* Realization (מימוש) -
  - הוא אף מוגד כך שיש מחלקה אחת שמממשת את פעולות המחלקה השנייה.

* Generalization (הכללה) Inheritance -
  - הוא אף נקרא גם "is a", מחלקה אחת יורשת את כל התכונות והפעולות של המחלקה השנייה.
  - יש שמחלקה יורשת ממחלקה אחרת, היא נקראת תת-מחלקה, בעוד שהמחלקה שממנה יורשים נקראת על-מחלקה.
  - אף A יכול B להיות, (לא חובה שיהיה לו לפחות מופע אחד).
  - הוא אף מייצג קשר בין מחלקה B לבין מחלקה A, שבו מופע של B יכול להיות מופע של A, אך לא להפך.

* Composition (הרכבה-חזקה יותר) -
  - הוא אף "יש לו" (has a) בין מחלקות, הוא מוגדר כקשר חזק.
  - כלומר, אם מחלקה A מכילה מחלקה B, אז אם מחלקה A נמחקת, גם מחלקה B נמחקת.

* Aggregation (הרכבה-חלשה יותר) -
  - הוא אף מייצג קשר בין מחלקות (אבל פחות חזק מהרכבה).
  - כלומר, אם מחלקה A נמחקת, מחלקה B לא נמחקת. (מופיעה יותר בחלשה).

  - כלומר, אם יש מופע של A בB, אז לא חייב להיות מופע של B בA.
  - הוא אף מייצג קשר בין מחלקה A למחלקה B, שבו מופע של A יכול להכיל מופע של B (אבל לא חייב).

* Association (חיבור) -
  - מחלקות יכולות להשתמש במחלקות אחרות, הוא גם מאפשר קשר הדדי.

  - הוא אף מייצג קשר בין מחלקה A למחלקה B, שבו מופע של A יכול להשתמש במופע של B, אבל לא חייב לכלול אותו כחלק מהמבנה שלו.

* Dependency (תלות) -
  - הוא אף מייצג קשר בין מחלקה A למחלקה B, שבו שינוי במחלקה B עלול להשפיע על מחלקה A.

תלות בין רכיבים עם ממשקים שונים אך
מאפשר לפי זה שהם מתקשרים ביניהם בלי שינוי
יהיה צורך לשנות את הקוד. ולכן ניתן לבצע
- **Adapter** דפוס מתאם - מטרה היא לאפשר לרכיבים
שונים לעבוד יחד. איך? ע"י יצירת ממשק
ביניים שמתרגם בין הממשקים השונים.

* Srp - Single Responsibility Principle - לכל מחלקה
יש אחריות אחת בלבד.
* Ocp - Open Closed Principle - מחלקות צריכות
להיות פתוחות להרחבה וסגורות לשינוי.
* Lsp = Liskov Substitution Principle - מחלקות
יורשות צריכות להיות ניתנות להחלפה
* Isp = Interface Segrigation Principle - לא
לממש ממשקים שלא נחוצים
* Dip = Dependency Inversion Principle - מודולים
ברמה גבוהה (רחב) לא צריכים
להיות תלויים במודולים ברמה נמוכה (ספציפי).
יש להשתמש בהפשטות.

SOLID - עקרונות מנחים לכתיבת קוד מונחה עצמים

קל יותר לשנות מחלקה שכבר קיימת מאשר ליצור מחלקה חדשה לגמרי.

* שיטות וירטואליות - הוא רצף של מצביעים לפונקציות שמוגדרות בתוך המחלקה ומאותחלות בבנאי.
* שיטות וירטואליות טהורות: `virtual ____ = 0;`
* יורשים יכולים לדרוס שיטות וירטואליות.
* אם יש שיטה וירטואלית טהורה אחת המחלקה נקראת מחלקה אבסטרקטית (abstract class).
* אי אפשר ליצור אובייקטים ממחלקה אבסטרקטית. כמו כן כל מחלקה שיורשת ממנה חייבת לממש את כל השיטות הטהורות.
* מחלקה שאינה אבסטרקטית נקראת קונקרטית (concrete class).
* מחלקה אבסטרקטית יכולה להכיל: שדות, בנאים, שיטות וירטואליות, שיטות וירטואליות טהורות.
* ירושה פרטית - כל השדות והשיטות הציבוריות של האב הופכים לפרטיים אצל הבן. לא ניתן לגשת אליהם מבחוץ.
* ירושה מוגנת - כל השדות והשיטות הציבוריות של האב הופכים למוגנים אצל הבן. ניתן לגשת אליהם מבחוץ.
* ירושה ציבורית - כל השדות והשיטות הציבוריות של האב נשארים ציבוריים אצל הבן. ניתן לגשת אליהם מבחוץ.
* ירושה וירטואלית - מונעת יצירת עותקים כפולים של שדות ושיטות של אב משותף.
* ירושה מרובה - מחלקה יכולה לרשת ממספר מחלקות אב.
* ירושה - מחלקה יכולה לרשת ממחלקה אחת או יותר.
* הורשה - מנגנון שמאפשר למחלקה לרשת שדות ושיטות ממחלקה אחרת.

עקרונות SOLID בתכנות מונחה עצמים:

Single Responsibility Principle

Factory class - מחלקה שתפקידה ליצור אובייקטים.
Abstract Factory - מחלקה אחראית ליצור מספר מחלקות Factory.
יתרון: ניתן להחליף מחלקה אחת ולכל היותר מספר קטן.
Factory Method - יוצרת אובייקט מסוג מסוים, אך תתי-המחלקות יכולות לשנות את סוג האובייקט שייווצר.

* תבנית עיצוב Factory -
שם אחראי, נוסף ומוכר. ליצור אובייקט.
מטרה - לצמצם תלויות בין מחלקות, ולאפשר הרחבה בקלות של המערכת בעתיד. מחלקה שאחראית ליצור אובייקטים אחרים נקראת Factory. תתי-המחלקות יכולות לשנות את סוג האובייקט שייווצר, ומספר הפרמטרים הנדרשים.

* תבנית עיצוב Command -
- תבנית Strategy מאפשרת לשנות את האופן שבו מחלקה מבצעת משימה מסוימת.
לדוגמה שבהינתן מספר אלגוריתמים כך שבזמן הריצה ניתן יהיה לבחור באחד מהם.
מחלקה אחת אחראית על ביצוע האלגוריתם הנבחר, ומחלקות אחרות מספקות את האלגוריתמים השונים.

* תבנית עיצוב Observer -
- תבנית Observer מאפשרת לאובייקט אחד (נושא) לעדכן באופן אוטומטי אובייקטים אחרים (צופים) כאשר מצבו משתנה.
- תבנית Proxy - מחלקה שמשמשת כממשק לאובייקט אחר.
- סוגים: Protection proxy - מגן על האובייקט המקורי, ומאפשר גישה רק למשתמשים מורשים.
- סוגים: Virtual proxy - יוצר אובייקטים "יקרים" לפי דרישה.
- סוגים: Remote proxy - מול "עולם" אחר שנמצא במרחב אחר.
מטרה - לשלוט בגישה לאובייקט.

* תבנית עיצוב Iterator -
מאפשרת לעבור על אוסף של אובייקטים ללא חשיפת המבנה הפנימי של האוסף.
מחלקה המממשת את ה-Iterator יכולה לעבור על אוסף מסוים ולא משנה מה.

* תבנית עיצוב Adapter -
מתאים בין ממשקים שונים.

מבוא לתכנות מתקדם
אוספים (collections) וגנריות (Generics)

* **אוספים** - מבנה נתונים המכיל אוסף של אובייקטים מסוג מסוים.
* **אלגוריתמים** - פעולות שונות המוגדרות על אוספים שונים.
* **איטרטורים** - מנגנון המאפשר לעבור על כל איברי האוסף בצורה אחידה, ללא תלות במבנה הנתונים הספציפי.

- ממשק Collection מגדיר פעולות בסיסיות שכל אוסף צריך לתמוך בהן (הוספה, הסרה, בדיקת הכלה, גודל וכו').
- ממשק Iterable מגדיר פעולה אחת: iterator() המחזירה איטרטור לאוסף.
- ממשק Iterator מגדיר פעולות לקבלת האיבר הבא, בדיקה אם יש עוד איברים, והסרת האיבר הנוכחי.
- ממשק List מייצג רשימה מסודרת של איברים, שיכולים להופיע יותר מפעם אחת.
- ממשק Set מייצג קבוצה של איברים ללא חזרות.
- ממשק Queue מייצג תור עדיפויות שבו האיבר הראשון הוא בעל העדיפות הגבוהה ביותר.
- ממשק Map מייצג מיפוי בין מפתחות לערכים, כאשר כל מפתח מופיע לכל היותר פעם אחת.
- מחלקות מימוש: ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue, HashMap, TreeMap.

