האוניברסיטה הפתוחה

המחלקה למתמטיקה ולמדעי המחשב

תוכנת אמת מידה לתבניות עיצוב

פרויקט מתקדם לתואר שני

מגיש:

שרון הרוני

העבודה הוכנה בהדרכתו של פרופסור שמואל טישברוביץ

אפריל 2021

Contents

דיאגרמות

קוד

גרפים

מטרת הפרויקט

סוגים שונים של תבניות, במיוחד תבניות עיצוב (Design Patterns), הינם מושגים פופולריים ושימושיים בהנדסת תוכנה. תבניות אלה באות לתאר דרך לפתרון בעיה, אשר עשויות להיות שימושית במצבים רבים. במקרים מסוימים השימוש בתבניות עיצוב גורר ירידה בביצועי היישום. הואיל וביצועים הם לרוב הכרחיים לשמירה על איכות התוכנה, כדאי לחקור אילו תבניות עיצוב יכולות להשפיע על ביצועי היישומים, ובאיזה אופן. בפרויקט זה נבחן באופן מעשי האם שימוש בתבניות עיצוב יכול לגרום לירידה בביצועי המערכת, ובכמה אחוזים.

רקע

בספרות המקצועית מייחסים חשיבות רבה לפיתוח באמצעות תבניות עיצוב ולשימוש בקוד נקי כדוגמת עקרונות SOLID [1]. פרויקט מתקדם זה מהווה המשך ישיר לעבודה מסכמת אשר בה חקרנו עמידה של תבניות עיצוב בעקרונות SOLID. בעבודה מסכמת הראינו תחילה מדוע דווקא עקרונות אלו חשובים ביותר, לאחר מכן הצגנו את היתרונות בשימוש בעקרונות אלו, כגון: האצת תהליכי פיתוח, תחזוקה נוחה יותר, קריאות גבוהה יותר, ושיפור יכולת הבדיקה של המודולים. בנוסף הראינו כיצד תבניות עיצוב ויתר על כן שילוב של מספר תבניות עיצוב לשפת עיצוב (Design Patterns Language[2] [3]) שומרים על עקרונות אלו. נעזרנו בתבניות עיצוב אשר צוינו בספר GoF [3] לצורך ההדגמה. אחד העקרונות החשובים ביותר בפיתוח תוכנה אשר אליו לא התייחסנו בעבודה המסכמת, הינו עמידה בביצועים [4]. גם בספרות המקצועית אשר מגדירה תבניות עיצוב, נושא הביצועים לרוב אינו מוזכר, וזאת כיוון שביצועים תלויים במספר פרמטרים ובראש וראשונה בשפת התכנות.

באומרנו ביצועים אנו מתכוונים למשאבים אשר היישום צורך בזמן ביצועו. המשאבים העיקריים הנבחנים הינם הזיכרון וזמן ריצת המעבד (CPU). שימוש מיותר במשאבים אלו יכולים לגרום למספר בעיות כגון:

1. היישום ירוץ לאט יותר. בעיה זו יכולה לגרום במקרה הטוב לפגיעה בחוויית המשתמש, אך במקרה הרע - לאי עמידה מוחלטת במטרות היישום.
2. היישום יאט את היישומים האחרים במערכת. היישום משתמש במשאבי המערכת על חשבונם של יישומים אחרים, לדוגמא:
    a. צורך יותר מידי זיכרון, מה שמוביל למספר רב של page fault בזיכרון.
    b. צורך זמן ריצת מעבד רב, מה שמוביל לכך שיישומים אחרים מקבלים מעט מידי זמן ריצה ולכן נפגעים ביצועיהם.
3. ככל שהיישום משתמש יותר במעבד, כך צריכת ההספק במערכת עולה, ומשליכה ישירות על עליה בהוצאות החשמל במערכות נייחות [5] וירידה בזמני הסוללה במכשירים ניידים [6]. שמירה על הספק נמוך הינו אחד האתגרים הגדולים ביותר במערכות תוכנה כיום [5].

שימוש לא נכון ב- design patterns עלול אם כן, לגרום למספר רב של בעיות.

תיאור הפרויקט

פרויקט זה מדגים תוכנת אמת מידה (Benchmark) עבור מספר תבניות עיצוב אשר צוינו בספר GoF [3]. בספר, תבניות העיצוב מחולקות לשלוש קטגוריות: תבניות יצירה, תבניות מבנה, ותבניות התנהגות. בכדי לבחון את יעילותן או אי יעילותן של תבניות העיצוב, יצרנו מאגר של יישומים עבור תבניות עיצוב שאותן נרצה לבחון. לצורך יצירת המאגר נעזרנו בספרים כגון [7] ו-[8] ובאתרים מובילים ברשת. עבור כל יישום עם תבנית עיצוב, ייצרנו יישום דומה ללא תבנית עיצוב (הורדת תבנית עיצוב מיישום הינה פעולה פשוטה יחסית, המותירה את הפונקציונליות של היישום ללא שינוי). זה המקום לציין שביצענו מספר מועט של שינויים כגון הוספת ממשקים לתבניות העיצוב בכדי להתאימן לתוכנת אמת המידה.

שפת התכנות אשר אנו בוחנים עליה את ביצועים של תבניות עיצוב הינה JAVA. חשוב להדגיש שביצועי שפת תכנות אחת אינם מעידים על ביצועי שפות תכנות אחרות.

בחינת ביצועים בשפת JAVA אינה משימה פשוטה. ישנן אופטימיזציות רבות שה- JVM או החומרה הבסיסית עשויים להחיל על יישום כאשר אנו מריצים עליו את תוכנת אמת המידה. ייתכן שלא ניתן יהיה להחיל אופטימיזציות אלה כאשר היישום הינו חלק מיישום גדול יותר. בנוסף כתיבת תוכנת אמת מידה והטמעתה בצורה לא מיטבית עשויים לגרור תוצאות לא נכונות במדידה. לכן אנו חייבים לבודד את מה שברצוננו לבחון, כלומר יישומים קטנים, ובנוסף למזער ככל האפשר את האופטימיזציות של ה-JVM ושל החומרה.

לשם נטרול אופטימיזציות ה-JVM נעזר ב-JMH (Java Microbenchmark Harness), ונריץ תוכנת אמת מידה אשר בוחנת את היישומים על מספר מחשבים. כמו כן נשתמש ביישומים קטנים ככל האפשר.

דגש חשוב: ניתן יהיה להרחיב פרויקט זה עבור כל תבנית עיצוב אשר תוגדר בהמשך או אשר מוגדרת כיום, לדוגמא בספר [8] Patter Oriented Design Architecture. בנוסף יהיה אפשר להגדיל את כמות היישומים עבור כל תבנית אם נרצה בכך.

טבלה 1 - תבניות עיצוב GoF

Scope \ Family	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

JMH (Java Microbenchmark Harness)

בכדי לבחון ביצועים של תבניות עיצוב אנו משתמשים ב-JMH, שהינו toolkit (ארגז כלים) המאפשר מדידת ביצועים ביישומי JAVA בצורה נכונה תוך כדי נטרול JVM. יכולותיו העיקריות של ה-JMH אשר בהן אנו משתמשים בפרויקט זה, הינן:

1. מספר Fork לטובת הריצה.
2. Warmup – כמות איטרציות לחימום. איטרציות אלה משמשות לנטרול אופטימיזציות. תוצאות איטרציות אלו אינן מובאות בתוצאות הסופיות. בנוסף ניתן לציין כמה זמן לוקחת כל איטרציה.
3. Iteration – כמות האיטרציות עבורן נבחן את תוצאות הסופיות. בנוסף ניתן לציין כמה זמן לוקחת כל איטרציה.
4. הרצת איטרציות עם פרמטרים שונים.
5. שימוש ב- Xmx 2g , Xms 2g לטובת קביעת זיכרון התחלתי ומקסימלי לריצה (בנספח א' מוצגות תוצאות עבור קונפיגורציה נוספת).
6. הרצת Garbage Collector (GC) לאחר כל איטרציה.
7. שמירת נתונים ב-log file עבור כל תבנית עיצוב.
8. הצגת הנתונים ביחידות זמן מיקרו–שניה.
9. אופן הצגת זמנים – MODE.

המלצת הקונפיגורציה של JMH הינה ברירת המחדל:

יכולת	ברירת המחדל
Fork	**1**
Warmup #	**<5**
Warmup time	**10s**
Iteration #	**<5**
Iteration time	**10s**
GC	**true**
Mode	**Average Time**
args	**depends on design pattern**
Xms, Xmx	**2g**

אנו נותנים אופציה לשינוי פרמטרים אלה, אך נבחן ונתייחס אך ורק לתוצאות תחת קונפיגורציה הנ"ל.

