Creational Patterns
  + [Prototype](https://en.wikipedia.org/wiki/Prototype_pattern)  
    (מובנה ב-javascript. כל העצמים מיישמים את ה-design pattern הזה באופן טבעי. מכיוון שאין שם הורשה, זו למעשה הדרך הכי קרובה לשיים זאת, אם כי זו לא באמת הורשה.)
* Structural Patterns
* Behavioral Patterns
  + [Interpreter](https://en.wikipedia.org/wiki/Interpreter_pattern)

# [Creational Patterns](https://en.wikipedia.org/wiki/Creational_pattern)

אלו הם Design Patterns שמתעסקים עם יצירת עצמים. הם נותנים מנגנונים שונים ליצירת עצמים בצורה שמתאימה למצבים שונים, מכיוון שבחלק מהמקרים, הצורה הבסיסית ביותר ליצירת עצמים (בנאי) יכולה ליצור בעיות עתידיות או להוסיף סיבוך לתחזוקת הקוד. הם מבוססים על שני רעיונות עיקריים: הפרדת הקוד מידיעת הסוג הממשי של העצם והפרדת אופן יצירת העצמים.

## [Builder](https://en.wikipedia.org/wiki/Builder_pattern)

אם יצירה של עצם מורכבת ממספר חלקים, נוכל להשתמש ב-Builder כדי ליצור את העצם חלק חלק ובסוף לקבל אותו כשלם.

דוגמות בספרית ה-.NET:

* StringBuilder – שרשור מחרוזות מספר פעמים רב הוא לא יעיל וצורך משאבים רבים, גם מבחינת זיכרון וגם מבחינת כוח עיבוד, לכן StringBuilder יוצר את המחרוזת הסופית צעד צעד בצורה זולה בהרבה משתי הבחינות.  
  אולם חשוב לדעת כי אם משרשרים מחרוזות מספר נמוך מאוד של פעמים (תלוי גם באורך), כנראה יהיה זול יותר להשתמש בשרשורים רגילים.
* DbConnectionStringBuilder – מהווה מחלקת בסיס ל-builder-ים של מחרוזות חיבור למסדי הנתונים השונים כגון SqlConnectionStringBuilder הבונים מחרוזות חיבור מפתח מפתח בזמן ריצה.
* מחלקות מ-System.Reflection.Emit – namespace המכיל מחלקות שבעזרתן ניתן ליצור assembly בזמן ריצה: AssemblyBuilder, TypeBuilder, MethodBuilder וכדומה.

דוגמה:  
יש לנו מחלקה המייצגת אימייל ואנחנו צריכים שהיא תהיה [immutable](https://en.wikipedia.org/wiki/Immutable_object):

|  |
| --- |
| public class Email {  public string Subject { get; }  public string Body { get; }  public string From { get; }  public string[] To { get; }   public string Cc { get; }  public string Bcc { get; }  public string[] Attachments { get; }   public Email(string subject, string body, string from, string[] to)  {  this.Subject = subject;  this.Body = body;  this.From = from;  this.To = to;  }   public Email(string subject,  string body,  string from,  string[] to,  string cc,  string bcc,  string[] attachments)  {  this.Subject = subject;  this.Body = body;  this.From = from;  this.To = to;   this.Cc = cc;  this.Bcc = bcc;  this.Attachments = attachments;  } } |

יש איתה מספר בעיות:

* אם נעשה בנאי לכל אפשרות של הפרמטרים האופציונליים, יהיו לנו המון בנאים, ויהיה קשה לזכור איזה בנאי עושה מה. כמו כן לא בהכרח נוכל לעשות בנאי לכל קומבינציה מפני שיהיו שתי קומבינציות או יותר עם אותן החתימות. בנוסף, יהיה קשה מאוד להוסיף פרמטר אופציונאלי נוסף מכיוון שנצטרך ליצור המון בנאים נוספים.
* לא כל המאפיינים הם חובה ולכן, אם אני רוצה לתת ערך רק לחלק, אני חייב לציין גם את השאר
* זה קשה לזכור את סדר הפרמטרים
* מכיוון שאנו רוצים שהמחלקה תהיה immutable, חובה לדעת את כל המידע מראש לפני יצירת המופע.

\*ב-C# ניתן גם לתת ערכים לפי סדר שאנחנו קובעים כך:

|  |
| --- |
| var email = new Email(to: new string[] { "Genji" },  body: "Simple geometry",  from: "Hanzo",  subject: "Shimada",  cc: "Tracer",  attachments: null,  bcc: null); |

כעת ניצור את מחלקת ה-Builder שלנו:

|  |
| --- |
| public class EmailBuilder {  public EmailBuilder() : this(new List<string>(), new List<string>()) { }  public EmailBuilder(List<string> to, List<string> attachments)  {  this.To = to;  this.Attachments = attachments;  }   public string Subject { get; set; }  public string Body { get; set; }  public string From { get; set; }  public List<string> To { get; }   public string Cc { get; set; }  public string Bcc { get; set; }  public List<string> Attachments { get; }   public void AddTo(string to)  {  this.To.Add(to);  }  public void AddRangeTo(IEnumerable<string> collection)  {  this.To.AddRange(collection);  }   public void AddAttachment(string attachment)  {  this.Attachments.Add(attachment);  }  public void AddAttachments(IEnumerable<string> attachments)  {  this.Attachments.AddRange(attachments);  }   public Email Build()  {  return new Email(this.Subject,  this.Body,  this.From,  this.To.ToArray(),  this.Cc,  this.Bcc,  this.Attachments.ToArray());  } } |

נראה שבעזרת ה-Builder אין צורך לזכור את סדר הפרמטרים של הבנאי של המחלקה שלנו, ואין צורך לציין את כל הפרמטרים שאינם חובה במידה ויש לנו אחד כזה (Cc במקרה שלנו):

|  |
| --- |
| Email email = new EmailBuilder() {  From = "Hanzo",  To =  {  "Genji"  },  Subject = "Shimada",  Body = "Simple geometry",  Cc = "Tracer" }.Build(); |

כמו כן, אם נשמור מצביע ל-Builder, נוכל גם לקבוע את הערכים בצורה דינאמית יותר לפי רצוננו.

## Factory

יצירת עצמים על ידי בנאי ישירות בתוך המחלקה שמשתמשת בהם זה לא גמיש מפני שזה מקבע את המחלקות של העצמים האלו בתוך המחלקה ויוצר מצב שאין אפשרות לשנות את יצירת המופע מבחוץ אליה, חייבים לשנות את המחלקה עצמה.  
בנוסף, הדבר יכול ליצור בעיות נוספות, כגון: זה עלול ליצור שכפול קוד, יצירת המופע עלולה לדרוש מידע שלא נגיש לקוד (כמו שבירת [כמיסות](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)), מחלקה פרטית שצריכה להיות נגישה רק למחלקה מסוימת), יצירת המופע צורכת לוגיקה נוספת (או רבה, ולכן קוד היצירה בתוך המחלקה יהיה מסורבל), זה לא לוגי לשים את יצירת העצם שם (מכיוון שזה לא חלק מהתפקיד של הקוד).

ה-Factory בא לפתור את הבעיות האלו על ידי ייפוי יצירת העצם למקום אחר, ובכך מפריד בין שימושו לבין יצירתו, מאפשר לבחור את המחלקה של העצם שנוצר בזמן ריצה באופן דינאמי מבלי שהקוד שמשתמש בו בהכרח חייב לדעת את המחלקה המדויקת של העצם מאחר ורק אכפת לו מהממשק המופשט שלו שבו הוא משתמש.  
כתוצאה מכך, בעזרת ה-Factory ניתן ליצור מחלקות חדשות מבלי לשנות את הקוד שמשתמש בהן.

\*יש לציין שאין design pattern קיים הנקרא “Factory”. מילה זו היא מונח שמשנה את משמעותה לפי ההקשר, היא יכולה להתייחס למספר דברים. אולם יש 2 design pattern-ים (בעלי שם שונה) שהם “Factory”.

ה-“Factory” מחולק ל-3 חלקים (מסודרים לפי רמת המופשטות, מורכבות וגמישות, בסדר עולה כך שהכי פחות מופשט, מורכב וגמיש ראשון והכי מופשט, מורכב וגמיש אחרון). חשוב לציין שככל שרמת המופשטות עולה, כך הפרוייקט יצרוך יותר מחלקות.

### Simple Factory

\* זהו אינו design pattern רשמי, אולם זהו מקרה נפוץ ושימושי, לכן ראיתי לנכון כי הוא מספיק חשוב בשביל לציינו.

ה-Simple Factory הוא פעולה (לרוב סטטית) במחלקה אשר יוצרת עצם לפי פרמטרים מסוימים. לחילופין, מחלקה (בין אם היא סטטית או לא) שעיקר תפקידה היא להכיל פעולות כאלו. במקרה שהיא לא סטטית, גם היא נקראת “Factory”.  
יש לציין שבשיטה זו, קיימים מצבים בהם נדע את המחלקה המדויקת של העצם אותו, ויש מקרים בהם איננו נדע את המחלקה המדויקת, ובמקרה כזה היא העצם יממש מחלקה מופשטת או interface שדרכם ניתן לבצע את הפעולות הדרושות (משפחה אחת של עצמים).

יש לציין שבשיטה זו, עדיין קשה לשנות את סוג העצם שנוצר מחוץ לקוד של המחלקה.

דוגמות בספרית ה-.NET:

* Guid.NewGuid – יוצר GUID חדש (לא ריק)  
  (GUID ריק - {00000000-0000-0000-0000-000000000000})  
  (במקרה זה אנו יודעים את הסוג המדויק של העצם, וגם יש לוגיקה מסובכת לשם יצירתו (הפעולה מייפה את יצירת ה-GUID לפעולה [CoCreateGuid](https://msdn.microsoft.com/en-us/library/windows/desktop/ms688568(v=vs.85).aspx) מה-win32 API))
* TaskFactory, TaskFactory<T> – מכילים פעולות רבות שכל פעולה היא דוגמה ל-Simple Factory, במקרה זה הן לא סטטיות. כתוצאה מכך, המחלקות נחשבות כ-“Factory”.
* ההעמסות השונות של XmlReader XmlReader.Create
* ההעמסות השונות של XmlWriter XmlWriter.Create
* ההעמסות השונות של FileStream File.Create – יוצר קובץ חדש או דורס קיים לקריאה וכתיבה
* ההעמסות השונות של FileStream File.Open – פותח קובץ קיים לקריאה בלבד, כתיבה בלבד או שניהם
* FileStream File.OpenRead – פותח קובץ קיים לקריאה בלבד
* FileStream File.OpenWrite – פותח קובץ קיים או יוצר חדש לכתיבה בלבד
* StreamReader File.CreateText – דורס קובץ קיים או יוצר חדש לכתיבת טקסט מקודד UTF-8
* StreamWriter File.OpenText – פותח קובץ לקריאת טקסט מקודד UTF-8
* StreamWriter File.AppendText – פותח קובץ קיים או יוצר חדש לכתיבת טקסט מקודד UTF-8, אם הקובץ קיים, מוסיף מהסוף של הקובץ

### [Factory Method](https://en.wikipedia.org/wiki/Factory_method_pattern)

\* זהו design pattern של GoF.