* **גנריות** - Generics:
  - מאפשרת כתיבת מחלקות ושיטות שעובדות עם טיפוסים שונים, תוך שמירה על בטיחות טיפוסים.
  - מונעת את הצורך בהמרות טיפוסים (casting) ומפחיתה את הסיכוי לשגיאות בזמן ריצה.
  - מאפשרת יצירת מחלקות ושיטות גנריות שיכולות לעבוד עם טיפוסים שונים.

* **Producer or Consumer?**
  - Producer - מוציא ממנו ערכים לשימוש, כמו כל המחלקות שיורשות מ Mammal. כלומר אפשר להוציא מ Mammal או מכל מחלקה שיורשת ממנו.
  - Consumer - מכניס אליו ערכים לשימוש, כמו כל המחלקות שהן אב קדמון של Mammal. כלומר אפשר להכניס ל Mammal או לכל מחלקה שיורשת ממנו.

* **PECS: Producer Extends (PE) - Consumer Super (CS)**
  - Producer Extends (PE) - הוא משתנה גנרי, כל מחלקה/ממשק שיורשים ממנו יכול להיות הטיפוס הגנרי שלו. לדוגמה: List<? extends Mammal> אומר שהרשימה יכולה להכיל כל סוג שיורש מ Mammal, כולל Mammal עצמו.
  - Consumer Super (CS) - הוא משתנה גנרי, כל מחלקה (או ממשק) שהוא יורש ממנו יכול להיות הטיפוס הגנרי שלו. לדוגמה: List<? super Mammal> אומר שהרשימה יכולה להכיל כל סוג שהוא אב קדמון של Mammal, כולל Mammal עצמו.
  - לשניהן חשוב לשים לב שאם יש, כמו למעלה, int אז לא יכול להיות List<Integer> list

Java Streams

* Lambda - ביטוי: (parameters) -> {func body};
  - Predicate - מקבל T ומחזיר boolean: Predicate<Integer> isOdd = n -> n % 2 == 0;
  - Function - מקבל T ומחזיר R: Function<String, Integer> toWords = w -> w.split(" ").length;
  - Consumer - מקבל T ולא מחזיר כלום: Consumer<String> printing = w -> System.out.println(w);
  - Supplier - לא מקבל כלום ומחזיר T: Supplier<Double> rand = Math.random();
  - Runnable - לא מקבל כלום ולא מחזיר כלום: Runnable r = () -> System.out.println("Hello!");
  - BiFunction - מקבל 2 פרמטרים T, R ומחזיר R: BiFunction<Integer, Integer, Integer> sum = (a,b) -> a+b;
* Streams - רצף איברים מסוג מסוים, שלא מחזיק:
  1. מבנה של מידע כלשהו: Collections, Lists of Fib, Arrays, List
  2. פעולות על Streams: filter(), forEach(), sorted(), distinct(), findFirst(), skip()
  3. מחזיר תוצאה סופית: reduce(), collect(), forEach()
  - כדי למיין Comparator ממיין אובייקטים לפי הסדר הטבעי (מספרים סדר, מחרוזות) אלא אם כן האובייקט מממש את הממשק Comparable שמגדיר את הסדר הטבעי של האובייקט (compareTo).
* Reduce Function - מקבל Streams, מקבל 2 ערכים, ומחזיר ערך יחיד שמשלב אותם לפי הפונקציה (BiFunction) Accumulator שסופקה.
  String result = letters.stream().reduce("", String::concat);
* Collectors - מחלקה עם פונקציות שימושיות לעבודה עם Streams, מאפשרת לאסוף תוצאות של Stream.collect() לתוך מבנה נתונים, כמו רשימה או מפה.
  stream.collect(Collectors.toList());

עקרונות לתכנון חבילות:

* Package Cohesion (לכידות)
  - Reuse/Release Equivalence Principle (REP)
  - Common Reuse Principle (CRP)
  - Common Closure Principle (CCP)
* Package Coupling (צימוד)
  - Acyclic Dependencies Principle (ADP)
  - Stable Dependencies Principle (SDP)
  - Stable Abstractions Principle (SAP)

* (REP) Reuse/Release Equivalence Principle - מגדיר כיצד ניתן לעשות שימוש חוזר בחבילות (מחלקות, מודולים וכו') ומבטיח שכל המחלקות בחבילה משתחררות יחד תחת מספר גרסה אחד.
* (CRP) The Common Reuse Principle - מבטיח כי מחלקות בחבילה משתמשות זו בזו. אם משתמשים במחלקה אחת בחבילה, סביר להניח שנשתמש בכל המחלקות בחבילה. כלומר, אם נשנה מחלקה אחת בחבילה, נצטרך לשחרר מחדש את כל החבילה.
* (CCP) The Common Closure Principle - מבטיח כי מחלקות בחבילה משתנות מסיבות דומות. אם נדרש לשנות חבילה כלשהי, סביר להניח שנצטרך לשנות את כל המחלקות בחבילה. מטרתו לצמצם את מספר החבילות שיש לשנות בעת שינוי דרישות.
* (ADP) The Acyclic Dependencies Principle - מונע יצירת מעגלים בתלויות בין חבילות. כלומר, אם חבילה A תלויה בחבילה B, אז חבילה B לא יכולה להיות תלויה בחבילה A.
  - ישנם 2 פתרונות אפשריים ליצירת מעגל: 1. שימוש בעקרון Reuse ⇐ REP, CRP. 2. שימוש בעקרון Maintenance ⇐ CCP.
* (SDP) The Stable Dependencies Principle - מגדיר כיצד חבילות יציבות צריכות להיות מקושרות זו לזו. חבילה יציבה היא חבילה שקשה לשנות אותה.
  - הגדרת יציבות של חבילה: מספר החבילות שתלויות בחבילה. ככל שמספר החבילות שתלויות בחבילה גדול יותר, כך החבילה יציבה יותר.
* (SAP) The Stable Abstraction Principle - מגדיר יחסים בין חבילות יציבות לחבילות מופשטות. כלומר, חבילה יציבה צריכה להיות מופשטת ככל האפשר, כדי שניתן יהיה להרחיב אותה בקלות.

בס"ד

פיתוח תוכנה מונחה עצמים

Software frameworks

* היפוך שליטה Inversion Of Control - עקרון תוכנה שאומר שהזרימה של התוכנה נקבעת ע"י ה Framework. המשמעות, במקום שהקוד שלנו יקרא לקוד של ספריות חיצוניות (כגון Math) אז השימוש ב Frameworks החיצוני יקרא לקוד שלנו.

- כלומר, הקוד לא שולט ברצף התוכנה, אלא נשלט ע"י המסגרת.

* מאפיינים - Annotations - ניתן להוסיף הערות למחלקות, פונקציות וכו' שבקוד. ההערות משפיעות על אופן הרצת הקוד.

- סימנים מיוחדים בקוד המספקים מידע נוסף לקוד, המשמש בעיקר להפעלת המהדר' להפעיל בדיקה מיוחדת בזמן הקימפול: דוגמא:

@Override - הערה שמציינת שהפונקציה דורסת פונקציה מהורישה. המהדר יבדוק שאכן הפונקציה קיימת, ואם לא, יציג הודעת שגיאה.

* תכנות מונחה היבטים - Aspect Oriented Programming - שיטה להפרדת קוד לוגי שחוזר על עצמו (כמו לוגים, טיפול בשגיאות וכו') מהקוד הראשי.

- נקודת חיבור - Join point - שלב בביצוע תוכנה שבו ניתן להוסיף התנהגות.
- Point Cut - פרדיקט שמגדיר באיזה join point נוסיף את ההתנהגות.
- Advice - ההתנהגות הנוספת.

הגדרת Pointcut:

```
@Pointcut("execution(* *.*.set*(..))")
```

- המחלקה Pointcut - ה Advice מקבל פרמטר מסוג JoinPoint שמכיל מידע כללי (למשל מידע על המתודה וגם מה שקיבלה וכו')
- ויש כמה אפשרויות מתי להפעיל את ה Advice למשל @Before @After.

* הזרקת תלויות (Inject) - מאפשר להחליף רכיבים של מחלקה, לשנות את המימוש שמועבר בזמן ההרצה ע"י ההערה @Inject. כך המחלקות יהיו גמישות לפי סוג הרשאה.

- גם יש כמה מימושים לצורך. ויש כמה שיטות להזרקת מימוש:
  - לסמן את המימוש ברירת המחדל בהערה @Default
  - הזרקת Constructor - צריך להזריק את הערך בהשמת הבנאי.
  - הזרקת Setter - הזרקה ע"י setters ומאפשרת שינוי אחרי הבנאי.