אופן הצגת התוצאות הסופיות:

Benchmark	arg	Mode	Cnt	Score	Error	Units
testName1	100	avgt	5	13.276	± 1.129	us/op
testName1	10000	avgt	5	1405.492	± 120.687	us/op
testName2	100	avgt	5	15.114	± 3.314	us/op
testName2	10000	avgt	5	1436.151	± 277.952	us/op

הסבר לאופן הצגת התוצאות:

שמו של היישום הנבדק יופיע תחת Benchmark. ארגומנט המועבר ליישום (loop) יופיע תחת arg. Mode מציין באיזו שיטה ברצוננו לראות את התוצאות. כאמור נתייחס לממוצע של כל האיטרציות אשר נריץ. Cnt מציין את מספר הפעמים אשר מריצים כל יישום. בדוגמא לעיל אנו מריצים את testName1 חמש פעמים עבור arg=100 וחמש פעמים עבור arg=10000.

Score –הזמן שלקח לתוכנית לרוץ ביחידות מיקרו-שניה. מכיוון שאנו מריצים כל יישום 5 פעמים, יתכנו סטיות, שאותן ניתן לראות ב-Error Units שגם הן מוצגות ביחידות מיקרו-שניה.

אופן הצגת תוצאות ביניים:

Operation	**Description**
JMH version: 1.26 #	גרסת JMH
VM version: JDK 14, Java HotSpot(TM) 64- # Bit Server VM, 14+36-1461	גרסת Virtual Machine
VM invoker: C:\Program Files\Java\jdk- # 14\bin\java.exe	גרסת JDK
VM options: -Xms2g -Xmx2g #	אופציות של Virtual Machine
Warmup: 5 iterations, 10 s each #	איטרציות חימום: מספר כולל של איטרציות חימום, משך כל איטרציה
Measurement: 5 iterations, 10 s each #	איטרציות אמת: מספר כולל של איטרציות דגימה, משך כל איטרציה
Timeout: 10 min per iteration #	זמן מירבי לאיטרציה
Threads: 1 thread, will synchronize iterations #	מספר תהליכונים הפעילים
Benchmark mode: Average time, time/op #	אופן בדיקת אמת המידה
Benchmark: testName1 #	שם היישום אותו מודדים
Parameters: (arg = 100) #	ארגומנט שמועבר ליישום
Run progress: 80.00% complete, ETA 00:03:33 #	התקדמות: חלק הבדיקה שהסתיים באחוזים, זמן משוער עד סוף הבדיקה
Fork: 1 of 1 #	
Warmup Iteration 1: 4.606 us/op #	תוצאת איטרציות חימום
Warmup Iteration 2: 7.440 us/op #	תוצאת איטרציות חימום
Warmup Iteration 3: 9.424 us/op #	תוצאת איטרציות חימום
Warmup Iteration 4: 8.098 us/op #	תוצאת איטרציות חימום
Warmup Iteration 5: 8.108 us/op #	תוצאת איטרציות חימום
Iteration 1: 8.074 us/op	תוצאת איטרציות אמת
Iteration 2: 8.084 us/op	תוצאת איטרציות אמת
Iteration 3: 8.050 us/op	תוצאת איטרציות אמת
Iteration 4: 8.101 us/op	תוצאת איטרציות אמת
Iteration 5: 8.094 us/op	תוצאת איטרציות אמת

סיכום כללי עבור כל יישום:

Result "design_pattern_benchmarking.testName1:"

1.129 (99.9%)13.276± us/op [Average]

(min, avg, max) = (12.969, 13.276, 13.699), stdev = 0.293

CI (99.9%): [12.146, 14.405] (assumes normal distribution)

כאמור, את התוצאות נריץ על שני מחשבים:

- מחשב נייד – intel core i5
- מחשב נייח – intel core i7

מבנה כללי של הפרויקט

הפרויקט מכיל מספר רב של מחלקות (כ-170 מחלקות).

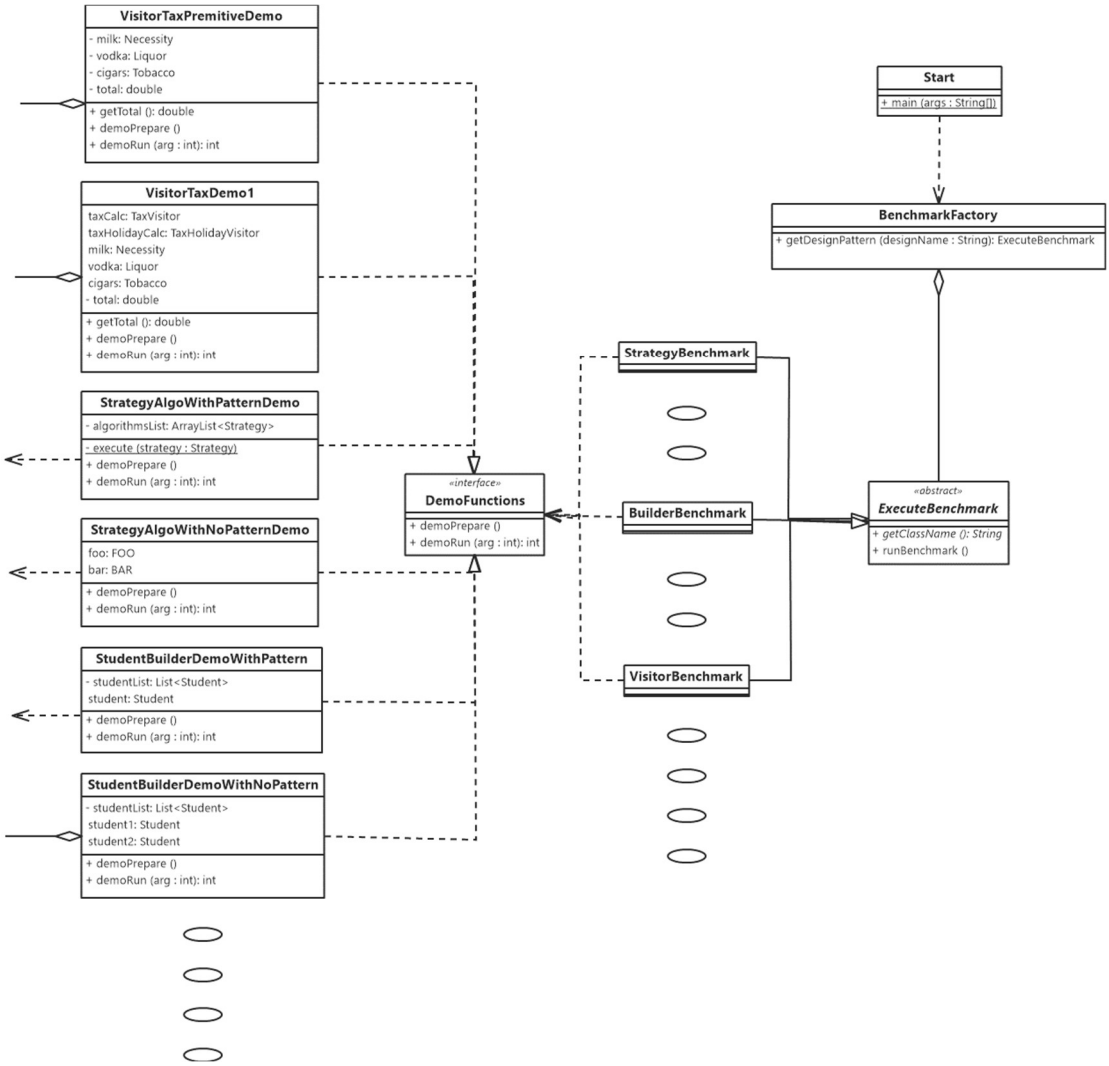
את הקוד ניתן למצוא ב- : https://github.com/sharon-haroni/benchmark-designPattern

את המחלקות של הפרויקט ניתן לחלק לשתי קבוצות עיקריות :

1. מחלקות אשר אחראיות על התפעול והביצוע של אמת המידה.
2. מחלקות אשר מממשות את תבניות העיצוב.

בפרק זה נפרט את מבנה הקבוצה הראשונה בלבד. פרוט עבור קבוצה השנייה יופיע בפרקים ותתי פרקים בהמשך.