נשתמש ב-Factory Method כאשר אנחנו רוצים להשתמש בעצם, אך איננו יודעים את הסוג הממשי שלו בזמן הידור, נדע זאת רק בזמן ריצה.

אופן היישום:  
נגדיר פעולה לא סטטית במחלקה לא סגורה (sealed) ומחלקות בנים דורסות אותה, ובכך משנות את סוג העצם שנוצר. דבר זה מאפשר לייפות את יצירת המופע של העצם לזמן ריצה, וגם מאפשר למחלקת האם להשתמש בפעולה זו כדי ליצור עצם מסוג שהיא אינה יודעת את סוגו בזמן ההידור ולהשתמש בממשק המופשט שלו מאחר ולא אכפת לה מהסוג הממשי שלו.  
חשוב לציין שבדומה ל-Simple Factory, ה-Factory Method מתעסק עם יצירה של עצמים ממשפחה אחת בלבד. אולם בשונה מ-Simple Factory, סוג העצם שנוצר יכול להשתנות מחוץ לקוד של המחלקה ולפי הצורך של מחלקת הבן.

דוגמות בספרית ה-.NET:

* DbCommand DbConnection.CreateDbCommand – יוצר פקודת SQL חדשה המתאימה למסד הנתונים של החיבור שקוראים עליו את הפעולה, לדוגמה SqlConnection יחזיר SqlCommand המתאים רק למסד נתונים של SQL Server, אך לא מתאים למסד נתונים של Oracle לדוגמה. פעולה זו מתעסקת עם המשפחה של פקודות SQL למסדי נתונים מסוגים שונים.

דוגמה:  
אנחנו רוצים לכתוב מחלקה המייצגת פיצריות שתפקידיהן הם להכין פיצות ולייצא משלוחים, ועל כל פיצרייה להכין פיצה מהסוג שמיוחד לה. איך נעשה זאת?  
נתחיל בליצור את מחלקות הליבה:

|  |
| --- |
| public abstract class Pizza {  public Pizza(string[] toppings)  {  this.Toppings = toppings;  }   public string[] Toppings { get; }  public abstract string Type { get; } }  public class IsraeliPizza : Pizza {  public IsraeliPizza(string[] toppings) : base(toppings) { }   public override string Type => "Israeli Pizza"; }  public class ItalianPizza : Pizza {  public ItalianPizza(string[] toppings) : base(toppings) { }   public override string Type => "Italian Pizza"; } |

כעת נגדיר את מחלקת הבסיס לפיצרייה:

|  |
| --- |
| public abstract class PizzaPlace {  protected abstract Pizza MakePizza(string[] toppings);   public void OrderPizza(string[] toppings, string address)  {  Pizza pizza = this.MakePizza(toppings);  this.DeliverPizza(pizza, address);  }   protected virtual void DeliverPizza(Pizza pizza, string address)  {  Console.WriteLine("An " + pizza.Type + " was delivered to " + address);  } } |

נסיים בליצור את מחלקות הבנים שייצגו את סוגי הפיצריות השונות (פיצרייה ישראלית ופיצרייה

איטלקית):

|  |
| --- |
| public class IsraeliPizzaPlace : PizzaPlace {  protected override Pizza MakePizza(string[] toppings) => new IsraeliPizza(toppings); }  public class ItalianPizzaPlace : PizzaPlace {  protected override Pizza MakePizza(string[] toppings) => new ItalianPizza(toppings); } |

קוד הלקוח:

|  |
| --- |
| PizzaPlace israeliPizzaPlace = new IsraeliPizzaPlace(); israeliPizzaPlace.OrderPizza(new string[] { "Olives" }, "Tel Aviv");  Console.WriteLine();  PizzaPlace italianPizzaPlace = new ItalianPizzaPlace(); italianPizzaPlace.OrderPizza(new string[] { "Onion" }, "Milan");  Console.WriteLine(); |

### [Abstract Factory](https://en.wikipedia.org/wiki/Abstract_factory_pattern)

\* זהו design pattern של GoF.

המטרה של ה-Abstract Factory היא לספק ממשק עבור יצירת עצמים ממשפחות הקשורות אחת לשנייה מבלי לציין את סוגם הממשי.  
בדומה ל-Factory Method, אנחנו רוצים עצם מסוים, אך איננו יודעים את הסוג המדויק שאנו רוצים, ונוכל לדעת זאת רק בזמן ריצה.

אופן היישום:  
בניגוד ל-Factory Method, נבצע זאת על ידי שימוש ב interface או במחלקה מופשטת הממומשים במחלקות נפרדות שתפקידן הן ליצור את העצמים.  
הממשק של ה-Abstract Factory יגדיר מספר פעולות שמחזירות עצם ממשפחה אחת בלבד כך שהעצמים ממשפחה מסוימת הם בעלי interface משותף שהם ממשים או מחלקה מופשטת משותפת שהם יורשים ממנה. כל פעולה תחזיר עצמים רק בצורתם המופשטת (מצביע לעצם רק בתור ה-interface או המחלקה המופשטת הנ"ל).

מחלקה שרוצה להשתמש ב-Abstract Factory, תכיל אותו בתור שדה (או מאפיין) ותייפה את יצירת העצמים אליו ולפעולות שהממשק שלו מגדיר כדי ליצור את העצמים הנדרשים. יצירת העצמים מייופת לזמן הריצה לעצם נפרד שזהו תפקידו. בדומה ל-Factory Method, לקוד שמשתמש בו לא אכפת מהסוג הממשי של העצמים מאחר שהוא רק משתמש בממשק המופשט שלו.

בעקבות זאת, ה-Design Pattern Abstract Factory מאפשר להחליף מימושים שונים של עצמים (שכל אחד הוא Abstract Factory שמחזיר מימושים שונים של ה-interface-ים או המחלקות המופשטות) מבלי לשנות את הקוד שמשתמש בהם. ניתן אפילו להחליף Abstract Factory בזמן ריצה מאחר שהוא עצם לכל דבר.

אם כך, יש לנו מספר מחלקות ממשיות שהן Abstract Factory, כל אחת מחזירה עצמים הקשורים לנושא הכללי שלה. כתוצאה מכך, בעזרת ה-Abstract Factory ניתן ליצור עצמים ממספר משפחות שונות הקשורות אחת לשנייה ושיש להם נושא כללי משותף, ובעצם נוצר מצב שיש לנו משפחה של מחלקות של ה-Abstract Factory, וכל המשפחות שכל Abstract Factory יחיד (מחלקה שמממשות אותו) מאגד, גם הן קשורות לנושא כללי עוד יותר.

\* לסיכום ההבדל בין Abstract Factory ל-Factory Method:

* Factory Method משמש ליצירת משפחה אחת בלבד של עצמים, לעומת זאת Abstract Factory משמש ליצירה של מספר משפחות הקשורות אחת לשנייה
* ה-Factory Method היא פעולה, לעומת Abstract Factory שהוא interface (או מחלקה מופשטת). יש משפחה משפחה של מחלקות של Abstract Factory עם מספר פעולות שהמטרה של כל פעולה היא ליצור עצם ולהחזיר אותו בצורתו המופשטת.
* ה-Abstract Factory לרוב ימומש על ידי מספר Factory Method-ים, אולם ניתן גם לממש אותו בעזרת Prototype.
* ה-Factory Method משתמש בהורשה כדי לייפות את יצירת העצמים לפעולה אחרת שניתנת לדריסה בעזרת הורשה, לעומת זאת ה-Abstract Factory מגדיר שייפוי יצירת העצמים הוא בעזרת הכלה של עצם אחר שיוצר את העצמים (הכלה לעומת הורשה).

\* חומר נוסף על ההבדל ביניהם:

* <https://stackoverflow.com/questions/1001767/what-is-the-basic-difference-between-the-factory-and-abstract-factory-patterns>
* <https://stackoverflow.com/questions/5739611/differences-between-abstract-factory-pattern-and-factory-method>

דוגמות בספרית ה-.NET:

* DbProviderFactory – מהווה מחלקת בסיס עבור מחלקות אשר יוצרות מופעים של הדברים השונים הנדרשים כדי לעבוד עם מסדי נתונים שונים כגון: חיבור, פקודה, פרמטר וכו'.  
  שימו לב כי נוצרה משפחה של Abstract Factory שמתעסקת עם מסדי נתונים וכל מימוש שלה יוצר מופעים של משפחות הקשורות למסד הנתונים מהסוג שלה (MSSQL, Oracle וכו'). המשפחות הנ"ל הם משפחת החיבורים (DbConnection), פקודות (DbCommand) וכו'. מימוש לדוגמה SqlClientFactory

דוגמות:

* יש לנו בעיה של יצירת פקדים שונים בין מערכות הפעלה שונות. לדוגמה נצטרך כפתורים ותיבות טקסט שונות עבור Windows, Linux ו-Mac.  
  נדגים את זה כאן בגוש אחד בלי יותר מדי הסברים.

