Design Patterns - DP

[אתר מומלץ ללימוד Design Patterns](https://refactoring.guru/design-patterns)

design – עיצוב, pattern – תבנית

design pattern – פתרונן תכנותי למשימה תכנותית סטנדרטית, המשימה יכולה להופיע בוריאציות שונות, אבל הפתרון הוא אותו פתרון למשימות מסוגים זהים.

gang of Four – קבוצה של ארבעה אנשים (GOF)

יתרונות לשימוש ב design patterns:

1. הפתרונות ש DP מציעים נבדקו על ידי מתכנתים רבים ונמצאים בשימוש בתוכנות רבות, והוכחו כיעילים.
2. שפה משותפת בין מתכנתים – מספיק להשתמש בשם של התבנית ומתכנת שני ידע על מה מדובר.

design Patterns אינם מתייחסים לשפת תכנות מסוימת, אלא מציגים פתרון תכנותי, שיכול להיות מיושם בהרבה בשפות תכנות, בתנאי שמדובר בשפת תכנות שהיא OOP

DRY – don’t repeat yourself

OOD – Object Oriented Design – עיצוב ותכנון של האובייקטים בכל המערכת שבונים, כאשר יש הקפדה מאד גדולה על מספר עקרונות.

SOLID – ר"ת של חמישה עקרונות לעיצוב נכון:

* [**S** - Single-responsiblity Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#single-responsibility-principle) – לכל מחלקה או פונקציה יש אחריות לבצע פעולה אחת בלבד (ולא פקודה אחת בלבד)
* [**O** - Open-closed Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#open-closed-principle) - פתוח להרחבה, סגור לשינויים. לדוגמא: מבצעים קוד לציור צורות. ניצור מחלקת בסיס עם מחלקות יורשות נפרדות שבכל מחלקה יש מימוש לפונקציה לציור צורה, **במקום** מחלקה אחת שיש בה פונקציות לציור עיגול, ריבוע, משולש וכו'. אם היינו יוצרים מחלקה אחת ובה פונקציות, בכל פעם שנרצה להוסיף צורה למאגר הצורות, נצטרך לשנות את אותה מחלקה ובה כל הצורות.
* [**L** - Liskov Substitution Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#liskov-substitution-principle)
* [**I** - Interface Segregation Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#interface-segregation-principle)
* [**D** - Dependency Inversion Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#dependency-inversion-principle)

UML – שפה של צורות שמתארות מבנה של מחלקות וקשרים ביניהם.

DP מתחלקים ל 3 קבוצות:

1. creational design patterns – תבניות שקשורות לאופן יצירת אובייקטים
2. structural design patterns – תבניות שקשורות למבנה של האובייקטים
3. behavioral design patterns – תבניות שקשורות התנהגות של האובייקטים

**Creational Design Patterns**

Singleton

מתי נרצה להשתמש בו? כשנרצה מופע אחד מסוג המחלקה. זה בדרך כלל מתאים למחלקה שניגשת ל DB, לניהול קבצים א כל ניהול שהוא, למחלקות של Logger וכו'.

1. מגדירים private ctor

private LogManager()

{

}

1. מגדירים static private member מסוג המחלקה עצמה, בתוך המחלקה

static LogManager logManager;

1. מגדירים פונקציה public static שמחזירה מופע מסוג המחלקה, בדרך כלל נקרא לה משהו בסגנון של GetInstance

public static LogManager GetInstance()

{

if (logManager == null)

{

logManager = new LogManager();//here we can create an instance, because we are inside the class, so we can use the private ctor

}

return logManager;

}

1. בתוך הפונקציה בודקים אם כבר יש ערך ל private member, אם לא – יוצרים, ובכל מקרה – מחזירים את המופע. (הפעולה הזו נקראת Lazy Instantiation – אתחול עצל, כי הוא בעצם אומר: רק כשיבקשו ממני מופע אני אצור אותו. למעשה זה יותר יעיל כי אם לא יהיה שימוש במופע – הוא לא יווצר לחינם).
2. כאשר מעוניינים להשתמש במופע, צריכים לגשת באמצעות שם המחלקה נקודה שם הפונקציה הסטטית.