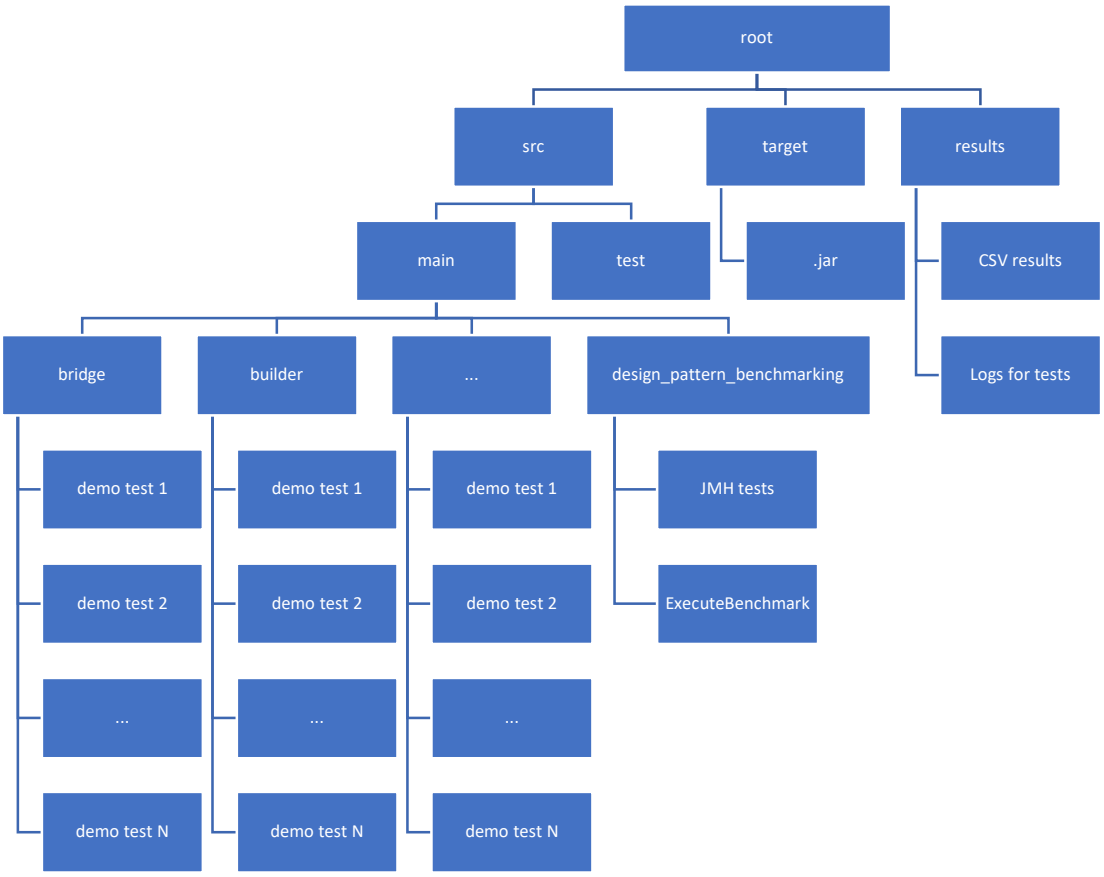
חשוב לציין, הדיאגרמה אינה מכילה את כל תבניות עיצוב.



דיאגרמת מחלקות 1 - מבנה כללי

תיקיות ומבנה הקבצים

התוכנית נכתבה תחת eclipse ושימוש ב-Maven.



ניתן לראות את מבנה הקבצים בספריית main המכילה את הקוד לפרויקט.

ספריית design_pattern_benchmarking מכילה את קוד המחלקות האחראיות על תפעול וביצוע אמת המידה.

המחלקות המממשות את תבניות העיצוב מופיעות תחת ספריה מתאימה (תתי ספריות הנוספות של main).

ספריית results מחזיקה את תוצאות הריצה אשר נותחו כאן (קבצי csv ו-log). כל הניתוחים בפרקים הבאים מבוססים על ריצה על שני מחשבים וזיכרון JVM של 2G. בנוסף ישנן ספריות נוספות עבור גדלי זיכרון JVM שונים.

ספריית target מחזיקה את התוכנית המקומפלת.

Design pattern benchmarking Main

לקבוצה הראשונה שייכות המחלקות הבאות:

- מחלקת Start – אחראית על בחירה של benchmark. המשתמש יוכל לבחור באחת או יותר אופציות. מחלקה זו יוצרת מחלקה מסוג BenchmarkFactory, ופונה אליה עם תבנית העיצוב הנדרשת לבדיקה ומריצה את הבדיקה.

```
public class Start {
	public static void main(String[] args) throws RunnerException {

		BenchmarkFactory benchmarkFactory = new BenchmarkFactory();
		System.out.println("Start Benchmark");

		ExecuteBenchmark builderBenchmark = benchmarkFactory.getDesignPattern("Builder");
		builderBenchmark.runBenchmark();

		ExecuteBenchmark strategyBenchmark = benchmarkFactory.getDesignPattern("Strategy");
		strategyBenchmark.runBenchmark();

		ExecuteBenchmark visitorBenchmark = benchmarkFactory.getDesignPattern("Visitor");
		visitorBenchmark.runBenchmark();

		System.out.println("End Benchmark");
	}
}
```

קוד 1 – קוד חלקי Start Class

- מחלקת BenchmarkFactory – מחלקה זו אחראית על תפעול ה-Benchmark הנבחר. כמו שניתן להבין מהשם, אנו משתמשים כאן בתבנית עיצוב factory method.

```
public class BenchmarkFactory {

	public ExecuteBenchmark getDesignPattern(String designName){
		if(designName == null){
			return null;
		}

		if(designName.equalsIgnoreCase("Strategy")){
			return new StrategyBenchmark();
		}
		else if(designName.equalsIgnoreCase("Builder")){
			return new BuilderBenchmark();
		}
		else if(designName.equalsIgnoreCase("Visitor")){
			return new VisitorBenchmark();
		}

		return null;
	}
ו
```

קוד 2 – קוד חלקי BenchmarkFactory

- מחלקת ExecuteBenchmark – מחלקה מופשטת. כל מחלקה אשר יורשת ממחלקה זו חייבת לממש את הפונקציונליות .getClassName. פונקציונליות runBenchmark ממומשת במחלקה זו ומשתמשים בה במחלקות היורשות. מתודה זו אחראית על הגדרת הפרמטרים הבסיסים לתוכנת אמת המידה ועל הפעלתה.

```
public abstract class ExecuteBenchmark {

	public abstract String getClassName();
	public void runBenchmark() throws RunnerException {
		System.out.println("ExecuteBenchmark: "+getClassName());

		Options opt = new OptionsBuilder()
				.include(getClassName())
				.shouldDoGC(true)
				.shouldFailOnError(true)
				.forks(1)
				.threads(1)
				.result(getClassName()+".csv")
				.resultFormat(ResultFormatType.CSV)
				.output(getClassName()+".log")
				.warmupIterations(5)
				.warmupTime(TimeValue.seconds(10))
				.measurementIterations(5)
				.measurementTime(TimeValue.seconds(10))
				.timeUnit(TimeUnit.NANOSECONDS)
				.jvmArgs("-Xms2g", "-Xmx2g")
				.mode(Mode.AverageTime)
				.build();

		new Runner(opt).run();
	}
}
```

קוד 3 - ExecuteBenchmark

- מחלקות אשר מתפעלות benchmark עבור כל תבנית עיצוב. בדיאגרמה ניתן לראות (לשם נוחיות והבנה) רק שלוש מחלקות כאלה (StrategyBenchmark, BuilderBenchmark, VisitorBenchmark). כל מחלקה אחראית לתפעל את המחלקות אשר מממשות תבניות עיצוב. תפעול המחלקות מתבצע באמצעות הממשק DemoFunction.

  בנוסף מחלקות אילו משתמשות ב-JMH לטובת ביצוע אמת המידה, תוך מימוש המתודות הבאות : setup ו- benchmark.

```
@State(Scope.Thread)
public class BuilderBenchmark extends ExecuteBenchmark{
	@Param({"1"})
  public int arg;

	StudentBuilderDemoWithPattern studentBuilderDemoWithPattern;
	StudentBuilderDemoWithNoPattern studentBuilderDemoWithNoPattern;

	@Setup
	public void setup() {
		studentBuilderDemoWithPattern = new StudentBuilderDemoWithPattern();
		studentBuilderDemoWithPattern.demoPrepare();
		studentBuilderDemoWithNoPattern = new StudentBuilderDemoWithNoPattern();
		studentBuilderDemoWithNoPattern.demoPrepare();
	}

	@Benchmark
	@OperationsPerInvocation(1)
	public int measureStudentBuilderDemoWithPattern() {
		return studentBuilderDemoWithPattern.demoRun(arg);
	}

	@Benchmark
	@OperationsPerInvocation(1)
	public int measureStudentBuilderDemoWithNoPattern() {
		return studentBuilderDemoWithNoPattern.demoRun(arg);
	}

	public String getClassName()
	{
		return BuilderBenchmark.class.getSimpleName();
	}
}
```

קוד 4 - דוגמא חלקית BuilderBenchmark

- ממשק DemoFunction – ממשק זה מכיל שתי מתודות : demoPrepare ו- demoRun. demoPrepare נקראת בזמן יצירה של האובייקטים (במידה ויש בכך צורך). מתודה זו אינה כלולה בחישובי הביצועים של היישום, כלומר אנו לא מריצים אמת מידה עבורה. demoRun נקראת בזמן ריצה של התוכנית, ועבורה אנו מפעילים את תוכנית אמת המידה.