|  |
| --- |
| public interface IGuiControl {  void Draw(); }  public interface IButton : IGuiControl {  string Text { get; set; }  event EventHandler Click; }  public interface ITextBox : IGuiControl {  string Text { get; set; }  event EventHandler TextChanged;  event KeyPressEventHandler KeyPress; }  public interface IGuiControlsFactory {  IButton CreateButton();  ITextBox CreateTextBox(); }  public class WindowsButton : IButton {  public string Text { get; set; }   public event EventHandler Click;   public void Draw() => Console.WriteLine("Drawing Windows button..."); }  public class WindowsTextBox : ITextBox {  public string Text { get; set; }   public event EventHandler TextChanged;  public event KeyPressEventHandler KeyPress;   public void Draw() => Console.WriteLine("Drwaing Windows text box..."); }  public class LinuxButton : IButton {  public string Text { get; set; }   public event EventHandler Click;   public void Draw() => Console.WriteLine("Drawing Linux button..."); }  public class LinuxTextBox : ITextBox {  public string Text { get; set; }   public event EventHandler TextChanged;  public event KeyPressEventHandler KeyPress;   public void Draw() => Console.WriteLine("Drwaing Linux text box..."); }  public class MacButton : IButton {  public string Text { get; set; }   public event EventHandler Click;   public void Draw() => Console.WriteLine("Drawing Mac button..."); }  public class MacTextBox : ITextBox {  public string Text { get; set; }   public event EventHandler TextChanged;  public event KeyPressEventHandler KeyPress;   public void Draw() => Console.WriteLine("Drwaing Mac text box..."); }  public class WindowsControlsFactory : IGuiControlsFactory {  public IButton CreateButton() => new WindowsButton();  public ITextBox CreateTextBox() => new WindowsTextBox(); }  public class LinuxControlsFactory : IGuiControlsFactory {  public IButton CreateButton() => new LinuxButton();  public ITextBox CreateTextBox() => new LinuxTextBox(); }  public class MacControlsFactory : IGuiControlsFactory {  public IButton CreateButton() => new MacButton();  public ITextBox CreateTextBox() => new MacTextBox(); } |

קוד הלקוח (נבנה את הקוד מחדש עבור כל platform עם ה-preprocessor השונים

לפי הפלטפורמה):

|  |
| --- |
| IGuiControlsFactory guiControlsFactory; #if WINDOWS guiControlsFactory = new WindowsControlsFactory(); #elif LINUX guiControlsFactory = new LinuxControlsFactory(); #elif MAC guiControlsFactory = new MacControlsFactory(); #else throw new NotImplementedException("Code should not reach here."); #endif  IButton button = guiControlsFactory.CreateButton(); button.Text = "My Button"; button.Draw(); |

* כעת נממש את הדוגמה מ-Factory Method בעזרת Abstract Factory. ההבדל הוא שפה תהיה לנו רק פיצרייה אחת שיכולה לשנות את סוגי הפיצות שהיא מכינה באופן דינאמי.  
  נראה פה רק את הקוד שהשתנה:

נתחיל בלהגדיר interface שישמש כממשק משותף בין ה-Abstract Factory שלנו וגם את המימושים שלו.

|  |
| --- |
| public interface IPizzaMaker {  Pizza MakePizza(string[] toppings); }  public class IsraeliPizzaMaker : IPizzaMaker {  public Pizza MakePizza(string[] toppings) => new IsraeliPizza(toppings); }  public class ItalianPizzaMaker : IPizzaMaker {  public Pizza MakePizza(string[] toppings) => new ItalianPizza(toppings); } |

נשנה את הקוד של PizzaPlace כדי להתאים אותו לשימוש ב-Abstract Factory:

|  |
| --- |
| public class PizzaPlace {  public PizzaPlace(IPizzaMaker pizzaMaker)  {  this.PizzaMaker = pizzaMaker;  }   private IPizzaMaker PizzaMaker { get; }   public void OrderPizza(string[] toppings, string address)  {  Pizza pizza = this.PizzaMaker.MakePizza(toppings);  this.DeliverPizza(pizza, address);  }   protected virtual void DeliverPizza(Pizza pizza, string address)  {  Console.WriteLine("An " + pizza.Type + " was delivered to " + address);  } } |

קוד הלקוח:

|  |
| --- |
| var israeliPizzaPlace = new PizzaPlace(new IsraeliPizzaMaker()); israeliPizzaPlace.OrderPizza(new string[] { "Extra Cheese", "Mushrooms" }, "Ramat Gan");  Console.WriteLine();  var italianPizzaPlace = new PizzaPlace(new ItalianPizzaMaker()); italianPizzaPlace.OrderPizza(new string[] { "Corn" }, "Rome");  Console.WriteLine(); |

ניתן לשם לב כי (באופן תיאורטי), נוכל לשנות את ה-Abstract Factory שכל מופע של PizzaPlace שמשתמש בו בזמן ריצה על ידי שינוי המאפיין PizzaMaker.

## [Singleton](https://en.wikipedia.org/wiki/Singleton_pattern)

ה-Singleton היא מחלקה אשר יש לה רק מופע אחד שהמחלקה שולטת ביצירתו (לקוד חיצוני אין שליטה על יצירה של מופעים של מחלקה זו). דבר זה שימושי כאשר יש צורך רק במופע אחד של עצם ממחלקה מסוימת בכל המערכת.

דוגמות:

* ב-Android קיימת המחלקה *Application* המייצגת את היישום עצמו, לכן רצוי וכדאי שיהיה רק מופע אחד.
* ב-ASP .NET ו-ASP MVC, המחלקה Global מהווה סוג של Singleton הרי שהמתכנת משתמש רק במופע אחד של המחלקה. זהו מקרה חריג בו בזמן הריצה נוצרים מספר מופעים שלה, אך הפעולה Application\_Start נקראת עבור רק לאחד מהם, וזהו המופע של המתכנת.
* SqlClientFactory
* SqlProviderServices

ממשים Singleton באופן הבא:

נגדיר בנאי אחד ללא פרמטרים שהוא פרטי למחלקה. לאחר מכן, מגדיר מאפיין (או פעולה) שיחזירו כל פעם את אותו המופע (ניתן לממש ב-Eager או ב-Lazy, ראה Lazy Initialization).

|  |
| --- |
| public class ConfigReader {  private ConfigReader () { }   public ConfigReader Instance { get; } = new ConfigReader(); } |

# [Structural Patterns](https://en.wikipedia.org/wiki/Structural_pattern)

## [Adapter](https://en.wikipedia.org/wiki/Adapter_pattern)

ה-Adapter הוא ממש כמו שהוא נשמע, מתאם, כמו בעולם האמיתי.  
לדוגמה: מתאם בין סוגי שקעים שונים, סוגי חיבורים שונים (DVI to VGA, HDMI to DVI וכו') וכדומה.  
ה-Adapter מאפשר לקוד לעבוד עם הממשק (לא interface בהכרח, אלא הכוונה לשלד החיצוני של המחלקות והפעולות שהוא עובד איתן) שמתאים לו אף על פי שהממשק של המחלקות והפעולות בפועל אינו תואם לצרכיו, בהינתן שהשימושיות שלהם מתאימה לצורך.  
כתוצאה מכך ה-Adapter ממחזר קוד ואין צורך לשנות את קוד המקור של המחלקות שאנו עובדים איתן. דבר זה הוא יתרון משום לא תמיד כדאי או אפשר לשנות את קוד המקור של המחלקות הללו.  
איך עושים זאת? מגדירים מחלקה שמהווה את ה-Adapter שממירה את הממשק של הלא תואם (Adaptee) לממשק אחר (Target) שהקוד (Client) דורש.

דוגמות בספרית ה-.NET:  
\* ראה TextWriter ו-TextReader מהדוגמות ב-Decorator.

* StringWriter – מתאם בין StringBuilder ל-TextWriter. מאחר ו-StringBuilder מטבעו מובנה עם היכולות הבסיסיות של TextWriter, אין יכולות נוספות בהיבט זה, אין אפשרות לבחור קידוד (מפני שבסופו של דבר StringBuilder משתמש ב-char, צורת הייצוג של תווים ב-C#, ולכן הקידוד הוא הקידוד הוא הקידוד של התווים ב-C# (UTF-16)), אך זה כן מאפשר להעביר StringBuilder בתור TextWriter לפעולות שונות.
* StringReader – מתאם בין string ל-TextReader, משמש גם כ-Decorator, ובנוסף גם כ-Adapter מפני שהוא מאפשר להעביר string בתור TextReader לפעולות שונות.

דוגמה:  
נניח ויש לנו מחשב עם חיבור DVI למסך בלבד, ולמסך שלנו יש חיבור VGA בלבד. יש לנו בעיה משום שהחיבור בין המסך למחשב אינם תואמים. הפתרון הוא לשים מתאם באמצע.

נתחיל להגדיר את החיבור (Target) שהמסך ((Client שלנו דורש:

|  |
| --- |
| // Target public interface IVgaSocket {  byte[,][] GetScreenBuffer(); } |

נמשיך בלהגדיר את החיבור שקיים לנו במחשב (Adaptee) שאינו תואם לחיבור שקיים למסך:

|  |
| --- |
| // Adaptee public interface IDviSocket {  byte[,][] GetScreenBuffer(); }  public class DviSocket : IDviSocket {  public byte[,][] GetScreenBuffer()  {  // A screen of 1920x1080 while each pixel is represented by 4 bytes  // to form an ARGB value of color.  // An array of rank 2 of byte arrays that represents a 4-byte pixel  // each.  byte[,][] buffer = new byte[1920, 1080][];   int len1 = buffer.GetLength(0); // 1st rank length: 1920  int len2 = buffer.GetLength(1); // 2nd rank length: 1080  for (int i = 0; i < len1; i++)  {  for (int j = 0; j < len2; j++)  {  // random filling just for the sake of the example  var random = new Random();  buffer[i, j] = new byte[4]  {  (byte)random.Next(0, 255),  (byte)random.Next(0, 255),  (byte)random.Next(0, 255),  (byte)random.Next(0, 255)  };  }  }   return buffer;  } } |

כעת נגדיר את ה-Adapter:

|  |
| --- |
| // Adapter public class DviToVgaAdapter : IVgaSocket {  public IDviSocket DviSocket { get; set; }   public byte[,][] GetScreenBuffer()  {  byte[,][] buffer = this.DviSocket.GetScreenBuffer();   // Code that modifies the buffer so it would be compatible with VGA   // should go here  // ...   // Return the new buffer  return buffer;  } } |

לבסוף, נראה קטע קוד שידגים את השימוש במתאם:

|  |
| --- |
| IVgaSocket dviToVgaSocketAdapter = new DviToVgaAdapter {  DviSocket = new DviSocket() };  // Client code byte[,][] vgaScreenBuffer = dviToVgaSocketAdapter.GetScreenBuffer(); |

## [Composite](https://en.wikipedia.org/wiki/Composite_pattern)

ה-Composite אומר שמתייחסים לקבוצה של עצמים מסוג מסוים באותו אופן שמתייחסים לעצם יחיד. מתייחסים לחלק (כל עצם בקבוצה בפני עצמו) ולשלם (הקבוצה כולה) בצורה אחידה. לרוב מדובר במבנה נתונים של עץ (לא בהכרח בינארי).