* הדגש בלמידת תבניות הוא על יכולת הזיהוי שלהן בקוד, ולא על יכולת הכתיבה שלהן מאפס.
* הבדלים מהותיים בין Strategy ל State: Strategy - מחלקות מתחרות ביניהן (Composition) ומבצע פעולה אחת.
State - מחלקות שמייצגות מצבים שונים של אובייקט אחד.
* Factory method - תבנית עיצוב שמטרתה ליצור אובייקטים.
* Singleton - מבטיח שיש מופע אחד בלבד במערכת. (getInstance)
* Adapter - זה מקרה שאובייקט מסוים מתאים את עצמו.
* Bridge - זה מצב בהיר כי מדובר בשמות ברורים, כל אחד מתייחס לדבר אחר.
* Command - יש לקרוא בהקשר, זה מקרה שאובייקט מחליף פעולה אחרת.
* Decorator, Proxy ⇐ שניהם עוטפים את האובייקט המקורי. ההבדל ביניהם הוא שבדקורטור יש מספר עוטפים, ובפרוקסי יש רק אחד.
* Strategy, State ⇐ שניהם מחליפים התנהגות בזמן ריצה.
* Observer ⇐ מתאים כאשר יש אובייקט אחד שמשנה את מצבו, ואובייקטים אחרים צריכים לדעת על השינוי.
* מתי נשתמש ב Factory?
כאשר אנחנו לא יודעים מראש איזה סוג של אובייקט נרצה ליצור.
* מה ההבדל בין Factory ל Abstract Factory?
Abstract Factory מייצר משפחות של אובייקטים קשורים.
* מה ההבדל בין Singleton ל Static?
Singleton - מחלקה רגילה שיש ממנה מופע יחיד.
Static - שייך למחלקה ולא לאובייקט.

* השתמש ב Proxy כי צריך לו גישה ל Persistence בשביל לבדוק מדיניות (Policy).
* אנחנו משתמשים ב StationToolkit - מעלה הפשטה וזה מה שכל פיצ'רים נוספים יקבלו, לשלב בעתיד.
WeatherComponent משתמש בזה שמשתמש ב WeatherStation.
* WeatherStation מציב מדדי MonitoringScreen
וכל אחד מהם מממש את Observer
* ה Observer יש לכם אחריות מה לעשות כשמתקבל update ו Observable כדי שיוכל לקרוא notifyObservers
* כל ה Observer מקבלים את אותו Observable ושומרים אותו כשדה
BarometricPressureSensor ו TemperatureSensor
* AlarmClock - AlarmListener מפרידים את הקשר
- AlarmClock לא יודע מי מאזין לו, רק שיש לו כזה.
- AlarmListener יודעים מה הפעולה שלהם.
- משתמש ב Proxy כדי לבדוק הרשאות לפני שמפעילים.
* השתמשתי ב Proxy כי, למרות שזה לא הכי יעיל לבדוק הרשאות כל פעם שמפעילים פונקציה, זה הכי נקי ומסודר.
* השתמשתי בפונקציה Bridge כי, למרות שזה לא הכי יעיל לבדוק הרשאות כל פעם שמפעילים פונקציה, זה הכי נקי ומסודר.
* השתמשתי בפונקציה AlarmListener - AlarmClock - הפילו אחריות ל Observer
- כלומר ה MonitoringScreen שהוא ה Observer לא יודע שהוא מאזין, לא יודע מה עושים כשמתקבל update בעתיד.
* לוקח מוסד ממשתמש ומחזיר לקוחה של מחלקה - אחלה פתרון ל Observer.
* השתמשתי ב Template Method כי רוצים שכל הפונקציות יעבדו באותו אופן, חוץ מדבר אחד שכל אחד עושה אחרת.
* השתמשתי ב Abstract Factory כי אלכס אמר שכך יותר נכון לעשות את זה.
כמעט כל מה שאנחנו עושים בתוכנה זה שכבות, חלק מהשכבות נמצאות בשרת וחלק אצל המשתמש.
* אנחנו StationToolkit ממשיכים לעשות כמה שיותר בתוכנה, חלק מהשכבות נמצאות אצל המשתמש.
* ה StationToolkit ממש כ Singleton, כלומר אנחנו לא רוצים שיהיו לנו כמה מופעים שלו (אין טעם, אחד זה) ומשתמשים בו בכל המקומות.
* בנוסף ה StationToolkit יהיה כאמור ה AbstractFactory ומייצר אובייקטים שונים - תלוי מאיזה סוג הוא.

שונות ממבחנים

* extends - ירושה (הרחבה) - כשמחלקה יורשת ממחלקה אחרת, וגם כשממשק יורש ממשק אחר. * מחלקה יכולה לרשת רק מחלקה אחת!

* implements - מימוש ממשקים. כאשר מחלקה מממשת ממשק.

* מחלקה יכולה לממש כמה ממשקים שהיא רוצה. חייבת לממש את כל הפונקציות.

- מחלקה רגילה או אבסטרקטית - יכולה לרשת extends מחלקה אחת בלבד. class A extends B
- ממשק - יכול לממש implements כמה שהוא רוצה (יכול לממש ממשקים) class A implements x,y
- ממשק ממשק - יכול להרחיב - לרשת extends כמה שירצה. interface A extends x,y

* Dependency A ---> B לA יש פרמטר מסוג B - A מכיר את B

* Association A ⟶ B לA יש תכונה מסוג B

* Aggrigation A◇⟶ B לA יש תכונה מסוג מצביע לB. (מערך/רשימה), אבל יכולים להתקיים בנפרד.

* Composition A◆— B בניגוד לאגרגציה B שייך רק לו, במחיקה של A גם B נמחק.

* Inheritance A —▷ B יורש - A מהסוג של B, A יורש מB

* Realization A ---▷ B קשר של ממשק - A מממש ממשק מסוג B

* בקיום של SRP (עקרון אחריות יחידה) מתקבל: שיפור הארגון של הקוד, קל יותר לשנות את המערכת, קל יותר להבין את המערכת.

* בקומפוזיציה דו כיוונית A◆—B A מכיר את B וגם לB יש מצביע לA, לכן בתוך הבנאי של A תהיה השורה: itsB.setItsA(this)

* קדם = או צמוד או מרחיב (חלש יותר), בתר = או צמוד או מצמצם (חזק יותר).
בקדם יורשים יכולים להיות גמישים בדרישות, בבתר יורשים צריכים להחמיר בדרישות.

* תבניות מבניות לפי מרכיביהן: Composite, Decorator, Proxy.

* אבא extends בן
בן super אבא

* תבנית עיצוב Bridge מבוססת על עקרון עיצוב DIP (היפוך תלויות).

* בהיררכיית אירועים מתחילים מהמצבים הפנימיים, ואם אין תשובה יוצאים החוצה למצבים החיצוניים

* חישוב האפשרויות ב Decorator: $2^{\text{מס' התוספות}}$ * מס' המחלקות

* כל מי שיש אותו את כל הכותרת בקובץ include למחלקה שלו, וזה בגלל שינוי ממנו חש כשמוסיף כלשהו למחלקה מסוימת. זה בגלל שאותה מחלקה משתנה וצריכה להתקמפל מחדש.

* במקרה המוכר בשעון סלולרי - יש רבוי מצבי ספרות, מעבר Digit וSet הם מצבי לחיצה (Button), המשך 2 מצבים להפעיל Set וכפתור ספרה.
המופיע: display מציג שעות.

סיכום מלא

* Pillars OOPS:
  - DCP - יותר קטנה יותר טוב בקוד שלך יותר.
  - DRY - לא לחזור על עצמך.
  - DEP - תלויות צריכות להיות מוזרקות ולא נוצרות בתוך המחלקה.
  - DST - תעדיף קומפוזיציה על פני ירושה (כשאפשר יותר)
  - DSI - תעדיף ממשקים קטנים ומדויקים על פני גדולים.
* Pillars לכתיבת טסטים:
  - DCP - תעדיף טסטים שלא תלויים בסביבה, בזמן ובסדר הטסטים.
  - DRY - לא לחזור על עצמך בטסטים, אבל גם לא להפוך אותם לסבוכים מדי.
  - DEP - כמו שהוא קורה במערכת, תן לטסט להיות עצמאי.
* Pillars לכתיבת מחלקות:
  - DSI - תעדיף ממשקים קטנים ומדויקים על פני גדולים.
  - DST - תעדיף קומפוזיציה על פני ירושה.
  - DEP - תלויות צריכות להיות מוזרקות ולא נוצרות בתוך המחלקה.
  - DCP - תשאיר קוד שנוגע לדברים שונים בנפרד.
* SpringFramework - שימוש ב Pointcut
  - Execution - הכי נפוץ, מאפשר להגדיר בדיוק איזה מתודות יתפסו.
  - Within - כל המתודות בתוך מחלקה/חבילה מסוימת.
  - @annotation - כל המתודות שמסומנות עם אנוטציה מסוימת.
  - args - תפיסת פרמטרים, למשל כל המתודות שמקבלות String.
  - this - כל המתודות של אובייקט מסוג מסוים.
  - target - כל המתודות של אובייקט שמממש ממשק מסוים.
  - bean - כל המתודות של Bean מסוים.
  - @within - כל המתודות במחלקה עם אנוטציה מסוימת.
  - @target - כל המתודות במחלקה שמממשת ממשק עם אנוטציה מסוימת.
  - @args - כל המתודות שמקבלות פרמטר עם אנוטציה מסוימת.

```
@Pointcut("execution(* * . * . set*(..))")
```
כל פונקציות שמתחילות ב set