לקבוצה השנייה שייכות המחלקות המיישמות את תבניות העיצוב אשר אנו מעוניינים לבדוק עליהן את אמות המידה.

אנו נבחן את תבניות העיצוב הבאות :

Builder ,Prototype ,Strategy ,Visitor ,Adapter ,Iterator ,Bridge ,State, Observer.

כעת נעבור לקבוצה השנייה (מחלקות אשר מממשות את תבניות העיצוב) ונציג כל תבנית עיצוב בנפרד.

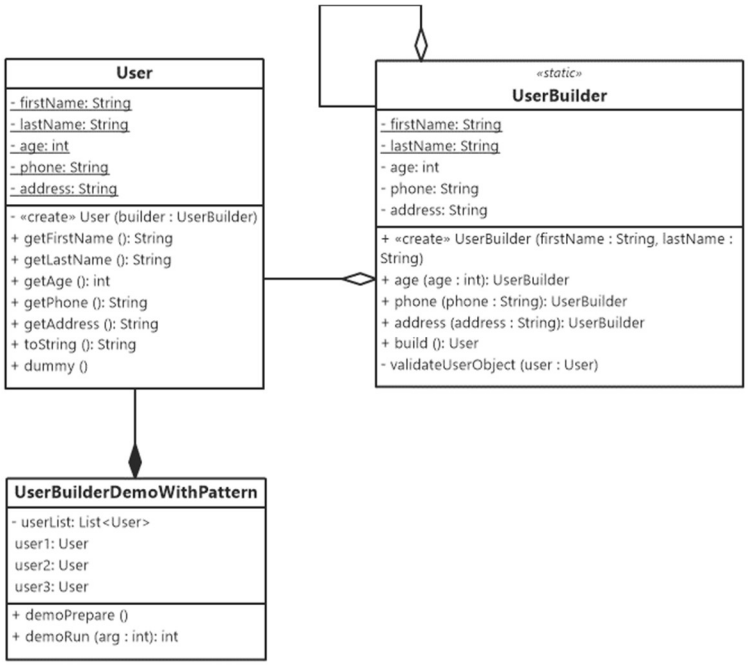
Builder

Builder הינה תבנית עיצוב מקבוצת creational אשר שייכת לתהליך יצירת האובייקטים. היא מאפשרת בנייה של אובייקטים שונים מבלי שעבור כל אחד מהסוגים השונים יהיה constructor נפרד ו/או שייווצר ממחלקה נפרדת, ובאופן שתהליך הבניה יהיה (כמעט) זהה לכולם כך שנוכל בקלות יחסית לעבור מבנייתו של אובייקט מסוים לבנייתו של אובייקט מסוג אחר. כדי לבחון באם תבנית עיצוב זו פוגעת בביצועים או לא, נבחן להלן מספר דוגמאות.

User

תחילה נראה שימוש בתבנית עיצוב Builder באופן קלאסי כאשר מחלקת User מחזיקה מחלקה פנימית בשם UserBuilder אשר אחראית לבנייה של המחלקה User לפי כל הקומבינציות האפשריות.

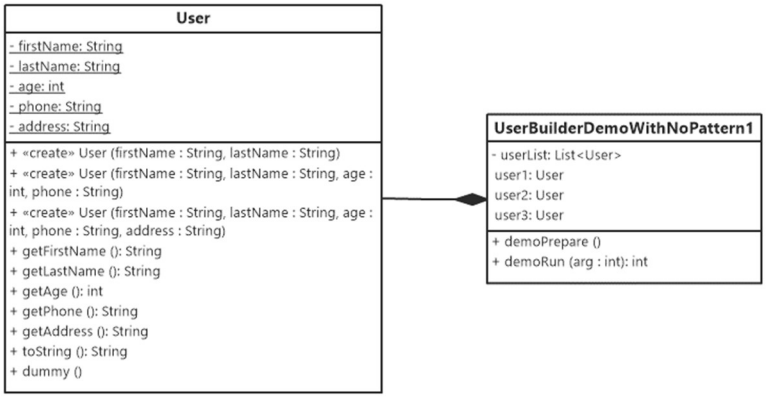
הקמת User מתבצעת באמצעות מחלקת Demo.



דיאגרמת מחלקות 2 – Builder UserWithPattern

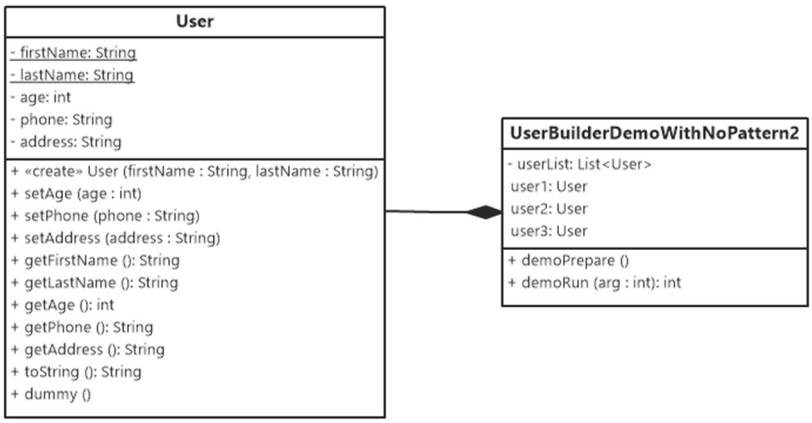
כעת נבחן שתי חלופות לתבנית העיצוב builder.

החלופה הראשונה הינה הקמת User לפי פרמטרים שונים, כאשר כל הפרמטרים אינם חובה ונחוצות מספר מתודות בנאי. ניתן לראות בדוגמא מספר מצומצם (לשם פשטות) של מתודות בנאי (שלוש מתודות).



דיאגרמת מחלקות 3 – Builder UserWithNoPattern1

בחלופה השנייה נשתמש במתודת בנאי בודדת, אבל לכל משתנה ניתן יכולת set. אופציה זו אינה אפשרית אם המחלקה צריכה לקבל את כל הפרמטרים בבנאי. אבל שימוש בשיטה זו חוסך כתיבת מספר רב של מתודות בנאי.

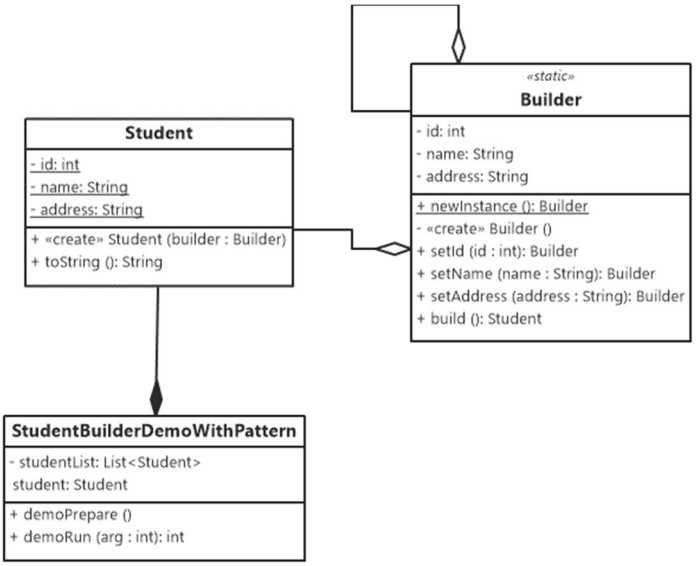


דיאגרמת מחלקות 4 – Builder UserWithNoPattern2

Student

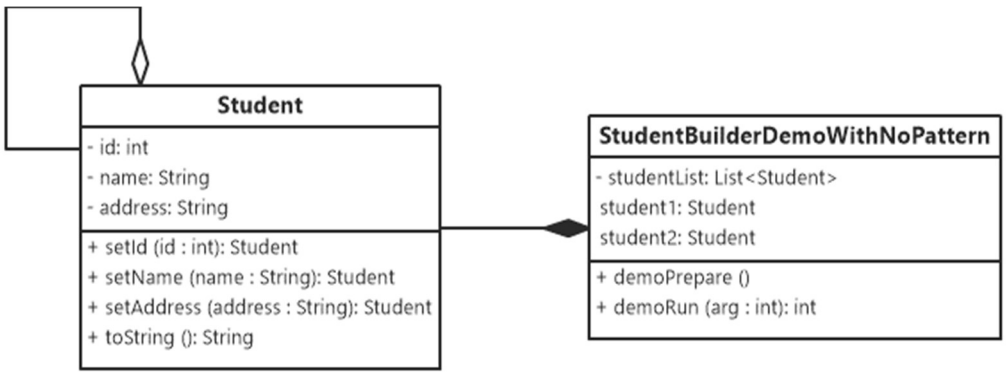
בדוגמא זו ניצור מחלקה אשר אחראית על יצירת סטודנטים. נעשה זאת בשני אופנים - עם ובלי תבנית עיצוב, ונבחן את זמן ריצה בכל אחד מהמקרים.