אופן המימוש:  
מגדירים ממשק משותף (Component) גם ליחיד (Leaf) וגם לקבוצה (Composite). היחיד מבצע את הפעולה ישירות והקבוצה מעבירה את הפעולות לבנים שלהם (שיכולים להיות קבוצה גם בעצמם).

דוגמה:  
אנחנו רוצים לצייר צורות על המסך שיכולות להיות מורכבות מכמה צורות.  
נתחיל בלהגדיר את הממשק המשותף (Component):

|  |
| --- |
| // Component public interface IGraphic {  void Draw(); } |

כעת נגדיר את הקבוצה (Composite):

|  |
| --- |
| // Composite public class CompositeGraphic : IGraphic {  private List<IGraphic> Graphics { get; }   public CompositeGraphic()  {  this.Graphics = new List<IGraphic>();  }   public void Add(IGraphic graphic)  {  this.Graphics.Add(graphic);  }   public void AddRange(params IGraphic[] graphic)  {  this.Graphics.AddRange(graphic);  }   public void Remove(IGraphic graphic)  {  this.Graphics.Remove(graphic);  }   public void RemoveAt(int index)  {  this.Graphics.RemoveAt(index);  }   public void Draw()  {  foreach (IGraphic childGraphic in this.Graphics)  {  childGraphic.Draw();  }  } } |

נגדיר את היחידים (Leaf):

|  |
| --- |
| // Leaf public class Ellipse : IGraphic {  public Ellipse(Point center, int a, int b)  {  this.Center = center;  this.A = a;  this.B = b;  }   public Point Center { get; }  public int A { get; }  public int B { get; }   public void Draw()  {  Console.WriteLine($"Drawing ellipse at ({this.Center.X}, {this.Center.Y}) with semi-major and semi-minor axes of ({this.A}, {this.B})");  } }  // Leaf public class Rectangle : IGraphic {  public Rectangle(Point upperLeft, Size size)  {  this.UpperLeft = upperLeft;  this.Size = size;  }   public Point UpperLeft { get; }  public Size Size { get; }   public void Draw()  {  Console.WriteLine($"Drawing rectangle at ({this.UpperLeft.X}, {this.UpperLeft.Y}) with width {this.Size.Width} and height {this.Size.Height}");  } } |

נראה את השימוש:

|  |
| --- |
| var composite1 = new CompositeGraphic(); composite1.Add(new Rectangle(new Point(3, 4), new Size(6, 1))); composite1.Add(new Ellipse(new Point(7, 8), 1, 1)); // a circle  var composite2 = new CompositeGraphic(); composite2.AddRange(new IGraphic[] {  new Ellipse(new Point(1, 0), 6, 9),  new Ellipse(new Point(0, -4), 3, 7),  new Rectangle(new Point(5, 2), new Size(7, 4)) });  var composite3 = new CompositeGraphic(); composite3.AddRange(new IGraphic[] {  composite1,  composite2 });  composite3.Draw(); |

## [Decorator](https://en.wikipedia.org/wiki/Decorator_pattern)

ה-Decorator מאפשר הוספה של יכולות למופעים מסוימים בצורה סטטית או דינאמית בזמן ריצה. כמו כן, מטבעם Decorator-ים הם זמניים, ומוקדש להם רק קטעי קוד משום שהם עובדים על מופעים מסוימים, לא משנים את היכולות של המחלקה עצמה, ולא תמיד צריכים את היכולות הנוספות שהם נותנים.

אופן היישום:  
מגדירים מחלקת Decorator שעוטפת את ה-Component על ידי הכלתו בתור שדה (או מאפיין), מאתחלת אותו בבנאים על ידי קבלתו ישירות בבנאי או על ידי יצירת מופע ברירת מחדל, ושמעבירה את כל הפעולות ל-Component עצמו בסופו של דבר. אם כך, ל-Decorator יש אפשרות לבצע פעולות לפני או אחרי שהוא מעביר את הפעולות ל-Component. מכאן שהשינויים נעשים על ה-Component עצמו, והם נשמרים גם לאחר שנפטרים ה-Decorator. כמו כן, רצוי ששדה ה-Component יהיה פומבי ונגיש לכולם למקרה הצורך.

יתר על כן, לפעמים עדיף לרשת מהמחלקה של ה-Component (או לממש את ה-interface שלו). במקרה כזה יהיה ניתן לערום Decorator-ים אחד על השני ולשלב ביניהם כרצוננו. אולם, לא תמיד כדאי לעשות זאת משום שזה לא נדרש או משום שכדאי לשמור את ההורשה למחלקה אחרת. במקרים כאלו לרוב יהיה interface או מחלקה מחלקת בסיס ל-Decorator-ים שעוסקים בדברים דומים. אולם, יש מקרים בהם ה-Decorator יהיה מחלקה בפני עצמה.

\* עטיפה (Wrapper) לעומת הורשה:  
עטיפה זוהי חלופה טובה להורשה להרחבת יכולות ולפעמים אף אפשרות גמישה יותר.  
בהורשה, ההרחבה טבועה במחלקה בצורה סטטית לאחר ההידור ואין אפשרות לשנות זאת בזמן ריצה. יש לציין שבהורשה ניתן גם לגשת ל-member-ים שהם protected ובעטיפה לא.  
כמו כן, לא תמיד אפשר לרשת (מפני שהמחלקה היא נעולה (sealed)).  
יתר על כן, ההבדל נהיה משמעותי יותר כאשר יש מספר דרכים להרחבת היכולות, משום בחלק מהשפות לא ניתן ליצור מחלקות בזמן ריצה, ומשום שלא ניתן לצפות מה יהיה צריך בהמשך.  
עם הורשה יהיה צריך ליצור מחלקה עבור כל צירוף אפשרי (המורכבים משילובים של מספר הרחבות גם כן), לעומת זאת, עצמים עוטפים נוצרים בזמן ריצה ובחלק מהפעמים, אפשר לשלב ביניהם לפי הצורך.

\* ב-javascript ניתן ליצור "מחלקות" בזמן ריצה. באופן תיאורטי גם ב-C#, אבל עדיף שלא כי זה לא פשוט וצורך משאבים רבים והקוד לשימושים בהן הוא מסורבל מאוד.

דוגמות בספרית ה-.NET:

* TextWriter – מהווה מחלקת בסיס עבור StreamWriter ו-StringWriter, מכיל פעולות בסיס לכתיבת טקסט
* TextReader – מהווה מחלקת בסיס עבור StreamReader ו-StringReader, מכיל פעולות בסיס לקריאת טקסט
* StreamWriter – עוטף Stream, מאפשר לכתוב טקסט ל-Stream במקום לכתוב בתים ישירות, מאפשר לבחור קידוד, מאפשר לבחור לנתיב לקובץ בבנאים במקום Stream (במקרה כזה ייווצר FileStream בבנאי)
* StreamReader – עוטף Stream, מאפשר לקרוא טקסט מ-Stream במקום לקרוא בתים ישירות, מאפשר לבחור קידוד ומאפשר לבחור לנתיב לקובץ בבנאים במקום Stream (במקרה כזה ייווצר FileStream בבנאי)
* BinaryWriter – עוטף Stream, מאפשר לכתוב primitives (מספרים, תווים ובוליאנים) ומחרוזות ל-Stream במקום להתעסק עם בתים ישירות ומאפשר לבחור קידוד לתווים ולמחרוזות
* BinaryReader – עוטף Stream, מאפשר לקרוא primitives ומחרוזות במקום להתעסק עם בתים ישירות ומאפשר לבחור קידוד לתווים ולמחרוזות.  
  \* אין אפשרות לבחור את אורך המחרוזת שקוראים מ-Stream.  
  \* אורך המחרוזת נקבע על פי מספר שכתוב לפני המחרוזת. הוא קודם מנסה לקרוא את המספר ואחרי את המחרוזת. כמו כן חשוב לדעת שאורך המחרוזת אינו יכול להיות גדול יותר מ-int.MaxValue, והוא נקבע על פי אלגוריתם הנקרא [7Bit Integer](https://stackoverflow.com/questions/1550560/encoding-an-integer-in-7-bit-format-of-c-sharp-binaryreader-readstring).
* StringReader – עוטף string, מאפשר קריאת תווים, שורות וטקסט מ-string

דוגמות:

* המבנה של תיקיות וקבצים במערכות ההפעלה
* נניח שאנחנו מתכנתים מערכת הפעלה עם חלונות. אנחנו בוחרים לייצג כל חלון עם המחלקה Window שמממשת את ה-interface IWindow. אנחנו רוצים לאפשר גלילה מאוזנת ומאונכת רק לחלק מהחלונות. נכון למצב זה, נוכל לעשות זה על ידי הורשה או על ידי Decorator. עבר קצת זמן, ועכשיו אנחנו רוצים גם להוסיף שוליים מיוחדים לחלק מהחלונות. אם מימשנו את הגלילה בעזרת הורשה, יש לנו בעיה מפני שהמחלקה שעשינו עבור הגלילה (שנקרא לה ScrollableWindow) לכאורה יצרה סוג אחר של חלון. אם כך, נצטרך ליצור מחלקה שיורשת מ-Window ומממשת את השוליים המיוחדים ומחלקה נוספת שיורשת המחלקה שיורשת מ-ScrollableWindow ומממשת את השוליים המיוחדים. הבעיה רק מחמירה עבור כל הרחבה נוספת שנצטרך. עם זאת, עם ה-Decorator נוכל פשוט ליצור מופעים בזמן הריצה, לעטוף את החלונות שאנו רוצים ב-Decorator-ים המתאימים ולעשות שילובים בין ה-Decorator-ים לפי הצורך שלנו: להשתמש ב-Decorator שמוסיף גלילה, ב-Decorator שמוסיף את השוליים המיוחדים או בשניהם.  
  כמו כן, במקרה שלנו, נדרוש שה-Decorator-ים יממשו את IWindow.  
  אולם, חשוב לציין שאם אנו רוצים לעשות את ההרחבות הללו על כל החלונות, לערוך את קוד המקור של המחלקה Window זו אפשרות. אולם יש מקרים בהם זה לא אפשרי או נוח (למשל כי המחלקה מוגדרת בספריה חיצונית).  
    