* Spring - שימוש ב singleton
- סקופ ברירת מחדל: אחד בכל Container
- סקופ אחר: prototype, request, session, application, websocket
- הזרקת תלות: @Autowired / @Inject, @Qualifier("high") / @Named, @Primary / @Produces
- סוגי הזרקה:
  - Field - הכי נפוץ, אבל לא מומלץ כי קשה לבדוק ולא ברור מה התלויות.
  - Constructor - מומלץ! ברור מה התלויות, אפשר final, קל לבדוק.
  - Setter - לתלויות אופציונליות שאפשר לשנות.
* Spring Boot:
  - @SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan
  - application.properties / application.yml - הגדרות
  - Profiles - סביבות שונות (dev, test, prod)
  - Actuator - /health, /info, /metrics, /env
* Spring Data JPA:
  - Repository - ממשק בלבד, Spring יוצר את המימוש
  - CrudRepository - save, findById, findAll, delete, count
  - JpaRepository - מוסיף flush, saveAndFlush, deleteInBatch
  - Derived Queries - findByNameAndAge(String name, int age)
  - @Query - כתיבת JPQL או SQL ידנית
  - Pagination - Pageable, Page, Slice
* Spring Security:
  - SecurityFilterChain - הגדרת כללי אבטחה
  - UserDetailsService - טעינת משתמשים
  - PasswordEncoder - BCrypt מומלץ
  - @PreAuthorize / @PostAuthorize - הרשאות ברמת המתודה
* Spring Testing:
  - @SpringBootTest - טעינת כל ה context
  - @WebMvcTest - בדיקת Controller בלבד
  - @DataJpaTest - בדיקת Repository בלבד
  - MockMvc - שליחת בקשות HTTP מדומות
  - @MockBean - החלפת Bean אמיתי ב Mock
* Spring Boot Actuator:
  - /health - בדיקת תקינות
  - /info - מידע על האפליקציה
  - /metrics - מדדים
  - /env - משתני סביבה
  - /beans - כל ה Beans
  - /mappings - כל ה endpoints
* Spring Boot Starters:
  - spring-boot-starter-web - Tomcat + Spring MVC + Jackson
  - spring-boot-starter-data-jpa - Hibernate + Spring Data JPA
  - spring-boot-starter-security - Spring Security
  - spring-boot-starter-test - JUnit + Mockito + MockMvc
  - spring-boot-starter-actuator - Actuator endpoints
* Spring Boot Configuration:
  - @Value("${property.name}") - קריאת ערך מ properties
  - @ConfigurationProperties(prefix="app") - מיפוי אובייקט שלם
  - @Profile("dev") - Bean שנטען רק בסביבה מסוימת
  - @Conditional - Bean שנטען רק אם תנאי מתקיים
* Spring Boot Exception Handling:
  - @ControllerAdvice - טיפול בחריגות גלובלי
  - @ExceptionHandler(SomeException.class) - טיפול בחריגה ספציפית
  - ResponseEntity - החזרת סטטוס HTTP + גוף התשובה
* Spring Boot Validation:
  - @Valid - הפעלת ולידציה על אובייקט
  - @NotNull, @NotBlank, @Size, @Min, @Max, @Email, @Pattern
  - BindingResult - קבלת שגיאות ולידציה
* Spring Boot Scheduling:
  - @EnableScheduling - הפעלת תזמון
  - @Scheduled(fixedRate = 5000) - כל 5 שניות
  - @Scheduled(cron = "0 0 * * * *") - כל שעה
* Spring Boot Caching:
  - @EnableCaching - הפעלת קאש
  - @Cacheable - שמירת תוצאה בקאש
  - @CacheEvict - מחיקה מהקאש
  - @CachePut - עדכון הקאש
* Spring Boot Logging:
  - SLF4J + Logback - ברירת מחדל
  - logging.level.root=INFO - רמת לוג כללית
  - logging.level.com.example=DEBUG - רמת לוג לחבילה
  - logging.file.name=app.log - שמירה לקובץ
* Spring Boot Deployment:
  - java -jar app.jar - הרצה
  - Docker - יצירת image עם Dockerfile
  - Environment Variables - הגדרות בזמן ריצה
* Spring Boot Best Practices:
  - Constructor Injection - תמיד
  - Repository - ממשק בלבד
  - Service - לוגיקה עסקית
  - Controller - רק ניתוב בקשות
  - DTO - לא להחזיר Entity ישירות
  - Validation - תמיד לוודא קלט
  - Exception Handling - לא לתפוס Exception כללי
  - Logging - לא להשתמש ב System.out
  - Testing - לכתוב טסטים לכל שכבה
  - Profiles - להפריד בין סביבות
  - Security - לא לשמור סיסמאות בטקסט
  - Actuator - להגן על endpoints רגישים
  - Caching - להשתמש בזהירות
  - Scheduling - לא לחסום את ה thread
  - Configuration - להשתמש ב @ConfigurationProperties
  - Dependencies - להשתמש ב starters
  - Versioning - לנהל גרסאות API
  - Documentation - להשתמש ב Swagger/OpenAPI
  - Monitoring - להשתמש ב Actuator + Metrics
  - Error Handling - להחזיר שגיאות ברורות
  - Performance - לבדוק ביצועים
  - Security - לעדכן תלויות
  - Code Quality - להשתמש ב Linters
  - Documentation - לתעד את הקוד
  - Testing - לכתוב טסטים לכל שכבה
  - Profiles - להפריד בין סביבות
  - Security - לא לשמור סיסמאות בטקסט
  - Actuator - להגן על endpoints רגישים
  - Caching - להשתמש בזהירות
  - Scheduling - לא לחסום את ה thread
  - Configuration - להשתמש ב @ConfigurationProperties
  - Dependencies - להשתמש ב starters
  - Versioning - לנהל גרסאות API
  - Documentation - להשתמש ב Swagger/OpenAPI
  - Monitoring - להשתמש ב Actuator + Metrics
  - Error Handling - להחזיר שגיאות ברורות
  - Performance - לבדוק ביצועים
  - Security - לעדכן תלויות
  - Code Quality - להשתמש ב Linters
  - Documentation - לתעד את הקוד

גיבוב מיותר עצמים - חשוב לזכור מודלים מבחנים

* במקרה הבוחן נציג תאור השימושים העיקריים עיצוב שונים:

- השתמשנו ב Observer כדי לשחרר את התלות בין שכבת האפליקציה ל GUI
- השתמשנו ב AlarmClock כדי לדלג בין השעון לתהליכים.(לא רשמי, דומה לObserver).
- השתמשנו ב bridge כדי לשחרר את התלות בין שכבת המימוש(החומרה) לשכבת האפליקציה.

* ממשק יכול להרחיב ממשק אבל לא לממש.

ממשק לא יכול לממש מחלקה, ממשק לא יכול להרחיב מחלקה

מחלקה לא יכולה לממש מחלקה

כשמרחיבים את הממשק את קודם כותבים extends ואז implements

* תבנית עיצוב Proxy נראית כך:

יש ממשק כדי שמי שמתכנת את הפונקציה לא ידע מה הוא מתכנת (את ה proxy או המקורי).

* ההגדרה של עקרון REP - חבילת תוכנה היא היחידה הקטנה ביותר שניתן ניהול גרסה.

* כשמשתמשים ב client שתלוי במחלקה ומשתמש בממשקים את מכיל ב עקרונות: DIP ממשק ולא מחלקה, ISP [illegible].

- העיקרון כלל: כמו עצמי להיות תלויים בהכי כמה שיותר מופשטים. (בממשק ולא במחלקה).

* תנאי compareTo - מחזירה מס' שלילי אם y קטן מ x, ומס' חיובי אם y גדול מ x   y.compareTo(x)

* תנאי קדם - כמה שיותר קלטים מה שמוגדרת - הוא יכול להיות יותר "חלש"

תנאי בתר - כדי לשמור על ההגדרה של הבסיס - הוא צריך להיות יותר "קשה".

* הוספת פעולה ופונקציונליות זה תבנית עיצוב Decorator.