באופן הראשון תבנית העיצוב דואגת לנו לכל הקומבינציות האפשריות והעתידיות ליצירת סטודנט לפי תבנית עיצוב Builder.



דיאגרמת מחלקות 5 – Builder StudentWithPattern

כעת נוותר על תבנית העיצוב ובמקומה נשתמש במתודות set. באופן זה נצטרך ליצור מתודות set לארגומנטים השונים, כאשר כל מתודה מחזירה את מחלקה 'סטודנט' (בשונה מהדוגמא הקודמת). כמובן שאופציה זו אינה נותנת את אותן היכולות אם אנו נדרשים ליצור את האובייקט רק במחלקת הבנאי.



דיאגרמת מחלקות 6 – Builder StudentWithNoPattern

אמת מידה

אמת המידה הורצה על מספר מחשבים עבור כל אחד מהיישומים.

מחשב מספר 1:

Benchmark	Samples	Score	Score Error (99.9%)	Unit
StudentBuilderDemoWithNoPattern	10	89.02104	1.70504	ns/op
StudentBuilderDemoWithPattern	10	107.7222	2.337033	ns/op
UserBuilderDemoWithNoPattern1	10	157.6171	3.564887	ns/op
UserBuilderDemoWithNoPattern2	10	167.187	2.257829	ns/op
UserBuilderDemoWithPattern	10	185.2488	4.184443	ns/op

מחשב מספר 2:

Benchmark	Samples	Score	Score Error (99.9%)	Unit
StudentBuilderDemoWithNoPattern	10	22.03884	2.000455	ns/op
StudentBuilderDemoWithPattern	10	23.39383	0.178539	ns/op
UserBuilderDemoWithNoPattern1	10	34.05911	0.32127	ns/op
UserBuilderDemoWithNoPattern2	10	35.44804	0.161221	ns/op
UserBuilderDemoWithPattern	10	51.28395	0.184628	ns/op

ניתוח התוצאות:

שתי תובנות ברורות עולות מהתוצאות:

1. תבנית עיצוב Builder עולה זמן (אך לא הרבה יותר מיישום ללא תבנית עיצוב). אך חשוב לזכור שתבנית זו שייכת למשפחת תבניות עיצוב creational אשר בד"כ לא רצות מספר רב של פעמים במהלך התוכנית.
2. ניתן לראות שהגדרת מחלקת בנאי לכל אופציה של יצירה (Pattern1) הינה החסכונית ביותר מבחינת הזמן. אך כאמור זה בא על חשבון קריאות ותחזוקה (וכמובן נפח זיכרון - memory).

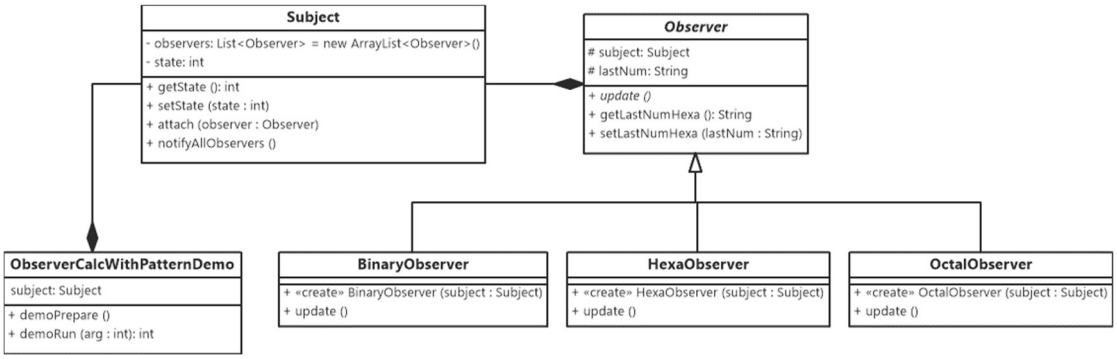
גרף 1 - סיכום ממוצע Builder

Observer

תבנית עיצוב זו שייכת למשפחת תבניות Behavioral אשר מאפיינות את הדרכים בהן מחלקות ועצמים מתקשרים למחלקות אחרות. היא מגדירה תלות של אחד לרבים בין עצמים, כך שכאשר האחד משנה את מצבו, העצמים התלויים מקבלים הודעה ומתעדכנים אוטומטית. כדי לבחון האם תבנית עיצוב זו פוגעת בביצועים או לא, נבחן מספר דוגמאות.

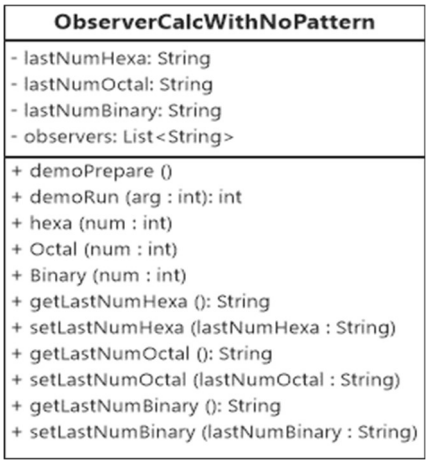
Calc

כאשר נשתמש בתבנית עיצוב למעשה נשתמש ביכולותיה כדי להודיע למחלקות אשר רשומות אליה. בדוגמא הבאה ניתן לראות שמחלקת Demo שולחת הודעה למחלקת Subject, וזו יודעת למי להעביר את ההודעה.



דיאגרמת מחלקות 7 – Observer CalcWithPattern

בדוגמא הבאה, ללא שימוש בתבנית עיצוב, נבצע את הקריאות ואת הפעולות בתוך מחלקה בודדת, כלומר שימוש ב if-else פשוט ללא פיצול למחלקות.