LogManager logger = LogManager.GetInstance();

logger.WriteMessage("Main started....");

UML



Thread-safe Instantiation

מה יקרה אם שניים ינסו לגשת ל GetInstance בדיוק באותו שבריר שניה? שניהם יצרו מופע! והרי אמרנו – זה חייב להיות מופע יחיד?

כדי למנוע בעיה כזו, יש אפשרות "לנעול" את קטע הקוד כך שבוודאות לעולם יתבצע רק ע"י גורם אחד.

ב java מבצעים זאת באמצעות מילת המפתח synchronized, כך:

public static ThreadSafeSingleton getInstance() {

synchronized (ThreadSafeSingleton.class) {

if (instance == null) {

instance = new ThreadSafeSingleton();

}

}

return instance;

}

מכיון ש synchronized קוד הוא בעל ביצועים פחות טובים משמעותית, נרצה למנוע מצב שבו בכל פעם שניגשים ל getInstance מפעילים synchronized code, שנדרש רק למקרה שיוצרים מופע. הדרך לעשות זאת נכון היא פשוט לעטוף אותו ב if זהה ל If הפנימי – שבודק אם המופע עדיין null, ורק אם כן (פעם אחת בלבד) יבוצע הקוד ה synchronized.

ב C# הקוד יהיה דומה, אך עם המילה lock.

נשים לב שבנעילה שולחים ל lock אובייקט כלשהו שמשמש כמנעול ואותו סוגרים/פותחים לפי הצורך.

נדאג שזה לא יהיה אובייקט שעלולים לנעול למטרה אחרת – לכן:

1. לא ננעל את ה this – יכול להיות שהשתמשו בו לנעילה, מחוץ לפונקציה שכרגע מתבצעת (הרי כל this הוא מופע שהוגדר מבחוץ וביצעו עליו איזשהו זימון פונקציה)
2. לא ננעל את ה types עצמם – כי יתכן שישתמשו בהם באמצעות האופרטור typeof במקומות אחרים.
3. Strings – כי מחרוזת יכולה להיות intern (ה CLR למעשה שומר רק מחרוזת אחת מכל רצף תווים – נניח רק פעם אחת abc, רק פעם אחת abcc, גם אם בקוד עצמו יש כמה משתנים שמכילים אותן, ואז המשתנים יצביעו למחרוזת הרצויה)

Factory

מתי מתאים? כאשר נרצה ליצור אובייקט, וסוג האובייקט ייקבע לפי פרמטר מסוים. הפרמטר יכול להגיע מכיוונים שונים – סוג מערכת הפעלה, זמן ביום או קלט ממשתמש מסוים.

איך מממשים בדרך כלל?

1. יוצרים מחלקת בסיס אבסטרקטית שהיא המכנה המשותף של כל האובייקטים שניצור.
2. יוצרים מחלקות נגזרות למחלקת הבסיס – לפי הצורך.
3. יוצרים מחלקת Factory, בדרך כלל נקרא לה בשם של מה שמייצרים (המכנה המשותף), וסיומת המילה Factory.
4. בתוך המחלקה ניצור פונקציה Create, בחתימתה היא תחזיר אוביקט מסוג המכנה המשותף.
5. בגוף הפונקציה, יתבצע מנגנון של בדיקה של פרמטר מסוים – על פיו יוחלט איזה אובייקט ליצור מהמחלקות היורשות.

abstract class MailItem

{

public Address MailAddress { get; set; }

public MailItem(Address address)

{

MailAddress = address;

}

public abstract int Send();

}

class TruckMail : MailItem

{

public TruckMail(Address address):base(address)

{

}

public override int Send()

{

return 2;

}

}

class MailItemFactory

{

public MailItem Create(string addressStr)

{

Address address = new Address(addressStr);

switch (address.Type)

{

case AddressType.InsideCountry:

return new TruckMail(address);

case AddressType.Abroad:

return new AirplaneMail(address);

case AddressType.OffPlanet:

break;

default:

break;

}

return null;

}

}

UML



Abstract Factory

מתי נשתמש?

נרצה לייצר מספר אובייקטים ששייכים לאותה "משפחה".