* תבנית עיצוב MultipleDecorator

תבנית עיצוב Composite

* Decorator - משכלל פונק' קיימת (ופונקציונליות), *Visitor - מוסיף פונק' חדשה (חיבור לממשק מחדש).

* [illegible]: event, בתנאי, 13 פעולות

* החלק הנקרא קונטקסט © יתבצע תמיד תחת התנאי והפעולה. והתוצאה הידועה מהקונטקסט ולא תלויים שאם יתבצעו כסדר בתוך לפני (לא ייקרם ויתר) ומלא הבדיקה הנבאים 2 מס' הרצות

* נשתמש ב (AOP) Aspect Oriented Programming כאשר יש צורך להוסיף התנהגות לפונ' קיימות מבלי לשנות את הקוד שלהן. מתאים ללוגיקה שחוצה את עצמה.

* השתמש ב Dependency Injections כדי להפריד תלויות ולהכניס רכיבים מבחוץ, אבל נדרשת גישה לקוד המחלקה. מתאים למקרה שבו נרצה להחליף רכיבים בקלות או להקל על בדיקות.

* התבנית Visitor עוזרת להוסיף מספר פעולות למבנה היררכי של עצמים.

* התבנית Command מייצגת פעולות כאובייקטים, מתאימה לביטול פעולות ורישום פעולות.

* התבנית Adapter משנה ממשק קיים כך שיתאים לממשק אחר.

* Adapter - מגשר, מתווך בין שני ממשקים שונים.

* Bridge - מבנה, מפריד בין ממשק למימוש.

* Composite - מאפשר יצירת מבנה עץ של אובייקטים.

* Decorator - מוסיף התנהגות נוספת לאובייקט בזמן ריצה.

* Proxy - מחליף אובייקט אחר ומשמש כמתווך ביניהם.

* Singleton - מבטיח שקיים רק מופע אחד של מחלקה.

```
public static singelton setInstance() {
    if (instance == null) {
        instance = new singleton(); }
    return instance; }
```

* Observer - מאפשר לאובייקטים להירשם לאובייקט אחר (Observable) ולקבל עדכונים כאשר מצבו משתנה.

```
private List<observer> observer = new ArrayList<>();
private void notifyObservers()
    for (observer o : Observer)
        o.update(value);
```

* Visitor - מאפשר להוסיף פעולות על אובייקטים מבלי לשנות את המחלקות שלהם.

```
void accept(visitor v) {
    v.visit(this); }
```

* State - מאפשר לאובייקט לשנות את התנהגותו כאשר מצבו הפנימי משתנה.

* Command - מאפשר לשמור פעולות כאובייקטים, ולספק לקוח לבקש מהמקבל לבצע פעולה מסוימת.

* Abstract Factory - מאפשר ליצור משפחות של אובייקטים קשורים, מבלי לציין את המחלקות הקונקרטיות שלהם.

* Factory Class - מחלקה שיוצרת אובייקטים (מחלקות) שונים, בהתאם לפרמטרים שהתקבלו.

* Factory Method (יצירה) - מאפשר ליצור אובייקט מבלי לציין את המחלקה הספציפית שלו.

```
public static String getMaxAvgSal(List<Employee> employees){
    Map<String, Double> avgSalByDep = employees.stream()
        .collect(groupingBy(Employee::getDepartment,
            averagingDouble(Employee::getSalary)));

    double maxAvgSal = avgSalByDep.values().stream()
        .mapToDouble(Double::doubleValue)
        .max().getAsDouble();

    return avgSalByDep.entrySet().stream()
        .filter(entry -> entry.getValue().equals(maxAvgSal))
        .map(Map.Entry::getKey).collect(joining(","));
}
```

```
@Aspect
public class Logging{
    @Pointcut("execution(* *.*(..))")
    private void selectAll(){}
    @Before("selectAll()")
    public void beforeAdvice(){
        System.out.println("Executing Function");
    }
}
```

* תבנית עיצוב Visitor

- Visit - מתודה שמקבלת אובייקט מסוג Visitor ומפעילה עליו פעולה.
- Accept - מתודה שמקבלת Visitor ומפעילה עליו את Visit.

מטרה: להוסיף פעולות חדשות לעץ אובייקטים מבלי לשנות את מחלקות האובייקטים עצמן. כאשר יש עץ אובייקטים מורכב, ורוצים להוסיף פעולות חדשות לכל האובייקטים בעץ, אפשר לעשות זאת בקלות ע"י הוספת מחלקה חדשה שמממשת Visitor, במקום לשנות את כל מחלקות האובייקטים בעץ.

יתרונות: קל להוסיף פעולות חדשות לעץ אובייקטים קיים. קל לשנות פעולות קיימות בעץ אובייקטים. ניתן לאגד פעולות קשורות במחלקה אחת.

חסרונות: קשה להוסיף אובייקטים חדשים לעץ אובייקטים קיים. יכול לגרום לשבירת כימוס של האובייקטים בעץ.

דוגמה:

```
interface visitor {
        void visit(Book b);
}

class printVisitor implements Visitor {
    public void visit(Book b) {
        System.out.println("Book: " + b.title);
    }
}
```

```
class Book {
    string title = "Design Patterns";
    void accept(Visitor v) {
        v.visit(this);
    }
}
```


* תבנית עיצוב Decorator

- בדומה ל Composite גם ב Decorator כל האובייקטים הם מאותו הטיפוס.
מטרה: מטרה לשנות פונקציונליות של אובייקטים בזמן ריצה. מעטפת לאובייקט שמוסיפה התנהגות נוספת לפני או אחרי קריאה לאובייקט המקורי, מבלי לשנות את האובייקט המקורי עצמו.

```
interface coffee {
    String getDescription(); }
class SimpleCoffee implements Coffee {
    Public String getDescription() {
        return "Coffee"; }
}
Class MilkDecorator implements Coffee {
    Private Coffee base;
    Public MilkDecorator(Coffee base) {
        this.base = base; }
    public String getDescription() {
        return base.getDescription() + "Milk"; }
}
```

* תבנית קוד State

מטרה - מימוש המאפשר לעצם לשנות התנהגות על ידי שינוי מצבו הפנימי של העצם.
כלומר - מחלקה שהתנהגותה תלויה במצב שבו היא נמצאת/נכנסת.
דוגמה: מכשיר אור/מנורה:

```
interface State{
    void handle();
}
class OnState implements State{
    public void handle(){System.out.println("Device is ON");}
}
class Device{
    private State state;
    public void setState(State s){
        state = s;}
    public void pressButton(){
        state.handle();}
}
```


* תבנית קוד Observer

- Observer (אחד) - מקבל הודעה (צורך הודעה)
  אובייקט (observers) המאזין לשינוי מצב.
- Observable (אחד) - מחזיק מצב שעלול להשתנות, מנהל רשימה של Observers (מודיע להם כאשר מצבו משתנה).

```
interface Observer {
    void update(String msg); }
class CaucreteObserver implements Observer{
    public void update(String msg){
        System.out.println("got update: "+msg);}
}
class Subject{
    private List<Observers> observers = new ArrayList<>();
    public void addObserver(Observer o){observers.add(o);}
    public void notifyAll(String msg){
        for(Observer o : observers) o.update(msg);
    }
}
```

* **Factory** נתונים
נתונים יצירה - נתונים שיוך (Creational)
יצירת אובייקט מתוך מחלקה.

יצירת אובייקט מתוך מחלקה אחרת, כאשר יש לנו לוגיקה מורכבת של בחירת סוג האובייקט שיווצר. לדוגמה: יצירת אובייקט מסוג Shape (מחלקת אב) בהתאם לקלט שהתקבל (Shape, Circle, Rectangle)
מטרה - יצירת אובייקטים מבלי לחשוף את הלוגיקה של יצירתם למחלקה שמשתמשת בהם.
יתרון - יצירת אובייקטים מבלי לדעת את הפרטים הספציפיים של יצירתם. יצירת קישור חלש בין האובייקטים.
חסרון - יצירת אובייקטים עלולה להיות מורכבת יותר, במיוחד כאשר יש הרבה סוגים שונים של אובייקטים.

* **Factory Class** - מחלקה שיש לה מתודה אחת או יותר שיוצרת אובייקטים.
* **Factory Method** - מתודה שיש לה פרמטר אחד לפחות שקובע איזה אובייקט ייווצר.
* **Abstract Factory** - מחלקה אבסטרקטית שמגדירה מתודות יצירה לאובייקטים, אך לא ממשת אותן. ממשת אותן מחלקות יורשות שיחזירו אובייקטים ספציפיים.