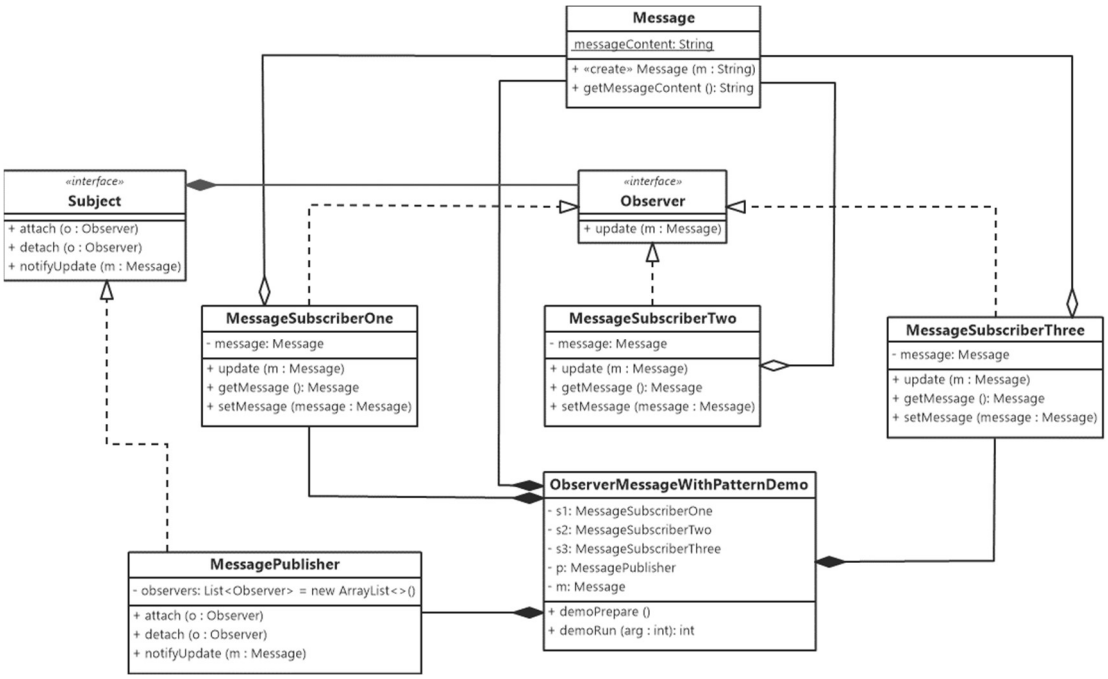


דיאגרמת מחלקות 8 – Observer CalcWithNoPattern

Message

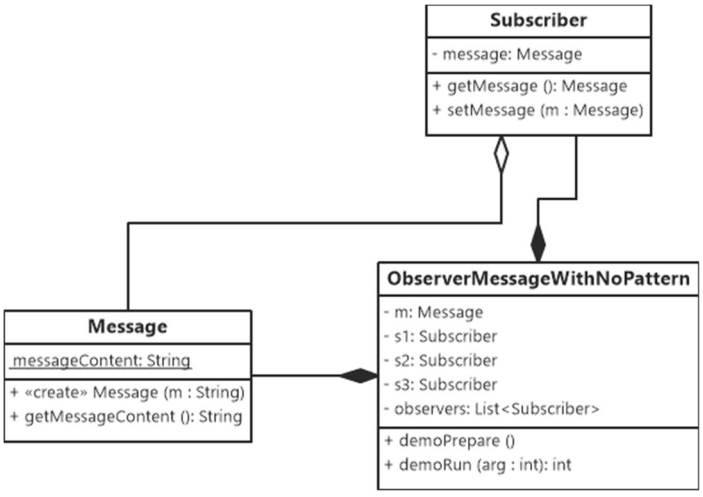
כעת נממש שתי דוגמאות המדגימות רישום ושליחת הודעות.

תחילה באמצעות תבנית העיצוב Observer. ניתן לראות מדיאגרמת המחלקות שיש שימוש במחלקת Observer לטובת שליחת הודעות לנרשמים להודעה, ובמחלקת MessagePublisher לטובת פרסום ההודעה.



דיאגרמת מחלקות 9 - Observer MessageWithPattern

וכעת ללא שימוש בתבנית עיצוב. בשונה מהדוגמא הקודמת (Calc), בדוגמא זו נשתמש במספר מחלקות לטובת רישום ושליחת הודעות, אך כאמור ללא תבנית עיצוב Observer.



דיאגרמת מחלקות 10 - Observer MessageWithNoPattern

אמת מידה

מחשב מספר 1:

Benchmark	Samples	Score	Score Error (99.9%)	Unit
ObserverCalcWithNoPattern	10	74.964506	1.644733	ns/op
ObserverCalcWithPatternDemo	10	86.073123	1.219948	ns/op
ObserverMessageWithNoPattern	10	10.200924	0.047155	ns/op
ObserverMessageWithPatternDemo	10	26.840292	2.548788	ns/op

מחשב מספר 2:

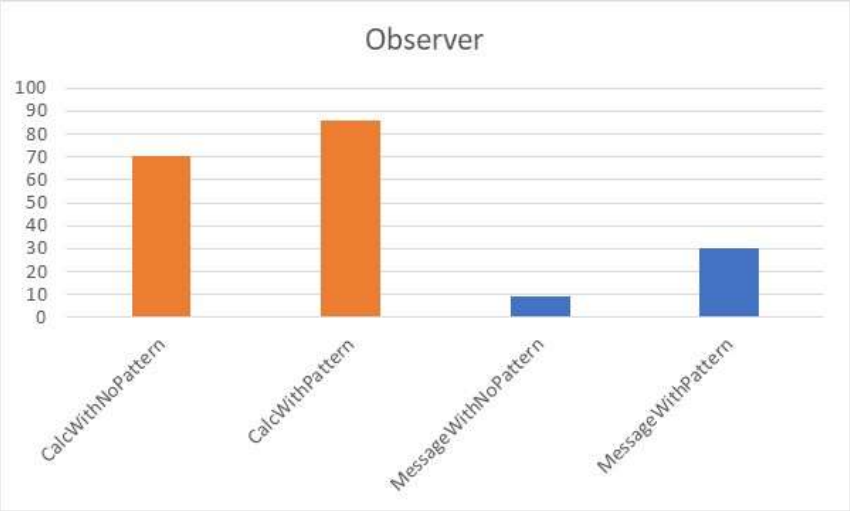
Benchmark	Samples	Score	Score Error (99.9%)	Unit
ObserverCalcWithNoPattern	10	65.919563	0.653009	ns/op
ObserverCalcWithPatternDemo	10	85.450224	4.476323	ns/op
ObserverMessageWithNoPattern	10	7.734798	0.05806	ns/op
ObserverMessageWithPatternDemo	10	33.552894	10.315675	ns/op

ניתוח התוצאות:

ניתן לראות ששימוש ב-pattern עולה בזמן, והזמן תלוי בגודל ובמורכבות ההודעה.

בדוגמא הראשונה - Calc - התשלום לא גדול יחסית, כיוון שכל תהליך היצירה מתבצע ב-demoPrepare. ב-demoRun אנו מודדים אך ורק את שליחת ההודעה. ובמקרה הזה notifyAll ללא בדיקה מי נרשם ומי לא.

בדוגמא השנייה - Message - התשלום גדול יותר. זאת בעיקר בגלל גודל ההודעה המועברת וכמות המחלקות והאובייקטים המעורבים.



גרף 2 - סיכום ממוצע Observer

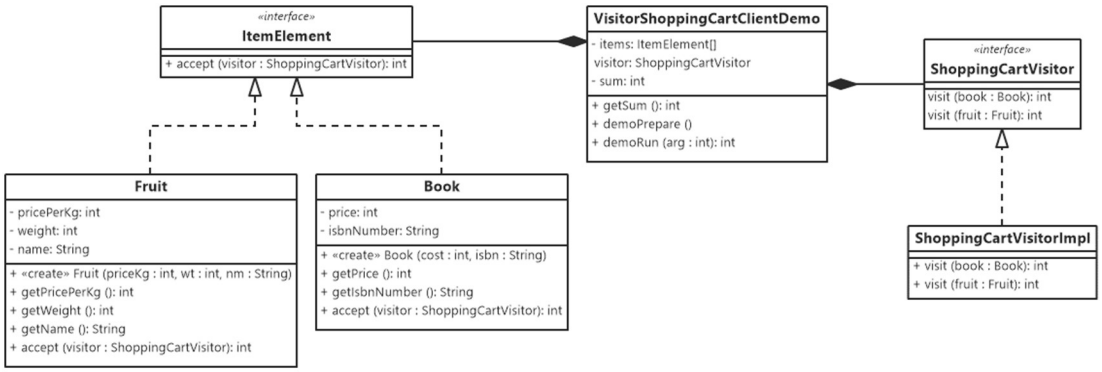
Visitor

תבנית עיצוב זה שייכת ל תבניות עיצוב ממשפחת Behavioral אשר מאפיינות את הדרכים בהן מחלקות ועצמים מתקשרים למחלקות אחרות. נשתמש בתבנית עיצוב זה, כאשר נדרש הוספת פעולות נוספות (או שינוין של פעולות קיימות) באובייקטים שעימם אנו עובדים מבלי שיהיה צורך לבצע שינויים בקוד המקור של המחלקות שמהן הם נוצרו.

כדי לבחון האם תבנית העיצוב זה, פוגעת בביצועים או לא, נבחן מספר דוגמאות.