  אופן היישום:  
  נתחיל להגדיר את המחלקות הבסיסיות:

|  |
| --- |
| public interface IWindow {  void Draw();  string Title { get; set; } }  public class Window : IWindow {  public void Draw()  {  Console.WriteLine($"Drawing window {this.Title}...");  }   public string Title { get; set; } } |

כעת נגדיר את מחלקת הבסיס ל-Decorator-ים:

|  |
| --- |
| public abstract class WindowDecorator : IWindow {  public WindowDecorator(IWindow window)  {  this.Window = window;  }   public IWindow Window { get; }   public string Title  {  get  {  return this.Window.Title;  }  set  {  this.Window.Title = value;  }  }   public virtual void Draw()  {  Console.Write("Decorated: ");  this.Window.Draw();  } } |

נגדיר את ה-Decorator-ים עצמם:

|  |
| --- |
| public class ScrollableWindow : WindowDecorator {  public ScrollableWindow(IWindow window) : base(window) { }   public override void Draw()  {  base.Draw();  this.DrawScrollbars();  }   protected virtual void DrawScrollbars()  {  Console.WriteLine("Drawing scrollbars...");  } }  public class SpecialBordersWindow : WindowDecorator {  public SpecialBordersWindow(IWindow window) : base(window) { }   public Color Color { get; set; }  public string Shape { get; set; }  public int Thickness { get; set; }   public override void Draw()  {  base.Draw();  this.DrawBorders();  }   protected virtual void DrawBorders()  {  Console.WriteLine($"Drawing special borders of color {this.Color}, shape {this.Shape} and thickness {this.Thickness}...");  } } |

אופן השימוש:

|  |
| --- |
| IWindow originalWindow = new Window();  // notice how we're stacking decorators on top of each other WindowDecorator window = new SpecialBordersWindow(new ScrollableWindow(originalWindow)) {  Title = "My Decorated Window",  Color = Color.Green,  Shape = "Star",  Thickness = 2 };  window.Draw(); window.Title = "New Title";  Console.WriteLine(); originalWindow.Draw(); // notice how the new title is still in effect |

## [Facade](https://en.wikipedia.org/wiki/Facade_pattern)

ה-Facade יוצר ממשק פשוט לפעולות מסובכות, הוא מבצע פעולות רבות מאחורי הקלעים. כמו כן, מטבעו הוא יכול לקבץ מספר פעולות מספריות שונות כדי לבצע פעולות שכיחות שמצריכות לרשום קוד רב לפעולות פשוטות לקוד הלקוח. כתוצאה מכך, ניתן להשתמש בספריות חיצוניות בקלות רבה יותר ולהבין אותן יותר לעומק. בעקבות זאת, יש פחות קוד חיצוני שצריך להסתמך עליו, וקוד שמשתמשים בו פעמים רבות מרוכז במקום אחד, ולכן יותר קל לבצע בדיקות, למנוע באגים ולפתור אותם.

דוגמה:  
נניח שיש לנו מחשב. הוא מורכב מחלקים רבים. בסופו של דבר, ביצוע פעולות בו, אפילו הפשוטה ביותר, להדליק אותו, מצריכות פעולות רבות מחלקים שונים כגון: מעבד, זיכרון RAM, דיסק קשיח וכרטיס מסך. כפתור ההדלקה של המחשב הוא בעצם Facade עבור בן האדם שמשתמש בו.  
אנחנו נדמה את פעולות האתחול שלו.

נדמה תחילה את החלקים:

|  |
| --- |
| public class CPU {  public void Freeze()   {  Console.WriteLine("CPU freezing");  }  public void Jump(long position)  {  Console.WriteLine($"Jumping to position {position}");  }  public void Execute()  {  Console.WriteLine("CPU executing...");  } }  public class Memory {  public void Load(long position, byte[] data)  {  Console.WriteLine($"Loading data of length {data.Length} to position {position}");  } }  public class HardDrive {  public byte[] Read(long lba, int size)  {  // Read from hard drive...   return new byte[size];  } } |

נגדיר את המחשב (ה-Facade):

|  |
| --- |
| // Facade public class Computer {  private const int BOOT\_ADDRESS = 0x10;  private const int BOOT\_SECTOR = 0x0;  private const int SECTOR\_SIZE = 0x200;   private CPU Processor { get; }  private Memory Ram { get; }  private HardDrive HardDrive { get; }   public Computer()  {  this.Processor = new CPU();  this.Ram = new Memory();  this.HardDrive = new HardDrive();  }   public void Start()  {  this.Processor.Freeze();  this.Ram.Load(BOOT\_ADDRESS, this.HardDrive.Read(BOOT\_SECTOR, SECTOR\_SIZE));  this.Processor.Jump(BOOT\_ADDRESS);  this.Processor.Execute();  } } |

קוד הלקוח (הקוד שמשתמש ב-Facade):

|  |
| --- |
| var computer = new Computer(); computer.Start(); |

שימו לב כמה פשוט זה נראה מקוד הלקוח לעומת אם היינו עושים את ישירות.

# [Behavioural Patterns](https://en.wikipedia.org/wiki/Behavioral_pattern)

## [Iterator](https://en.wikipedia.org/wiki/Iterator_pattern)

ה-Iterator מאפשר לקוד לקוח לעבור על אוסף פריט אחד אחרי השני מבלי לחשוף את ייצוגו וצורתו הפנימיים (של האוסף, מבנה הנתונים הפנימי). נעשה זאת על ידי הפרדת אלגוריתם המעבר על האוסף מהאוסף עצמו על ידי הגדרת מחלקה שכל תפקידה הוא לספק ממשק חיצוני בכדי לעבור על האוסף המסוים שלשמו הגדירו אותה.  
עדיף שהמחלקה הזו תהיה מחלקה פנימית בתוך המחלקה של האוסף כדי לאפשר למחלקה הזו גישה ל-member-ים הפרטיים של המחלקה של האוסף וכדי לשמור על כמיסות.

תבנית ה-Iterator ב-.NET:  
ה-Iterator הוא דבר מובנה ב-.NET (ב-C# ו-VB.NET) על ידי הלולאה foreach וה-interface-ים IEnumerable<T> ו-IEnumerator<T>. הם מהווים עמודי תווך לשפות ה-.NET, לכל האוספים בספריה, ולפעולות ההרחבה של LINQ המוגדרות במחלקה System.Linq.Enumerable ובכלל לכל קוד שמשתמש באוספים.  
האוסף הוא IEnumerable<T> וה-Iterator הוא IEnumerator<T>.

על ידי מימוש ה-interface IEnumerable<T>, המחלקה מוגדרת כאוסף של פריטים מהסוג T.  
נסתכל על ההגדרה שלו:

|  |
| --- |
| public interface IEnumerable<out T> : IEnumerable {  IEnumerator<T> GetEnumerator(); } |

על הפעולה GetEnumerator() להחזיר מופע חדש של IEnumerator<T> כל פעם שהיא נקראת. על המופע להיות במצב של לפני הפריט הראשון באוסף.

נסתכל על ההגדרה של IEnumerator<T>:

|  |
| --- |
| public interface IEnumerator<out T> : IDisposable, IEnumerator {  T Current { get; } }  public interface IEnumerator {  object Current { get; }  bool MoveNext();  void Reset(); } |

הפעולה MoveNext() מקדמת את ה-Iterator לפריט הבא באוסף. על הפעולה להחזיר שקר כאשר ה-Iterator עבר את הפריט האחרון באוסף.  
המאפיין Current מחזיר את הפריט הנוכחי שה-Iterator נמצא בו.  
הפעולה Reset() מחזיר את ה-Iterator למצבו ההתחלתי, שזה לפני הפריט הראשון באוסף.

כמו כן, IEnumerator<T> יורש מ-IDisposable כדי לתת אפשרות ל-Iterators לנקות כל מיני משאבים שהם השתמשו בהם לצורך המעבר על האוסף.

כעת נראה קוד לשימוש ב-Iterator ב-.NET. זהו בעצם קוד השקול לקוד מהודר של לולאת foreach.

|  |
| --- |
| IEnumerable<T> enumerable = // Getting an enumerable from somewhere. // For eaxample: // IEnumerable<T> enumerable = list; // where 'list' is List<T>. List<T> implements IEnumerable<T> so it’s ok. using (IEnumerator<T> enumerator = enumerable.GetEnumerator()) {  while (enumerator.MoveNext())  {  T item = enumerator.Current;  //  // code to do something should be placed here...  } } |

\* הכוונה בכך שעל הקריאה לפעולה MoveNext להחזיר שקר כאשר ה-Iterator עבר את הפריט האחרון באוסף היא שכאשר הקוד מגיע לתנאי הלולאה בפריט שלפני האחרון, הפעולה תקדם את ה-Enumerator לפריט האחרון. אולם אם היא תחזיר שקר כדי להגיד שאין יותר פריטים לאחר פריט זה, הלולאה תסתיים כי התנאי שלה יהיה שקר ולא נעבור על האיבר האחרון. לעומת זאת אם היא תחזיר אמת ורק בחזרה הבאה שלה תחזיר שקר, הלולאה תעבור גם על הפריט האחרון ותסתיים רק לאחר מכן.

\* הכוונה בכך שעל ה-Enumerator להיות מאותחל לפני הפריט הראשון היא שעל הקריאה הראשונה ל-MoveNext להעביר את ה-Enumerator לפריט הראשון. דבר זה נצרך משום שלפני שבכלל נכנס ללולאה, התנאי שלה יחושב, כלומר הפעולה MoveNext תיקרא בפעם הראשונה, ובזמן ובמקום הזה, ה-Enumerator ישנה את מצבו כדי להימצא בפריט הראשון.  
כמובן שדבר זה אינו סותר את הפסקה הקודמת, אלא רק מוכיח אותה יותר. כלומר, כאשר אין פריטים ב-Enumerator, על הקריאה הראשונה ל-MoveNext להחזיר שקר כדי שהקוד לא יגיע לגוף הלולאה. מצב זה הוא כמו המצב שסיימנו את החזרה של הפריט האחרון, המצב שדיברנו עליו בפסקה הקודמת.

עד עכשיו התעלמנו מהצורך במחלקה אחרת רק בשביל זה ומהצורך במופע חדש עבור כל קריאה ל-GetEnumerator. אז, למה מלכתחילה בעצם? Kappa. למה לא פשוט לשים את האלגוריתם של המעבר על האוסף במחלקה של האוסף עצמו?  
ניקח דוגמה פשוטה: מה יקרה אם נרצה לעבור על האוסף בלולאה מקוננת?  
הלולאה הפנימית תשפיע על החיצונית, ברגע שהיא תסתיים גם החיצונית תסתיים משום שה-[state](https://stackoverflow.com/questions/9967733/objects-with-state-and-behavior-in-oop) שלהם משותף.