* **Factory Class** - מחלקה שיש לה מתודה אחת או יותר שיוצרת אובייקטים שונים בהתאם לקלט שהתקבל. לדוגמה: מחלקה שיש לה מתודה שמקבלת מחרוזת ומחזירה אובייקט מסוג Shape (מחלקת אב) בהתאם לקלט שהתקבל (Circle, Rectangle)

* **Factory Method** - מחלקה שיש לה מתודה אחת או יותר שיוצרת אובייקטים שונים בהתאם לקלט שהתקבל. לדוגמה: מחלקה שיש לה מתודה שמקבלת מחרוזת ומחזירה אובייקט מסוג Shape (מחלקת אב) בהתאם לקלט שהתקבל (Circle, Rectangle)

* **Abstract Factory** - מחלקה אבסטרקטית שמגדירה מתודות יצירה לאובייקטים, אך לא ממשת אותן. ממשת אותן מחלקות יורשות שיחזירו אובייקטים ספציפיים של Factory Class מסוים. לדוגמה: מחלקה אבסטרקטית שמגדירה מתודות יצירה לאובייקטים מסוג Shape, אך לא ממשת אותן. ממשת אותן מחלקות יורשות שיחזירו אובייקטים ספציפיים של Factory Class מסוים.

Factory Class דוגמה

```
public class ShapeFactory{
    public Shape getShape(string shapeType){
        if(shapeType == null){
            return null; }
        if(shapeType.equalsIgnoreCase("CIRCLE"))
            return new Circle();
        else if(shapeType.equalsIgnoreCase("RECTANGLE"))
            return new Rectangle();
        return null
    }
}
```

* תבנית עיצוב Composite

- Composite מנסה לאחד, כלומר להתייחס אחיד בין קבצים ותיקיות.
אפשר - סוג ההורה יכיל רשימה של ילדים, לכן פעולות יכול לעבוד אחיד גם
בתבנית עיצוב אחרת בשלה בכלי, ובכלל פעולות יכול לעבוד אחיד גם
בתבנית עיצוב מבנית. ובאופן כללי הוא מאוד שימושי.


```
interface FileSystemItem {  // רכיב
    void display();
}

class File implements FileSystemItem {  // עלה (קובץ - אין לו ילדים)
    private string name;
    public void display() {
        system.out.println("File:" + name);
    }
}

class Folder implements FilesystemItem {  // מיכל (תיקיה - יכול להכיל ילדים)
    private string name;
    private List<FileSystemItem> item = new ArrayList<>();
    public void display() {
        syston.out.println("Folder:" + name);
        for (FileSystemItem item: items) {
            item.display(); }
    }
}
```

* תבנית עיצוב Bridge

- מטרתה להפריד בין מימוש (implementation) לאבסטרקציה (abstraction) כך שניתן יהיה לשנות אותם בנפרד.


```
interface DrawAPI {  // רכיב
    void draw(string shape); }

class RedDraw implements DrawAPI {  // עלה
    Public void draw(String shape) {
        system.out.println("Drawing" + shape + "in red");
    }
}
class shape {  // מיכל
    protected DrawAPI drawAPI;
    public void draw(string shapeName) {
        drawAPI.draw(shapeName);
}
```

* לגבי Lazy initialization - אתחול עצלני - זה כאשר אנו לא יוצרים את האובייקט מיד בתחילת הריצה, אלא רק כאשר הוא נדרש לראשונה.
* כאשר יש כמה threads שניגשים בו זמנית, יכול להיווצר מצב שבו שני threads יוצרים אובייקט. כדי למנוע זאת נשתמש ב synchronized.
* שימו לב: אין לשכוח לאתחל את המשתנה הסטטי לערך ברירת המחדל שלו.

```
public class ConfigurationManager {
    private static ConfigurationManager instance; // משתנה סטטי שמחזיק את האובייקט היחיד
    private ConfigurationManager() {
        configData = "Default"; // אתחול ברירת מחדל
    }
    public static ConfigurationManager getInstance() {
        if (instance == null) {
            instance = new ConfigurationManager(); // יצירת האובייקט היחיד בפעם הראשונה
        }
        return instance;
    }
}
```

* הערה: ניתן להפוך את ה constructor ל protected כדי לאפשר ירושה.

דוגמה: נניח שיש לנו מחלקה בשם ConfigurationManager. אנו רוצים שיהיה רק מופע אחד שלה בכל התוכנית.
נגדיר:
a. משתנה סטטי פרטי (private static) שיחזיק את המופע היחיד.
b. בנאי פרטי (private) כדי למנוע יצירת מופעים מבחוץ.
c. מתודה ציבורית סטטית (public static) שתחזיר את המופע היחיד.
d. בתוך המתודה, אם המופע עדיין לא נוצר, ניצור אותו ונשמור אותו במשתנה.
e. בפעמים הבאות, נחזיר את המופע שכבר נוצר.

* לגבי Singleton:
- היתרון הוא שאנו מבטיחים שיהיה רק מופע אחד של המחלקה בכל התוכנית.
- החיסרון הוא שיש לו השפעה שלילית על ה testability, כי קשה לבדוק מחלקות שתלויות ב Singleton.

* לגבי Singleton

יצירתי, יצור (creational)

* דוגמת קוד

- דוגמת Adapter

מטרת הדוגמה: מתאם בין ממשקים שונים. לדוגמה: נניח שיש לנו מערכת שמצפה לעבוד עם שקע אירופאי, אך אנחנו רוצים לחבר אליה תקע אמריקאי. כדי לעשות זאת נצטרך מתאם שיאפשר לנו לחבר את התקע האמריקאי לשקע האירופאי.

מרכיבי הדוגמה: 1) Target - ממשק השקע האירופאי, שאליו אנחנו רוצים להתחבר.
2) Adaptee - התקע האמריקאי עצמו.
3. Adapter - מחבר שמאפשר לנו לחבר תקע לשקע.

בדוגמה שלנו: 1) Target - ממשק השקע האירופאי, שאליו אנחנו רוצים להתחבר.
2) Adaptee - התקע האמריקאי עצמו.

```
public class USPlug {
    public void connectFlatPins()
}
```
תקע אמריקאי

```
public interface EuropeanSocket {
    void plugIn();
}
```
שקע אירופאי

```
public class USPlugAdapter implements EuropeanSocket {
    private USPlug usPlug;
    public USPlugAdapter(USPlug usPlug) {
        this.usPlug = usPlug;

    @Override
    public void plugIn() {
    usPlug.connectFlatPins();
    }
}
```


- דוגמת Iterator

מטרת הדוגמה: מתאם בין ממשקים שונים. לדוגמה: נניח שיש לנו מערכת שמצפה לעבוד עם שקע אירופאי, אך אנחנו רוצים לחבר אליה תקע אמריקאי. כדי לעשות זאת נצטרך מתאם שיאפשר לנו לחבר את התקע האמריקאי לשקע האירופאי.

תיעוד עיצובי - שלבים ונימוקים:

שלב 1 - תשתית Factory - הפרדת יצירת האובייקטים מהשימוש בהם.
שלב זה הוא הבסיס לכל השאר. הוא מאפשר להחליף את סוג החיישנים (אמיתיים / סימולציה) בשורה אחת בלבד, ללא שינוי בשאר הקוד.

Abstract Factory
שלב 2 - תוספת יצרן משפחה שלישית (Mock) לבדיקות, ללא שינוי בקוד הקיים. מדגים את יתרון ההרחבה של הדפוס.

Singleton - WeatherMonitoringStation מובטח להיווצר פעם אחת בלבד. מונע יצירת מספר תחנות מקבילות בטעות.

Reflection - טעינת מחלקות דינמית לפי שם מחרוזת מקובץ הגדרות. מאפשר החלפת יישום ללא שינוי קוד המקור.

Bridge - הפרדת ממשק המשתמש מהלוגיקה העסקית. מאפשר החלפת ממשק (Swing / Web) מבלי לגעת בלוגיקת החיישנים.

Observer - מנגנון ההתראות. מאפשר לכל רכיב להירשם לאירועים שמעניינים אותו, ללא תלות ישירה ביניהם.

AlarmClock הרחבת
הוספת יכולת שעון מעורר לתחנת מזג האוויר. השעון בודק בכל שנייה אם הגיעה השעה שהוגדרה, ומפעיל התראה חד-פעמית. ניתן להגדיר, לבטל ולבדוק סטטוס דרך ממשק ייעודי.

AlarmClock הרחבת - שלב 2
הוספת יכולת חזרתיות לשעון המעורר. ניתן להגדיר שעון מעורר שיופעל כל יום באותה שעה, ללא צורך להגדירו מחדש לאחר כל צלצול.

Bridge הרחבת - שלב 3
הוספת ממשק גרפי (Swing) כחלופה לממשק הטקסטואלי. שני הממשקים משתמשים באותה לוגיקה עסקית ומוצגים במקביל.