המימוש:

1. מגדירים עץ מחלקות לאובייקטים – לדוגמא: רהיט, כיסא, שולחן, ספה.
2. מגדירים עץ מחלקות של Factory, בראשו יהיה abstract factory .
3. ב factory עצמו תהיה פונקציה לייצור כל סוג. הערך המוחזר מהפונקציה – הרהיט הבסיסי.
4. כל Factory ספציפי, יירש את ה factory הבסיסי וייצר את האובייקט המתאים לפי מה שהוא מייצר.

ספה עתיקה

ספה מודרנית

ספה

שולחן עתיק

שולחן מודרני

כיסא עתיק

כיסא מודרני

כיסא

שולחן

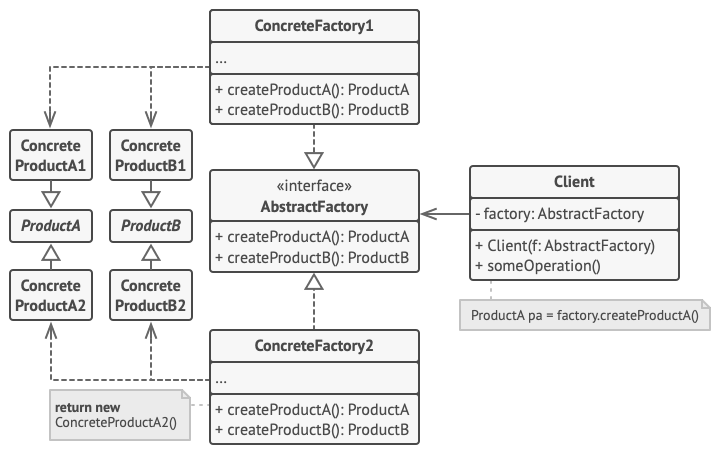
רהיט

FurnitureFactory

VintageFactory

ModernFactory

UML



Builder

המטרה: ייצור מוצר מסוים, בתהליך משתנה.

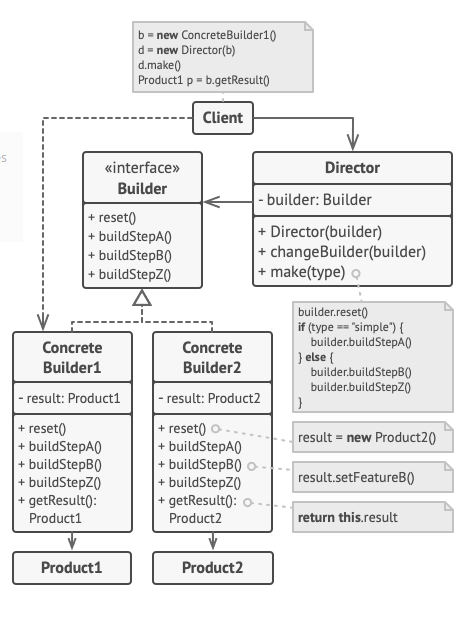
לדוגמא: הצגת דו"ח של תנועות כספיות של חברת אשראי.

המוצר (Product) – דו"ח תנועות

התהליכים המשתנים (Builders) – מחלקות עם פונקציות מתאימות לייצור כל סוג של דו"ח, לדוגמא: הוספת רשומה לדו"ח, הגדרת כותרת, הגדרת שורת סיכום, וכו'

המייצר (Director) – האובייקט שאחראי בסופו של דבר לזמן את הפונקציות המתאימות לצורך המוצר המוגמר. בדוגמא של הדו"ח – זו תהיה המכונה האוטומטית להצגת הדוחו"ת, כשהיא תתבקש להציג דו"ח מסוים, היא תזמן את הפונקציה להוספת רשומות לדו"ח, ואת הפונקציה שמציגה את הדו"ח הסופי.

UML



הערות:

1. צריך לאפשר לבצע ב- director שינוי של ה builder – פעמים רבות מתאים שה director יהיה אובייקט יחיד שיופעל בכל פעם על פי סט שונה של פעולות שנקבע ב builder
2. פעמים רבות נוכל לשלב Factory – שיבחר איזה builder לייצר ולשלוח ל director.