כמו כן, אסור לשנות את האוסף בזמן מעבר עליו משום של-Iterator אין שום דרך לדעת מה לעשות עם השינוי. האם עליו להתנהג כאילו השינוי לא קרה או האם עליו להתחשב בשינוי זה. כמו כן, זה מסובך בהרבה להתחשב בשינוי מאשר לא. זה מסתבך אפילו בהרבה יותר כאשר מדברים על מצבים אסינכרוניים. בעקבות זאת, כאשר נעשה שינוי לאוסף בזמן מעבר עליו. על כל פעולה ומאפיין של ה-Iterator לזרוק שגירה מסוג System.InvalidOperationException.

דוגמה:  
אנחנו רוצים לבנות רשימה משורשרת (יש את זה בספריה של .NET, אבל לשם הדוגמה).

הקוד של מחלקות הרשימה והחוליה  
(לשם הדוגמה, לא צריך יותר מאת הפעולה Add):

|  |
| --- |
| public partial class LinkedList<T> : IEnumerable<T> {  private LinkedListNode<T> first;  private LinkedListNode<T> last;   public bool IsEmpty => this.first == null;   public void Add(T data)  {  var link = new LinkedListNode<T>(data, null);  if (this.IsEmpty)  {  this.first = link;  this.last = link;  }  else  {  this.last.Next = link;  this.last = link;  }  }   public IEnumerator<T> GetEnumerator()  {  return new LinkedListIterator(this);  }   IEnumerator IEnumerable.GetEnumerator()  {  // Return a new enumerator  return this.GetEnumerator();  } }  public class LinkedListNode<T> {  public LinkedListNode(T data, LinkedListNode<T> next)  {  this.Data = data;  this.Next = next;  }   public T Data { get; set; }  public LinkedListNode<T> Next { get; set; } } |

כעת נמשיך בלכתוב את הקוד של ה-Iterator שלנו:

|  |
| --- |
| // Iterator partial class LinkedList<T> {  public struct LinkedListIterator : IEnumerator<T>  {  private readonly LinkedList<T> list;   // Link  private LinkedListNode<T> currentLink;   public LinkedListIterator(LinkedList<T> list)  {  if (list == null)  {  throw new ArgumentNullException(nameof(list));  }   // Initialize the state to before the beginning of the list  this.list = list;  this.currentLink = null;  this.Current = default(T);  }   // Data  public T Current { get; private set; }   public bool MoveNext()  {  if (this.currentLink == null)  {  // Before the beginning  this.currentLink = this.list.first;  }  else  {  // We're in the middle  this.currentLink = this.currentLink.Next;  }   // The end of the list  if (this.currentLink == null)  {  return false;  }   // Get the current data  this.Current = this.currentLink.Data;  return true;  }   public void Reset()  {  // Initialize the state to before the beginning of the list  this.currentLink = null;  this.Current = default(T);  }   public void Dispose() { }   object IEnumerator.Current => this.currentLink;  } } |

## [Observer](https://en.wikipedia.org/wiki/Observer_pattern)

ה-Observer design pattern מגדיר שעצם (Subject) מודיע לעצמים אחרים (Observers) על כך ששינה את מצבו. ה-Observer יכול להגדיר מספר הודעות כך שכל הודעה תתעסק עם שינויים מסוגים שונים. ה-Observer design pattern טוב למקרים בהם נדרש קשר של אחד לרבים בין העצם לעצמים רבים אחרים הדורשים לדעת מתי העצם משנה את מצבו. ה-Subject אינו מכיר ב-Observers, והוא לא בהכרח יודע גם אם בכלל יש אפילו Observer אחד (זה לא מעניין אותו). תבנית זו דומה מאוד לתבנית הנקראת [publish-subscribe](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern).

\* ה-Observer design pattern בספר של GoF הוא רעיון מאוד בסיסי. הוא אינו מתעסק עם הסרה של צופים, לוגיקה מסובכת או מותנת שנעשית על ידי ה-Subject לפני או אחרי שליחת ההודעות, הקלטת ההודעות, העברת ההודעות בצורה אסינכרונית, הבטחת קבלתן וכדומה. דברים אלו בדרך כלל מטופלים על ידי מערכות message queueing בהן ה-Observer design pattern הוא חלק קטן.

\* ההבדל בין Observer ל-publish-subscribe: <https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c>

תבנית ה-Observer ב-.NET:  
ה-Observer design pattern זהו דבר מובנה ב-.NET. ניתן לשיים אותו על ידי מילת המפתח event ו-delegate-ים (מצביעים לפעולות). ניתן דוגמה טובה לכך בפרויקט Windows Forms.

דוגמת קוד:

|  |
| --- |
| public class Form1 : Form {  // ...   private TextBox textbox1;   // ...   public Form1()  {  // ...  this.InitializeComponent();  // ...  }   // ...   private void InitializeComponent()  {  // ...  this.textBox1 = new TextBox();  // ...  // register the method to the event as an event handler  this.textBox1.KeyPress += this.textBox1\_KeyPress;  // ...  }   // ...   // This method is invoked when a key is pressed on the 'textBox1' textBox.  private void textBox1\_KeyPress(object sender, KeyPressEventArgs e)  {  // The sender is the text box.  var txtBox = (TextBox)sender;   // The character that was pressed  char pressedKey = e.KeyChar;   // Do some logic here  }   // ... } |

## [State](https://en.wikipedia.org/wiki/State_pattern)

ה-State מאפשר לעצם להתנהג באופנים שונים עבור מצבים שונים שהוא נמצא בהם. הוא מכמס את ההתנהגויות השונות מהעצם עצמו ומאפשר לרשום קוד נקי יותר מאשר קטע גדול של הצהרות תנאים עבור כל מצב ומצב. בעזרת ה-State, ניתן לפתח כל התנהגות המותאמת למצב מסוים באופן עצמאי, וקל להוסיף מצבים והתנהגויות חדשות מבלי שההתנהגויות של המצבים הקיימים ישתנו.

אופן היישום:  
יש להגדיר interface שיגדיר את הממשק להתנהגויות השונות. אחרי כן יש ליצור לכל State מחלקה אשר תממש את ה-interface ותבצע את ההתנהגות המסוימת של ה-State. על העצם להחזיק שדה המצביע ל-State המייצג את המצב הנוכחי שהוא נמצא בו, ולייפות את כל הפעולות של המצבים לשדה ה-State שלו במקום ליישם את ההתנהגויות הללו באופן ישיר בקוד (העצם ראשי לבצע קוד לפני ואחרי הקריאה לפעולות ה-State).

כך, העצם אינו מודע למצב שהוא נמצא בו, וניתן להוסיף מצבים חדשים על ידי הגדרת מחלקות חדשות הממשות את ה-interface של ה-State. לעומת זאת, כל State יכול להיות מודע לעצם ול-State-ים אחרים. בעקבות זאת, כל State צריך לדעת מתי לשנות את ה-State של העצם ל-State אחר שמתאים למצבים חדשים שנוצרים.  
מאחר והעצם עצמו אינו מודע ל-State שלו, הוא אינו יודע מתי לשנות את ה-State שלו ולאיזה State (כמובן שה-State צריך להשתנות מתישהו, ולכן איזשהו קוד אחר צריך לבצע זאת, כלומר הקוד של ה-State-ים עצמם).

יתר על כן, רצוי שה-State-ים יהיו מחלקות פנימיות בתוך המחלקה של העצם כדי שתהיה להם גישה ל-member-ים הפרטיים שלו, וגם רצוי שהם יקבלו בבנאי שלהם את העצם על מנת לשמור אותו אצלם בשדה (או מאפיין) משום שאם מוגדרות מספר פעולות על ה-interface של ה-State, זה מתיש להגדיר עבור כל פעולה פרמטר מסוג העצם ואז גם להעביר את העצם עצמו לפעולה כל פעם (או לחילופין את המאפיינים שצריכים ממנו).  
בנוסף לכך, אם ה-interface של ה-State הוא בעל פעולה אחת בלבד, ניתן לממש בעזרת delegate במקום, אך גישה זו בעייתית מכיוון שאם נרצה להוסיף עוד פעולה, נצטרך לשנות את הקוד כך שישתמש ב-interface במקום ב-delegate או להשתמש במספר delegate (מתיש לשנות את ה-State כשצריך).  
שימו לב גם שגישה זו לא תעבוד בשפות ללא מצביעים לפעולות, כגון java.

דוגמות:

* כספומט: בהתחלה עלינו להכניס כרטיס, אחר כך, עלינו להקיש את הקוד הסודי ולבסוף לבחור את כמות הכסף שאנו רוצים להוציא. בכל שלב, אין אפשרות לבצע פעולה אחרת.
* אנחנו רוצים לשרשר מספר מחרוזות אחת אחרי השנייה כך שבין כל מחרוזת ומחרוזת יהיה תו מפריד. נבחר לעשות את זה על ידי מחלקה אשר תהיה לה פעולה שתוסיף את התו המפריד אם צריך ואת המחרוזת. כמובן שאפשר לעשות זאת בפעולה (מתודה) פשוטה יחסית, אך לשם הדוגמה נעשה זאת במחלקה.

\* כמובן, היישום הפשוט ביותר הוא להחזיק משתנה בוליאני המתאר האם כבר הוספנו מחרוזת או לא. אם עדיין לא, לא צריך להוסיף את התו המפריד, אחרת, צריך להוסיף.  
כלומר, מימוש בעזרת State במקרה זה, זה Overkill ומסומך יתר על המידה.