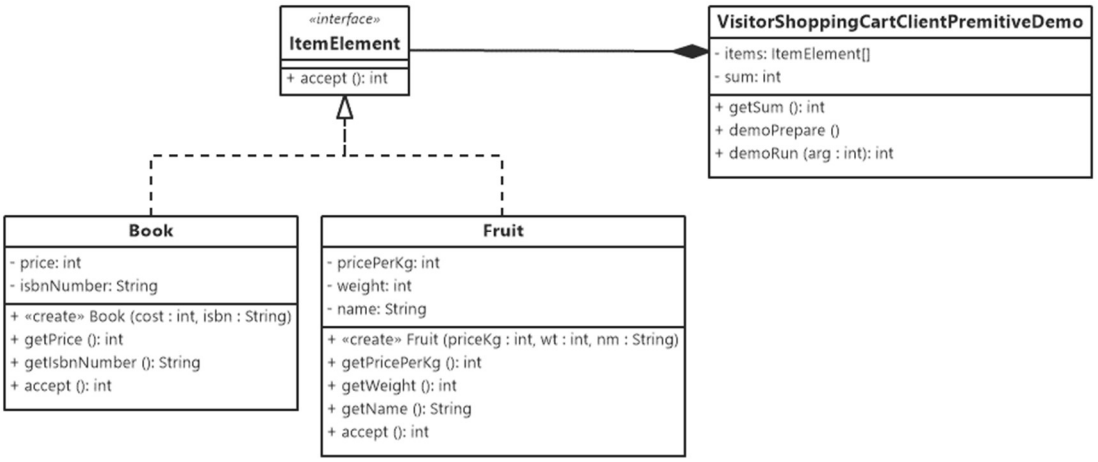
supermarket

מחלקת ה-Demo מחזיקה אובייקטים מסוג item ו- ShoppingCartVisitor. מתודת accept באובייקטים שממשים את Item מקשרים בין Fruit ו-Book לבין ShoppingCartVisitorImpl.



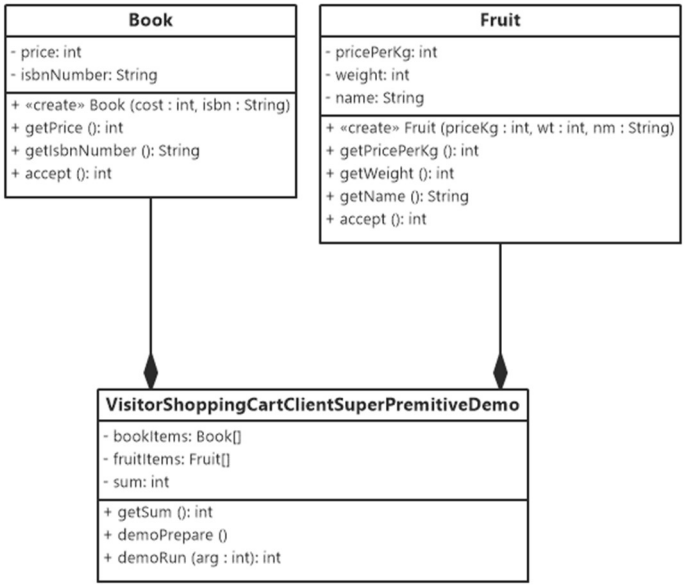
דיאגרמת מחלקות 11- Visitor SuperMarketWithPattern

כעת נרדד את תבנית העיצוב. כלומר לא תהיה מחלקה חיצונית בה נבקר בכדי לעשות את החישובים, אלא כל מחלקה תבצע את החישובים בעצמה. ניתן לראות שהפונקציונליות אינה נפגעת.



דיאגרמת מחלקות 12- Visitor SuperMarketWithNoPattern1

בשלב זה נרדד עוד יותר את המחלקות, תוך שמירה על הפונקציונליות. לשם כך נוותר גם על item, כלומר נוותר על הממשק שהאובייקטים מממשים.



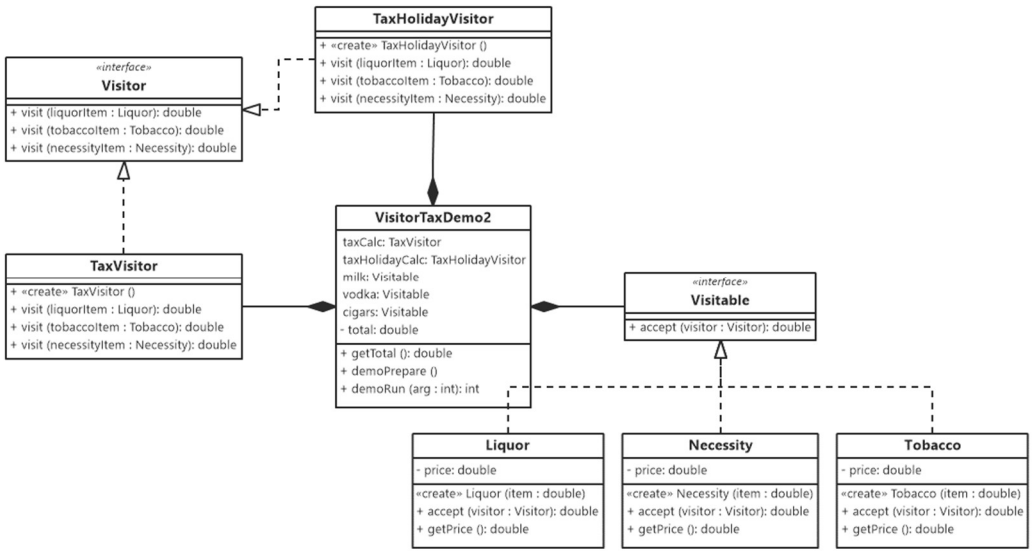
דיאגרמת מחלקות 13- Visitor SupermarketWithNoPattern2

Tax

בדוגמא שלהלן נתחיל שוב עם תבנית עיצוב.

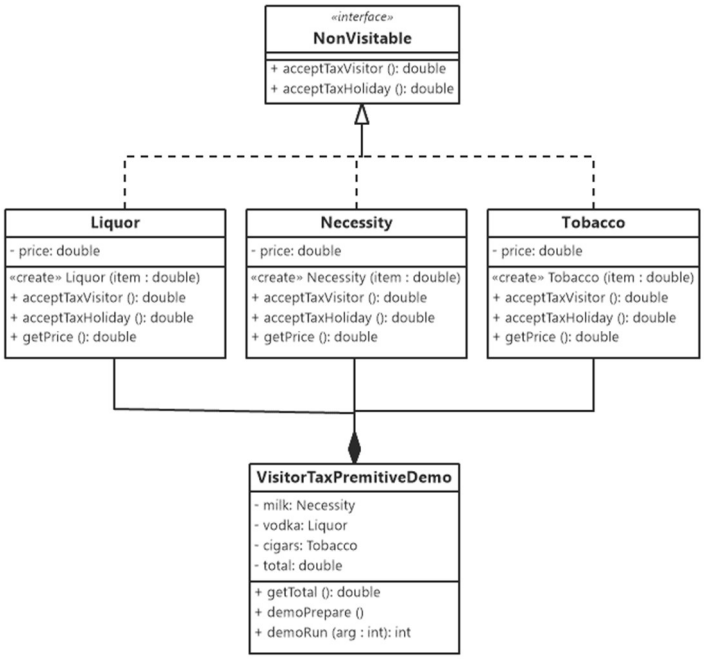
בדוגמא ישנם שני סוגי TAX - TAX רגיל ו-TAX לעונות של חופשה. מטרת היישום לחשב את הערך הסופי של המוצרים לפי סוג ה-TAX.

כאן אנו רואים דיאגראמת מחלקות ל-Demo2. ב-Demo1 מחלקת demo מחזיקה ישירות את המוצרים, כלומר אנו מוותרים על ממשק Visitable.



דיאגרמת מחלקות 14 - Visitor TaxWithPattern

בשלב זה, כמו בדוגמאות הקודמות, נרדד את תבנית העיצוב תוך שמירת הפונקציונאליות. כעת, כל מוצר מחזיק את חישוב של TAX - TAX רגיל ו-TAX בזמני חופשה.



דיאגרמת מחלקות 15- Visitor TaxWithNoPattern

אמת מידה

אמת המידה הורצה על מספר מחשבים עבור כל אחד מהיישומים.