Factory or Builder?

factory ו builder שניהם בסופו של דבר מייצרים אובייקטים שונים.

factory מייצר אובייקט על פי בחירה מסוימת, מחזיר אותו להמשך פעילות.

לעומת זאת, ב builder מייצרים מוצר סופי כלשהו, בתהליך, באמצעות סדרת פעולות שזהה בכל המוצרים.

Prototype

המטרה: העתקת אובייקט מסוים עם כל הנתונים שלו – לתוך אובייקט חדש. (אובייקט חדש ולא reference חדש – שזה בעצם שתי הצבעות לאותו מופע)

מתי נרצה להשתמש?

* כשיצירת אובייקט היא "יקרה" מבחינת משאבים – זמן ריצה, זכרון וכו'
* כשאובייקט נמצא במצב מסוים ורוצים העתק של המצב לפני ביצוע פעולה שמשנה את האובייקט
* כשאובייקט נמצא במצב שנדרש תהליך כדי להגיע אליו, רוצים לחסוך את התהליך
* חסכון בכתיבת קוד ליצירת אובייקט כשיש כבר אובייקט מוכן שניתן להעתיק אותו.

לפני שנתקדם לדרכי פתרון אפשריות ונבחר פתרון מועדף, כדאי לדעת:

לפעולה של שכפול אובייקט קוראים clone. נהוג להבחין בין שני סוגי clone:

1. clone רגיל – העתקה של מאפיינים שהם בעצמם פרימיטיביים (int, string, bool וכו')
2. Deep Clone – שכפול "עמוק" – אם לאובייקט X יש מאפיין Y שהוא בעצמו אובייקט – deep clone ישכפל גם את האובייקט Y.

איך היינו חושבות לבצע את זה:

1. יצירת מופע חדש, העתקת כל מאפייני המופע המועתק לתוך המופע החדש.

החסרון: באופן זה לא נוכל להעתיק את הנתונים ה privateים.

1. יצירת copy constructor – זהו ctor שמקבל אובייקט מסוג המחלקה (נגיד המחלקה Circle – ctor שמקבל אובייקט Circle). ב ctor נוכל לגשת גם למאפיינים ה private, בכך פתרנו את בעיה מס' 1 – אבל יש בעיה חדשה:

אנחנו צריכים לדעת את הסוג הספציפי שאותו אנחנו מעתיקים (Circle, Triangle וכו')– כי נצטרך לזמן את ה ctor שלו בצורה מפורשת.

נצטרך לעשות: Circle copied = new Circle(original)

ברגע שמציינים במפורש את סוג המחלקה – זה יוצר תלות (Dependency) של הקוד שלנו במחלקה הזו. אחד העקרונות של SOLID מנחה (מ בקמץ) להימנע מתלויות כשיש אפשרות אחרת. נשארנו עם בעיה של תלות שאולי אפשר להימנע ממנה.

נקודה נוספת: לעיתים יש לנו ייחוס מסוג interface, ואנו לא יודעים מה סוג המופע הספציפי שהוא מתייחס אליו.

**הפתרון**

ממשק או מחלקה אבסטרקטית שמכילים פונקציה. הפונקציה תבצע את ההעתקה ותחזיר אובייקט העתק. לפונקציה בדרך כלל נקרא Clone כי זוהי פעולתה. בשפת C# (ובדומה לכך, ב - (Java קיים interface ICloneable עם פונקציה אחת שנקראת Clone.

דוגמא לשימוש ב prototype:

מחלקת אבסטרקטית Shape, מספר צורות שיורשות ממנה.

נגדיר במחלקה פונקציה אבסטרקטית Clone. הצורות היורשות יהיו חייבות לממש (אלא אם כן הן בעצמן אבסטרקטיות – איך שלא יהיה, מימוש סופי יהיה חייב לממש)

כך נוכל להעתיק רשימה של Shapes כלשהי מבלי לדעת מה הסוג הספציפי של כל אובייקט – חסכון בתלות, אבל עדיין מתאפשרת העתקה.

List<Shape> copiedShapes = new List<Shape>();

foreach(var shape in shapes)

{

copiedShapes.Add(shape.Clone());

}

UML