קוד של היישום הפשוט (עם השדה הבוליאני):

|  |
| --- |
| public class JoinStringBuilder {  public JoinStringBuilder(char seperator) : this(new StringBuilder(), seperator) { }  public JoinStringBuilder(StringBuilder sb, char seperator)  {  this.StringBuilder = sb;  this.Separator = seperator;   this.IsFirstAppend = true;  }   public StringBuilder StringBuilder { get; }  public char Separator { get; }   private bool IsFirstAppend { get; set; }   public void Append(string value)  {  StringBuilder sb = this.StringBuilder;  if (!this.IsFirstAppend)  {  sb.Append(this.Separator);  }  else  {  // We're not supposed to append the separator here because  // it's the first append  this.IsFirstAppend = false;  }   sb.Append(value);   } } |

לעומת זאת, במימוש עם ה-State design pattern, נגדיר interface בעל פעולה אחת בלבד שתבצע את ההוספה של התו המפריד אם צריך ואת המחרוזת.  
קוד המשתמש ב-State:

הגדרת המחלקה הראשית (העצם שלנו):

|  |
| --- |
| public class JoinStringBuilder {  public JoinStringBuilder(char seperator) : this(new StringBuilder(), seperator) { }  public JoinStringBuilder(StringBuilder sb, char seperator)  {  this.StringBuilder = sb;  this.Seperator = seperator;   this.AppendState = new FirstAppendState(this);  }   public StringBuilder StringBuilder { get; }  public char Seperator { get; }   private IAppendState AppendState { get; set; }   public void Append(string value)  {  this.AppendState.Append(value);  } } |

הגדרת ה-State-ים:

|  |
| --- |
| private interface IAppendState {  void Append(string value); }  private class FirstAppendState : IAppendState {  public FirstAppendState(JoinStringBuilder joinStringBuilder)  {  this.JoinStringBuilder = joinStringBuilder;  }   public JoinStringBuilder JoinStringBuilder { get; }   public void Append(string value)  {  this.JoinStringBuilder.StringBuilder.Append(value);  this.JoinStringBuilder.AppendState = new NormalAppendState(this.JoinStringBuilder);  } }  private class NormalAppendState : IAppendState {  public NormalAppendState(JoinStringBuilder joinStringBuilder)  {  this.JoinStringBuilder = joinStringBuilder;  }   public JoinStringBuilder JoinStringBuilder { get; }   public void Append(string value)  {  StringBuilder sb = this.JoinStringBuilder.StringBuilder;  sb.Append(this.JoinStringBuilder.Seperator);  sb.Append(value);  } } |

## [Template Method](https://en.wikipedia.org/wiki/Template_method_pattern)

Template Method היא פעולה שמגדירה רק שלד של אלגוריתם מסוים. היא תהיה בתוך מחלקה (לרוב מופשטת), ולרוב היא אינה virtual כדי שמחלקות יורשות לא ידרסו אותה. הפעולה תייפה חלק המשלבים באלגוריתם למחלקות יורשות על מנת לתת להן לשנות את המימוש של שלבים אלו. החלקים הקבועים ימומשו במחלקת הבסיס והחלקים שמשתנים ימומשו במחלקות יורשות, כך שמחלקות יורשות לא יכולות לשנות את האלגוריתם עצמו, אלא הן רק "משלימות את החסר" עבור האלגוריתם. לא חובה לתת לחלקים המשתנים מימוש בסיסי, אך אפשר (לפי הצורך ולפי כל מקרה בנפרד אם הדבר מתאים). מחלקות יורשות חייבות לממש את החלקים המשתנים אשר אין להם מימוש בסיסי.  
בנוסף, ה-Template Method מעודדת שחזור קוד מאחר והאלגוריתם כתוב רק במקום אחד והשינויים מוגדרים במחלקות יורשות.

\* design pattern זה הוא שכיח (בצורה בסיסית של קריאה לפעולה מופשטת אחת) מאחר ואם מתכנת רוצה לרשום קוד המעודד [polymorphism](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)), ה-Template Method הוא דבר טבעי הנובע מרצון זה משום שהסיבה לכך שפעולה היא מופשטת (או וירטואלית עם מימוש בסיסי) היא כדי שיקראו לה בפעולה אחרת וישנו את המימוש שלה במחלקה יורשת.

\* design pattern זה משתלב באופן מעולה עם Factory Method מאותה הסיבה שההערה הקודמת ציינה.

דוגמות:

* הפעולה OrderPizza(string[], string) שניתנה ב-Factory Method כדוגמה היא Template Method מכיוון שהיא משתמשת במופע של העצם מסוג הפיצה שניתן לה שניתן לה מה-Factory Method. זו בעצם גם דוגמה יפה להערה הראשונה ב-design pattern זה.
* פעולה שהיא Template Method היא ממש מבצעת רשימה של פעולות, פעולה אחר פעולה כמו רשימת קניות. לדוגמה, ניקח הכנת ארוחה:
  1. לקנות את המצרכים
  2. להכין את הארוחה עצמה
  3. לאכול את הארוחה
  4. לנקות
* זו לא באמת דוגמה עם הרבה תכלית, אבל דוגמה טובה בכל זאת.

|  |
| --- |
| public abstract class AlgorithmSkeleton<T> {  public bool Execute()  {  this.Step1();  List<T> list = this.Step2();  if (list == null)  {  return false;  }   // passing the list we got from the 2nd step to the 3rd step  // so that the implementing class can manipulate the list and  // return a new one  IEnumerable<T> enumerable = this.Step3(list);  foreach (T item in enumerable)  {  // check for some arbitrary condition  if (this.IsMatch(item))  {  // if it is a match, do something with it...  this.ProccesItem(item);   // and then return true because we've found what we're  // looking for  return true;  }  }   return false;  }   // Notice how this method is not overridable  // because it's a core part in the algorithm.  private void Step1()  {  Console.WriteLine("Step 1 performed.");  }   protected abstract List<T> Step2();  protected abstract IEnumerable<T> Step3(List<T> list);   protected abstract bool IsMatch(T item);   // A default implementation is given.  // A deriving class can opt to override it completely  // or it can override it and still use it inside somewhere.  protected virtual void ProccesItem(T item)  {  Console.WriteLine(item);  } } |

# Other Patterns

אלו הם Design Patterns שאינם שייכים לקטגוריה מסוימת, והם אינם מופיעים בספר של GoF.

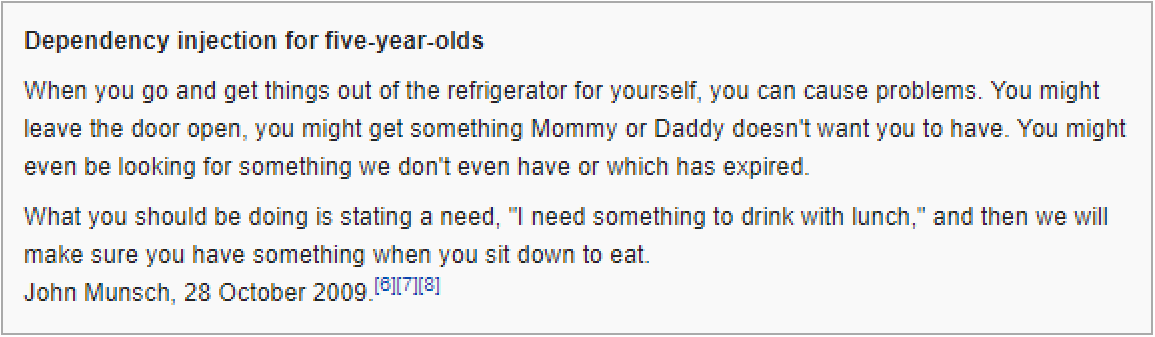
## [Dependency Injection](https://en.wikipedia.org/wiki/Dependency_injection)

Dependency הוא עצם שנותן שירות מסוים (service). Dependency Injection היא הזרקת עצם שירות לעצם אחר (client) בעת יצירתו (כמו בבנאי או ב-setter). האחראיות של סיפוק השירותים מייופת לקוד חיצוני (injector). בנוסף, נותנים ל-client עצם שירות רק בפעם אחת במהלך חייו, כלומר לא משנים את עצם השירות של עצם הלקוח, ועצם השירות ישרת את עצם הלקוח כל חייו (של עצם הלקוח). יש לציין שבדרך כלל, הקוד המזריק הוא שיוצר את עצם הלקוח. דברים אלו מאפשרים להשתמש בעצם השירות גם לאחר חיי עצם הלקוח אם יש בכך צורך, למשל להשתמש בו בתור שירות לעצם אחר.

לכן, Dependency Injection משתלב בצורה טובה עם Factory כלשהו, ובכך ניתן להקל על יצירת השירותים עצמם ולגרום למצב בו לעצם הלקוח אין צורך לדעת איך להרכיב את עצם השירות או את סוגו הממשי. דבר פותר את הבעיה בלתת למחלקת הלקוח ליצור את עצם השירות. במצב בו היא גם יוצרת את עצם השירות, נוספת לה אחראיות, מה שמקשה את תחזוקת הקוד מפני שאם רוצים לשנות את השירות, צריך לשנות את המחלקה של עצם הלקוח (ראה את הסיבות לשימוש ב-Factory).  
בנוסף, אם היצירה של המופע של השירות מתבצעת במחלקה של עצם הלקוח, זה קושר את המחלקה של עצם הלקוח למחלקה המסוימת של עצם השירות, וגורם לכך שלא ניתן לשנות את יצירת המופע של עצם השירות מבלי לשנות את המחלקה של עצם הלקוח.  
יתר על כן, עצם השירות אינו חייב להיות interface או מחלקה מופשטת. יש מקרים בהם חובה להשתמש בסוג הממשי של עצם השירות, אך גישה זו אינה מומלצת. יש להשתמש בה רק כשאין ברירה אחרת, למשל כשעצם השירות אינו מממש interface או מחלקה בסיסית יותר.

\* design pattern זה מופיע בכל מקום מאחר והוא מאוד פשוט וישיר. כמו כן, קשה מאוד לכתוב קוד בלעדיו (ממש צריך להתאמץ).

דוגמות:

* ב-AngularJS כשמגדירים controller, service, factory (וכדומה) חדשים, AngularJS מזריק כל מיני עצמי שירות לפונקציות.
* design pattern-ים לדוגמה שמשתמשים ב-Dependency Injection, הם State, Decorator ו-Adapter. בנוסף, Abstract Factory יכול לשמש בתור עצם שירות לעצם אחר.
* בדוגמה שניתנה ל-State במסמך זה, Dependency Injection מופיע מספר פעמים: פעם אחת בבנאי של JoinStringBuilder ופעמים נוספת בבנאים של כל State.
* כל הארכיטקטורה של ASP.NET Core בנוי על Dependency Injection. לדוגמה ב-Controller-ים, במחלקת ה-Startup וכו'.
* 

## [Lazy Initialization](https://en.wikipedia.org/wiki/Lazy_initialization)

\* זה לא Design Pattern, זו אסטרטגיה, אך שווה ציון. כמו כן, זו הדרך הנפוצה ביותר כדי ליישם את ה-[Lazy Loading](https://en.wikipedia.org/wiki/Lazy_loading) design pattern.