Reflection הרחבת - שלב 4
הוספת יכולת טעינת מחלקות בזמן ריצה לפי שם מחרוזת שנקרא מקובץ הגדרות. מאפשר החלפת יישום ללא שינוי בקוד המקור.

Snowin מחלקת הרחבת:
* מחלקת בסיס לחיישנים שמדמים ערכים משתנים בהדרגה. מחזיקה ערך נוכחי ומעדכנת אותו בכל קריאה.
* מחלקת בסיס לחיישנים בינאריים. מחזיקה מצב פנימי, מאפשרת הפעלה וכיבוי, ומחזירה 1 או 0 בהתאם למצב.
* מחלקת בסיס לחיישנים שמדמים ערכים משתנים בהדרגה. מחזיקה ערך נוכחי ומעדכנת אותו בכל קריאה.
* מחלקת בסיס לחיישנים בינאריים. מחזיקה מצב פנימי, מאפשרת הפעלה וכיבוי, ומחזירה 1 או 0 בהתאם למצב.

סיכום הקישור הנכון בין מחלקות

מה הקשר ההפוך?

* Dependency

משמעות: A מכיר את B (אין לו שדה מסוג B).

* Directed Association

משמעות: מחלקה A מחזיקה התייחסות למחלקה B

הערה: לפעמים - מסומנים ב־ אבל A יכול לגשת למופע של B.

* Bi-Directional Association

משמעות: יש אומר שכל A יודע, אבל כתוב שגם כל B יודע.

הערה: אין חיצים.

* Aggregation (סוג מיוחד של Association)

משמעות: מחלקה A היא מחלקה B, כך שלכל מופע B שלך יכול להיות מופע A יחיד. אין חיצים בקצה - A יודע מעט B יכול להיות מופע.

* Aggregation ב כלל

משמעות: כתוב שהרבה כל A יש, B מסוג אחד יש.

אבל כתוב שגם יש להם.

* Composition (סוג מיוחד של Aggregation)

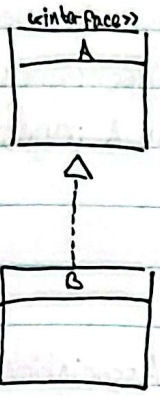
הערה: מחלקה B שדה (משתנה) שהוא A מסוג מסוים, שבו שום מופע אחר לא יכול להצביע.

משמעות: מחלקה מסוג A מכיל B ואין לשום אחד אחר.

* Composition ב כלל

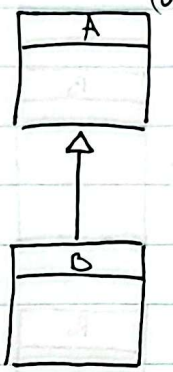
משמעות: כתוב שבכל כל A יש B מסוג אחד שהוא אחד יש.

אבל כתוב שגם יש להם.



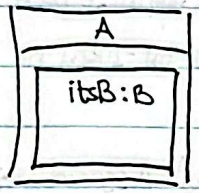
Realization *

A מממש את B :משמעות

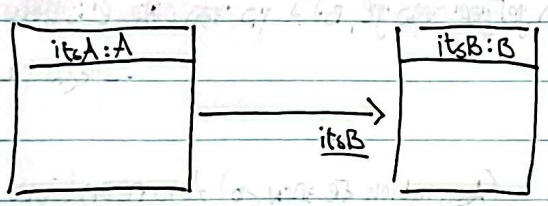


(Generalization) Inheritance *

A יורש מ B :משמעות



Composite *



Link *

LINK זה מופע של Association. כלומר

עיבוד מידע קבוצתי - Generics

- BiFunction - מקבל שני פרמטרים מסוג T,V ומחזיר פרמט מסוג R:
  דוגמה: `BiFunction<Integer, Integer, Integer> sum = (a,b) -> a+b;`
- Runnable - פונקציה ללא פרמטרים ומחזיר כלום: `Runnable r = () -> System.out.println("Hi");`
- Supplier - פונקציה ללא מחזיר T: `Supplier<Double> random = Math.random();`
- Consumer - מקבל T ולא מחזיר כלום: `Consumer<String> printing = w -> System.out.println(w);`
- Function - מקבל אחד T ומחזיר R: `Function<String, Integer> numOfWords = w -> w.split(" ").length;`
- Predicate - מקבל אחד מסוג T ומחזיר בוליאני: `Predicate<Integer> odd = n -> n.%2 == 0;`
- בג'אווה אין מושג מיוחד ב Java:
  דוגמא לביטוי למבדה: `(int x, int y) -> {x+y return x+y;}`
  למבדה: `(parameters) -> {body ...}`, היא אוסף רשימה של פרמטרים ובלוק קוד שיבוצע כאשר יקראו לה.
  * ביטוי למבדה (lambda): ביטוי אנונימי בלבד שמקבל כקלט אוסף של פרמטרים.
  לפונקציה מקבלת כפרמטר אחד או יותר מסוג A ומחזירה ערך מסוג A.
  - אם אין פרמטרים <> אז אפשר לכתוב רק סוגריים עגולים ואז אפשר לכתוב כל ביטוי.
  - Consumer Super (צרכן) - הפונקציה מקבלת כקלט List<? super A> ומחזירה ערך מסוג A.
  - Producer Extends (מפיק) - הפונקציה מקבלת כקלט List<? extends A> ומחזירה ערך כלשהו מסוג A.
  - פתרון ה"בעיה": אלגוריתם - הוא מתאר פתרון כללי לבעיה שניתן ליישם על כמה מופעים שונים של אותה בעיה.
  - מה זה אומר "?" וקלט PECS:
    כשאנחנו צריכים שיהיה אפשר לשלוח Mammal או משהו שיורש ממנו.
    כשאנחנו צריכים שיהיה אפשר לשלוח Mammal או משהו שהוא אב קדמון שלו.

  - Producer - מפיק, יוצר ערכים, Mammal ומעלה, כלומר יכול לשלוח Mammal או יורשים שלו.
  - Consumer - צרכן, יכול לקבל Mammal, כלומר יכול לקבל Mammal או אבות קדמונים שלו.
  * מחלקה אנונימית (anonymous class) היא מחלקה שאין לה שם ואנחנו יוצרים אותה רק פעם אחת.
  * לפעמים אפשר לממש פונקציונליות של מחלקה אנונימית בשורה אחת בלבד.
  * מחלקה אנונימית לא יכולה להכיל בנאי מפורש.

קיצור מלא לשעה - Streams

- Streams - מקבלת Source (כמו Lines of file, Arrays, Lists, collections וכו')
- פעולות שמחזירות stream (כמו skip(), flatMap(), distinct(), sorted(), map(), filter()) וכו')
ופעולות סופיות אחת (כמו anyMatch(), reduce(), collect(), forEach()) וכו').
- כל פעולה מקבלת stream ומחזירה stream, לכן ניתן לשרשר פעולות בין.
ניתן לשרשר רק פעולה סופית אחת, ולסוף חייב להיות terminal streams אחד.

דוגמה: מציאת המחרוזת השניה: `String string = stream.skip(1).findFirst().orElse(null);`
(פעולה סופית) (פעולה סופית)

- flatMap() - שטח את המבנה הפנימי של stream (stream של מערך/רשימה/קבוצות)
מקבל stream, מחזיר stream.

Packages - קיבוץ מחלקות לישות

- **REP - Reuse / Release Equivalence Principle** - המחלקות שנכללות יחד ברכיב צריכות לשמש ביחד
  ולכן ניתן לשחרר אותן יחד (באמצעות מערכת ניהול גרסאות - configuration managment).
  מומלץ שכל הקוד במחלקות יהיה בעל מטרה משותפת.
- **CRP - Common Reuse Principle** - אם רכיב משתמש בחלק מהמחלקות של רכיב אחר
  אז כל המחלקות ברכיב השני צריכות להיות רלוונטיות לו.
  כלומר, אין לרכיב לגרור קוד שלא נחוץ לו.
- **CCP - Common Closure Principle** - המחלקות ברכיב צריכות לשנות מסיבות דומות
  וכלל לא יותר מסיבה אחת. כלומר, אם מחלקה ברכיב צריכה להשתנות אז כל
  יתר המחלקות ברכיב צריכות להשתנות גם.
  כך שכל שינוי יגרום לשינוי ברכיב אחד בלבד.
- **ADP - Acyclic Dependencies Principle** - אסור שיווצר מעגל של תלויות בין הרכיבים.
  - תלויות יוצרות בעיות:
    כדאי לשמור את הרכיבים המשתנים הכי הרבה כמה שפחות תלויים בהם.
    יותר קל לשנות רכיב שלא תלויים בו רכיבים אחרים.
- **SDP - Stable Dependencies Principle** - תלויות בין רכיבים צריכות לכוון לכיוון היציבות.
  רכיבים צריכים להיות תלויים ברכיבים יציבים יותר מהם (פחות שינויים).
- **SAP - Stable Abstraction Principle** - רכיבים יציבים צריכים להיות מופשטים.
  כלומר, SDP ו SAP מתארים את אותו הדבר מכיוונים שונים.
  כל שינוי ברכיב יציב צריך להיות מבוסס על הרחבה ולא שינוי.
  - Dependency Inversion Principle - תלות בהפשטה ולא במימוש
    ראוי לתלות ברכיבים מופשטים יותר, שכן הם פחות משתנים.