Lazy Initialization היא דחייה של יצירת מופע, חישוב ערך מסוים או אתחול משתנים או משאבים עד לנקודה הראשונה שצריכים את המידע שהם מאתחלים או טוענים. ההפך מ-Lazy Initialization זה Eager Initialization, טעינת המידע בעת טעינת התוכנית, לא דוחים את הטעינה לאחר כך. Lazy Initialization יכול לייעל את התוכנית, בעיקר כאשר מודבר בדבר כבד שצורך טעינה ארוכה יחסית כמו תקשורת בין מחשבים, שליפה ממסד נתונים וכדומה.

Lazy Initialization ב-.NET:

* בספרית ה-.NET קיימת המחלקה Lazy<T> אשר מבצעת Lazy Initialization עבור כל דבר. היא מקבלת delegate שמחזיר עצם מסוג T. כמו כן, היא גם מאפשרת הגנה מפני multithreading וגם שליטה חלקית על איך תעשה זאת.
* בנאים שהם static נקראים (invoked) גם הם ב-Lazy Initialization. הם נקראים רק בפעם הראשונה שמנסים לגשת למחלקה.

Deferred Execution לעומת Lazy Initialization:  
<https://stackoverflow.com/questions/2530755/difference-between-deferred-execution-and-lazy-evaluation-in-c-sharp>  
כשמדובר על LINQ (באוספים, IEnumerable<T>), המונח *Lazy* הוא לא כל כך ברור ודו משמעי. לדוגמה, כנאמר שהאוסף נטען באופן "עצלני", לא ברור אם הכוונה היא לכך שברגע שניגש לפריט באוסף, כולו יטען בבת אחת או שהוא יטען פריט פריט כמו ב-*Deferred Execution*. לכן המונח *Deferred* בהקשר זה מדויק וברור יותר.

יש להשתמש במונח *Deferred Execution* כאשר מדובר ב-LINQ ב-.NET מאחר ו-*Lazy* הוא מונח מעורפל יותר בהקשר זה.

דוגמות:

* כשפותחים משחק (או תכונה כבדה לצורך העניין), המשחק לא טוען את הכל משום שדבר זה ייקח זמן רב וידרוש המון זיכרון RAM. הוא יטען רק את מה שצריך: את התפריט הראשי, יפתח חיבור לשרת המשחק אם צריך, הגדרות המשתמש (כגון גרפיקה ומקשים) ועוד. כשנבקש ממנו לשחק לפי שמירה מסוימת, הוא יטען את קובץ השמירה, המפה וכו' או לחילופין אם זה משחק ברשת, כשהוא ימצא לנו משחק מול אנשים אחרים, הוא יטען את המפה שהולכים לשחק בה, טקסטורות קשורות, גרפיקה וכדומה.
* אנו דוחים את יצירת המופע היחיד של המחלקה DbConn, את יצירת המופע של החיבור עצמו ואת הטעינה של הרשומות ממסד הנותנים עד לרגע שצריכים אותם.

|  |
| --- |
| public class DbConn {  private static DbConn instance;   private DbConnection \_dbConnection;   private DataSet \_customers;   private DbConn() { }   public static DbConn Instance  {  get  {  if (instance == null)  {  instance = new DbConn();  }   return instance;  }  }   public DbConnection DbConnection  {  get  {  if (this.\_dbConnection == null)  {  // initialize db connection here  //this.\_dbConnection = new SqlConnection("connection string");  }   return this.\_dbConnection;  }  }   public DataSet Customers  {  get  {  if (this.\_customers == null)  {  // make sure the database connection is initialized  DbConnection dbConnection = this.DbConnection;   // load customers from databasse  try  {  var command = new SqlCommand("SELECT \* FROM Customers");  var dataAdapter = new SqlDataAdapter(command);   dbConnection.Open();  var dataset = new DataSet();  dataAdapter.Fill(dataset);   this.\_customers = dataset;  }  finally  {  dbConnection.Close();  }  }   return this.\_customers;  }  } } |

## [Method Chaining](https://en.wikipedia.org/wiki/Method_chaining)

Method Chaining הוא קריאה למספר פעולות כך שכל פעולה מחזירה עצם כדי לאפשר לקרוא לכולן באותה ההצרה מבלי לצרוך משתני עזר שיאכסנו את תוצאות הביניים. בעזרת זה ניתן גם לבצע מספר פעולות על אותו עצם בהצהרה אחת על ידי כך שכל פעולה תחזיר את העצם עצמו. דבר זה שימושי מאוד עבור Builder.

דוגמות ב-.NET:

* Method Chaining על אותו עצם מתאפשר חלקית על ידי מאפיינים ו-[Object and Collection Initializers](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers). ניתן לקרוא דוגמה לכך בדוגמה שניתנה עבור Builder במסמך זה. נתנו ערך למספר מאפיינים של עצם ה-Builder בהצהרה אחת מבלי שנצטרך לשמור מצביע אליו. בשפת תכנות אחרת ללא דברים אלו, פעולות ה-setter-ים של ה-Builder היו מחזירים את ה-Builder עצמו:

|  |
| --- |
| Email email = new EmailBuilder()  .SetFrom("Hanzo")  .AddTo(new string[] { "Genji" })  .SetSubject("Shimada")  .SetBody("Simple geometry ")  .SetCc("Tracer ")  .Build(); |

\* כמובן שמאפיינים לבד הם לא יותר מאשר מעטפת יפה לפעולות getter-ים ו-setter כך שקריאה למאפיין של מאפיין (של מאפיין וכו' (בצורה רקורסיבית)) אינה דבר ייחודי ל-.NET.

* כל הפעולות מהמחלקות System.Linq.Enumerable ו-System.Linq.Queryable.  
  כאשר אנו משתמשים בפעולות LINQ, ניתן לשרשר אותן אחת אחרי השנייה באותה ההצרה מבלי לאחסן את הערכים שמוחזרים מהן במשתנים.

דוגמות נוספות:

* ספרית ה-JavaScript jQuery מאפשרת Method Chaining עבור עצמים המייצגים element אחד או יותר. לדוגמה:

|  |
| --- |
| $("Persons").slideDown("slow")  .addClass("grouped")  .css("margin-left", "11px"); |

## [Null Object](https://en.wikipedia.org/wiki/Null_object_pattern)

Null Object הוא עצם שהתנהגותו אינה עושה דבר (התנהגות ניטרלית או חוסר התנהגות, כמו 0 בחיבור או 1 בכפל). הרצון לעצם שכזה הוא שמשתנים המחזיקים מצביעים לעצמים יכולים להיות null ומצריכים בדיקה אם הם null לפני שמשתמשים ב-member-ים שלו (Property, Method, Event, Field). בערת Null Object ניתן להימנע מזריקת תעופה (exception).

דוגמות בספרית ה-.NET:

* DBNull.Value – עמודה שערכה NULL של רשומה ב-DB
* EventArgs.Empty – משמש לאירועים שאין להם נתונים
* Match.Empty – תוצאה מבדיקה של Regular Expression על מחרוזת מסוימת כאשר לא נמצאה התאמה

\*מ-C# 6.0 ומעלה ניתן להשתמש באופרטור *?.* ([null-conditional operator](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-conditional-operators)) שיחזיר null אם האופרנד השמאלי הוא null בעצמו (אלא אם התוצאה היא value type, במקרה כזה מוחזר Nullable<T> כאשר T הוא סוג התוצאה, או בקיצור T?).  
דוגמה:

|  |
| --- |
| string str; int? len = str?.Length; |

כמו כן, ניתן גם לשלב זאת עם extension methods כדי לבצע פעולות על מופעים שיכולים להיות null ועדיין להיראות בקוד כאילו קראו לפעולה הזו על המופע (הרי שהן רק פעולות סטטיות אחרי הכל). הקוד שישתמש בהן לא יהיה חייב לבדוק null, הפעולה תעשה זאת בשבילו.  
דוגמה:

|  |
| --- |
| public static int SafeGetLength(this string str) {  return str?.Length ?? 0; } |

דוגמות:

* יש לנו מחלקה שמייצגת צומת בעץ בינארי ויש לה מאפיין Size שמחשב את גודלו (מספר הצמתים היושבים תחתיו ועצמו):

|  |
| --- |
| public class Node {  public Node(Node left, Node right)  {  this.Left = left;  this.Right = right;  }   public Node Left { get; }  public Node Right { get; }   public int Size  {  get  {  int size = 1;  if (this.Left != null)  {  size += this.Left.Size;  }  if (this.Right != null)  {  size += this.Right.Size;  }   return size;  }  } } |

בחישוב של Size אין ברירה אלא לבדוק האם כל אחד מהצמתים שיושבים תחתיו ישירות הם null. אם נעשה כאן שימוש ב-Null Object, נגדיר את המאפיינים Left, Right ו- Sizeכ-virtual ונוסיף מחלקה פרטית בתוך המחלקה Node שתירש ממנה:

|  |
| --- |
| private class NullNode : Node {  private NullNode () { }   public static NullNode Instance { get; } = new NullNode();   public override Node Left => this;  public override Node Right => this;  public override int Size => 0; } |

נדרש גם להוסיף בנאי protected Node() { } כדי שיהיה ניתן ליצור את הבנאי  
private NullNode() { } משום שאין אפשרות ליצור את הבנאי  
private NullNode() : base(this, this) { }  
מפני שאי אפשר להדר את הקוד הזה.

לאחר מכן, נגדיר שדה public static readonly Node.Empty, שהוא יחזיק לנו מצביע פומבי ל-Node הריק שמשמש כה-Null Object שלנו כדי שתהיה גם גישה אליו מבחוץ אם צריך (לדוגמה כדי לעצור פעולה רקורסיבית או לולאה שעוברות על העץ).

כעת, אנו מגדירים (עושים הסכם בינינו) שבתוכנית שלנו משתמשים ב-Node.Empty במקום ב-null.

בסופו של דבר, המחלקה Node תיראה כך:

|  |
| --- |
| public class Node {  public Node(Node left, Node right)  {  this.Left = left;  this.Right = right;  }   private class NullNode : Node  {  private NullNode() { }   public static NullNode Instance { get; } = new NullNode();   public override Node Left => this;  public override Node Right => this;  public override int Size => 0;  }   static Node()  {  Empty = NullNode.Instance;  }   // This constructor is necessary because otherwise the NullNode node  // class cannot be instantiated  // because trying to define the following constructor NullNode() :  // base(this, this)  // is not possible because it doesn't compile.  protected Node() { }   public static readonly Node Empty;   public virtual Node Left { get; }  public virtual Node Right { get; }   public virtual int Size  {  get  {  return 1 + this.Left.Size + this.Right.Size;  }  } } |

כעת, אין לנו בדיקות null.