מחלקה/רכיב יציב:

מחלקה/רכיב לא יציב:

* Stability = Responsibility (אחריות) ? Independence (תלויות)

איזה מלא יפה - Packages

* **מדד יציבות המבנה:**
  - Ce: מס' המחלקות החיצוניות התלויות במחלקות בתוך החבילה (ביא לחלוש)
  - Ca: מס' המחלקות החיצוניות אליהן, שתלויות במחלקות בתוך החבילה (ביא יציבות)

  כל חלוקת מחלקות שם לפחות כלפי חוצה Ca או Ce.

  דוגמה למשהו:

  Ce של P_1 הוא חיצוני אחד Ca של P_2.

  - משתמשים בה כאשר רוצים שכל המחלקות בחבילה יהיו יציבות. כלומר שאף אחת לא תלויה במחלקות חיצוניות.

* **מדד יציבות החבילה I:** $I = \frac{Ce}{Ca+Ce}$ $\quad$ $S = \frac{Ca}{Ca+Ce}$ $\quad$ יציבות S
  - כאשר $S=1$ החבילה יציבה מאוד.
  - כאשר $S=0$ החבילה לא יציבה מאוד.
  - משתמשים ב S: אם רוצים שהחבילה תהיה יציבה, כלומר שאר החבילות יהיו תלויות בה.
  - I (חוסר יציבות) $=1-S$: אם רוצים שהחבילה תהיה לא יציבה. כלומר שהיא תהיה תלויה בחבילות אחרות.

* **מדד יציבות המבנה D:** $D = \left|\frac{(S-A)}{\sqrt{2}}\right|$
  - שווה 0 אם $A+S \approx 0.707$
  - שווה 1 $\quad D = |(S-A)|$

  - ממשק Interface - מחלקה שיש לה רק מתודות של מחלקות אחרות בלי. מ"מ.
  - יש שני סוגים של מחלקות אבסטרקטיות: מחלקות וירטואליות לחלוטין (pure virtual) שנקראות גם ממשקים. כלומר שכל המתודות שלהן הן וירטואליות אבסטרקטיות. כלומר אין להן מימוש.
  - יש גם מחלקות וירטואליות חלקיות שנקראות מחלקות אבסטרקטיות. כלומר שיש להן מתודות וירטואליות אבסטרקטיות.
  - דגש: מחלקה אבסטרקטית לא חייבת להכיל מתודות וירטואליות אבסטרקטיות. $A = \frac{\text{מס' המחלקות האבסטרקטיות}}{\text{סה"כ המחלקות בחבילה}}$

* **מדד האבסטרקטיות (אבסטרקטיביות) של החבילה:**
  - דוגמה: (מה שראינו) כל הוא 1 כמות המחלקות שיש לו כמות המחלקות.
  - משתמשים ב A: הוא נותן אפשרות כמה החבילה יציבה, כלומר שאר החבילות יהיו תלויות בה.
  - A (אבסטרקטיות) $=1$: כל המחלקות בחבילה הן אבסטרקטיות. כלומר שאר החבילות יהיו תלויות בה.
  - כאשר $A=0$ החבילה היא קונקרטית לחלוטין.

- יש לחשב D עבור כל חבילה, ואחר כך לחשב ממוצע של כל ה D-ים של החבילות.

עקרונות עיצוב של מערכות ג'אווה ותוכנה (DRY, DIP, OCP).
- עקרון: עשה את הפעולה הכי פשוטה שיש, אל תחזור על עצמך (Keep it simple, stupid).
אל תוסיף פונקציונליות שאתה לא צריך כרגע.
מחלקה שמקבלת אחריות על כמה דברים, כלומר, יותר מסיבה אחת לשינוי, היא לא טובה.
* מחלקת Factory - נותנת לך פונקציה שיוצרת ומחזירה אובייקטים בהתאם לפרמטרים שקיבלה.
- מחלקה שמכילה רק פונקציות סטטיות ואין לה מצב פנימי.
אין מתודות. כולן כן יכול להיות מופע כזה או אחר.
- כולן public static ומקבלות את כמות הפרמטרים שהיא צריכה.
- יוצר static protected משתנה אחד שמחזיק את המופע של המחלקה.
- לא, לא נוצרת יותר protected כי אף אחד מהמחלקות היורשות לא יוכל.
- עקרון: ניתן לשנות התנהגות של מחלקה בלי לשנות את הקוד של המחלקה עצמה.
* תבנית Singleton - נותנת פונקציה ומבטיחה שמחלקה מסוימת תחזיק רק מופע אחד.
- עקרון: הפרד ממשקים, כלומר, עדיף הרבה ממשקים קטנים ולא ממשק אחד גדול.
* תבנית Iterator - מאפשר לעבור על אוסף, בלי שנדע מה הוא ואיך הוא בנוי.
- עקרון: תלות במופשטות ולא במחלקות קונקרטיות.
* תבנית Adapter - מאפשר למחלקות שלא יכולות לעבוד ביחד לעבוד ביחד.
- תבניות מבנה: Visitor, Iterator
- תבניות התנהגות (Behavioral patterns)
עסוקות בדרכים שבהן מחלקות או אובייקטים יתקשרו.
- תבניות מבנה: Adapter,
מאפשרות לבנות אובייקטים מורכבים מאובייקטים פשוטים יותר.
- תבניות יצירה (Creational patterns)
Factory method, Factory class, Singleton
- תבניות מבנה (Structural patterns)
מתעסקות בדרכים שונות ליצור אובייקטים.
- תבניות יצירה - עוסקות ביצירת אובייקטים

- Decorator - תבנית עיצוב מאפשרת להוסיף התנהגות לאובייקט בזמן ריצה
  תבנית עיצוב מתווך - תבנית עיצוב מאפשרת לאובייקטים לתקשר ביניהם
  בצורה עקיפה ולא ישירה. מקטינה את התלות בין האובייקטים ומאפשרת
  שינוי התנהגות בקלות.
* תבנית עיצוב Observer - תבנית עיצוב התנהגותית. מטרה: מנגנון הודעות
  בין אובייקטים. אובייקט אחד (Subject) שומר רשימה של תלויים (Observers)
  ומודיע להם על שינויים.
* תבנית עיצוב Composite - תבנית עיצוב מבנית. מטרה: לבנות אובייקטים
  מורכבים מאובייקטים פשוטים. מאפשרת לטפל באובייקט בודד וגם
  בקבוצה של אובייקטים באופן אחיד.
  Factory method מאפשרת לדחות את יצירת האובייקט לתת-מחלקות.
  Abstract Factory - מספקת ממשק ליצירת משפחות של אובייקטים קשורים
  ללא ציון המחלקות הקונקרטיות.
- Factory class - מחלקה עם שיטה אחת שיוצרת אובייקטים שונים
  בהתאם לפרמטרים שמקבלת.
  Factory method / Factory class על ידי - Abstract factory
  מחלקת האב מגדירה ממשק ליצירת אובייקט, אך לא יודעת איזה
  אובייקט ספציפי ייווצר. תת-המחלקות קובעות את סוג האובייקט
  שייווצר.
  Factory method - מגדירה ממשק ליצירת אובייקט, אך מאפשרת
  למחלקות יורשות להחליט איזה אובייקט ייווצר. מאפשרת לדחות
  את יצירת האובייקט לתת-המחלקות.
  Visitor - מאפשר להוסיף פעולות חדשות על מבנה נתונים
  מורכב מבלי לשנות את המחלקות עצמן.
1. Accept מקבל אובייקט Visitor ומפעיל עליו את visit המתאים.
- Visitor מגדיר מתודת visit נפרדת לכל סוג של אובייקט במבנה.
  מאפשר הוספת פעולות חדשות בקלות, אך קשה להוסיף סוגי
  אובייקטים חדשים.
  מבנה נתונים מורכב; אלגוריתמים שרוצים להפעיל על כל
  האובייקטים במבנה; כאשר רוצים להימנע מזיהום המחלקות
  בפעולות לא קשורות.
  Visitor מבנה מורכב; אלגוריתמים שרוצים להפעיל על כל
  האובייקטים במבנה; כאשר רוצים להימנע מזיהום המחלקות
  בפעולות לא קשורות.