מחשב מספר 1 :

Benchmark	Samples	Score	Score Error (99.9%)	Unit
VisitorShoppingCartClientWithNoPattern1	10	10.997813	0.040415	ns/op
VisitorShoppingCartClientWithNoPattern2	10	9.89334	0.015761	ns/op
VisitorShoppingCartClientWithPattern	10	11.306983	0.049661	ns/op
VisitorTaxWithNoPattern	10	6.894508	0.040036	ns/op
VisitorTaxwithPattern1	10	7.211905	0.045725	ns/op
VisitorTaxwithPattern2	10	7.553914	0.120719	ns/op

מחשב מספר 2 :

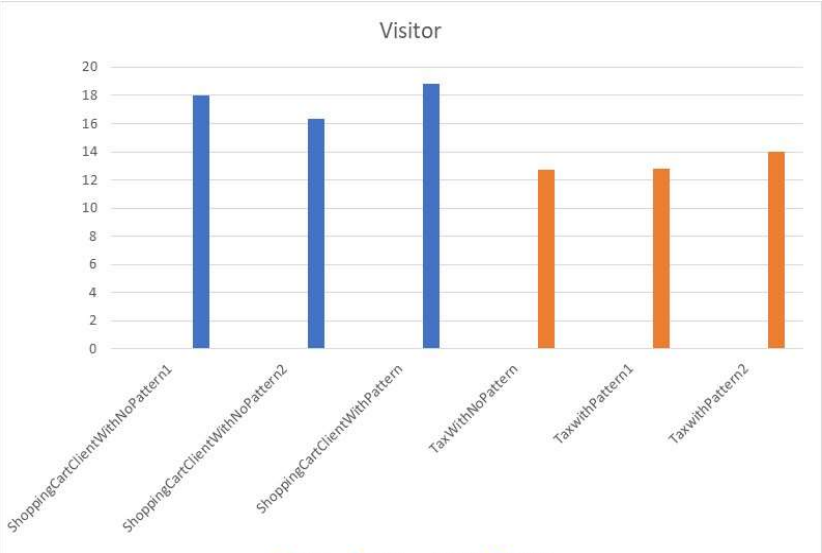
Benchmark	Samples	Score	Score Error (99.9%)	Unit
VisitorShoppingCartClientWithNoPattern1	10	7.022476	0.045738	ns/op
VisitorShoppingCartClientWithNoPattern2	10	6.435503	0.043278	ns/op
VisitorShoppingCartClientWithPattern	10	7.503286	0.053558	ns/op
VisitorTaxWithNoPattern	10	5.843536	0.787262	ns/op
VisitorTaxwithPattern1	10	5.619592	0.03914	ns/op
VisitorTaxwithPattern2	10	6.435933	0.03861	ns/op

ניתוח התוצאות :

ניתן לראות בשתי הדוגמאות שישנה עלות בשימוש בתבנית העיצוב visitor, אך לפי התוצאות שקיבלנו עלות זו זניחה למדי.

בנוסף, ניתן לראות בדוגמאות shopping שישנו הבדל בין כתיבת התוכנית ללא ממשקים (demo2) לבין כתיבת התוכנית עם ממשקים (demo1). אך עבור בדיקה שעשינו ביישום Tax קיבלנו תוצאות הפוכות. לכן לא ניתן להסיק שהוספת ממשקים משפיעה על זמני ריצת תוכנית.

מהנ״ל ניתן ללמוד ששימוש בתבנית עיצוב זו אינו משפיע על ביצועי התוכנית.



גרף 3 - סיכום ממוצע Visitor

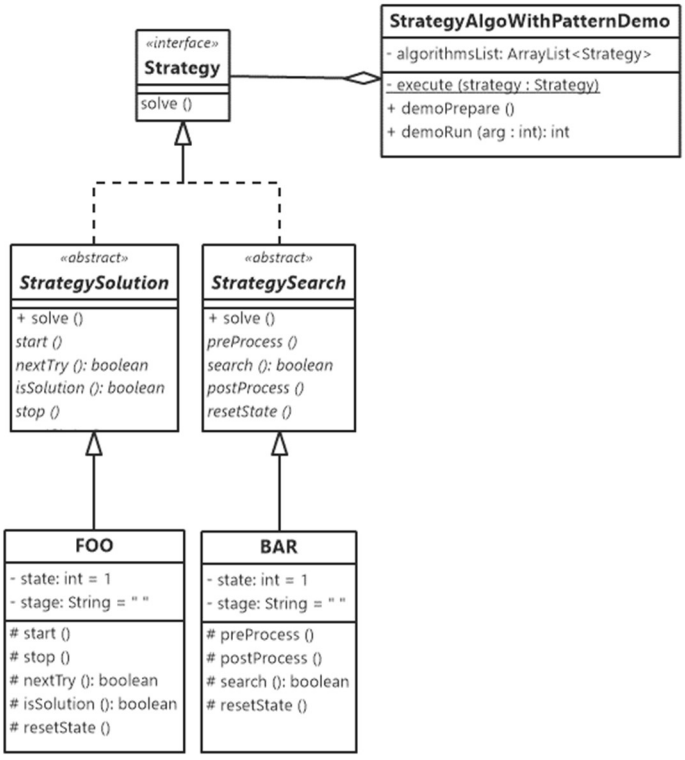
Strategy

תבנית עיצוב זו שייכת לתבניות עיצוב ממשפחת Behavioral אשר מאפיינות את הדרכים בהן מחלקות ועצמים מתקשרים למחלקות אחרות. תבנית עיצוב זו באה לספק הכמסה למשפחה של אלגוריתמים ולעשותם ברי החלפה, לאפשר לאלגוריתמים להשתנות באופן בלתי תלוי בלקוחות.

כדי לבחון באם תבנית עיצוב זו פוגעת בביצועים או לא, נבחן מספר דוגמאות.

Advanced Algo

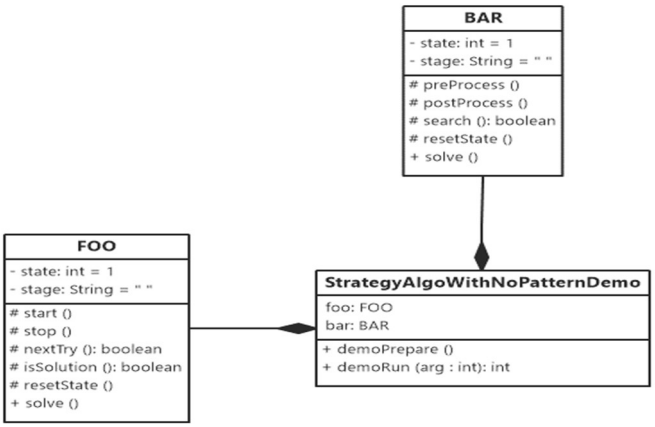
בדוגמא זו, ישנם שני סוגי אלגוריתמים. אלגוריתם מסוג search ואלגוריתם מסוג solution. תחילה נשתמש בתבנית העיצוב Strategy. client מכיר אך ורק את הממשק Strategy לטובת מימוש האלגוריתם, כאשר FOO ו-BAR מממשים את האלגוריתם. חשוב לציין, שהמחלקות האבסטרקטיות אינן חובה.



דיאגרמת מחלקות 16- Strategy AdvancedAlgoWithPattern

כפי שניתן לראות client מכיר את ממשק Strategy, ומפעיל את האלגוריתם באמצעות קריאה ל-Solve. בנוסף, BAR ו-FOO ממשים כל אחד אלגוריתם שונה ממש. האחד - אלגוריתם חיפוש, והשני - אלגוריתם פתרון. אך שניהם יורשים מהמחלקות האבסטרקטיות.

כעת נפשט את היישום על ידי כך שנוותר על המחלקות האבסטרקטיות ועל הממשק Strategy. כלומר FOO ו-BAR יישמו את האלגוריתמים ויספקו ממשק ל-client באופן ישיר.



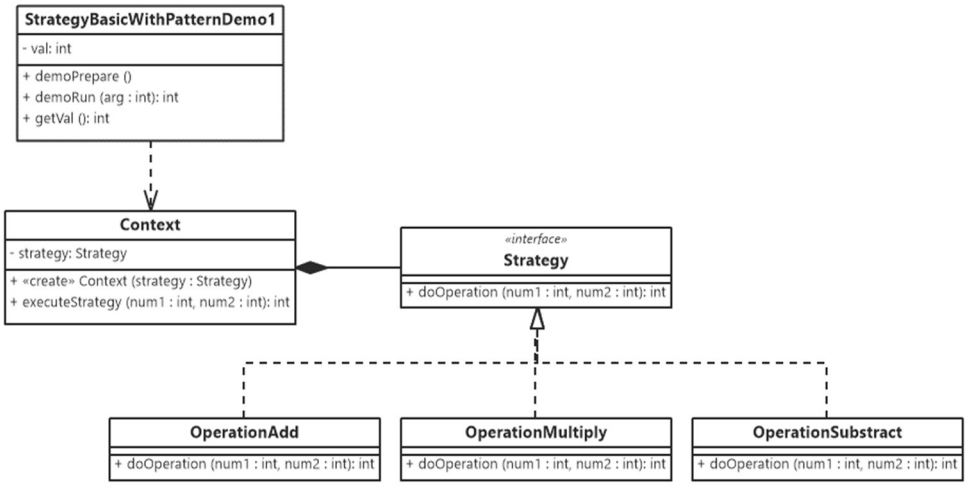
דיאגרמת מחלקות 17- Strategy AdvancedAlgoWithNoPattern

Simple Algo

כעת נבחן דוגמא נוספת למימוש Strategy. תחילה נראה מימוש אשר בו יצירת אובייקט מסוג Context וסוגים שונים של אלגוריתמים וכן קריאה לאלגוריתם (doOperation), מתבצעים בזמן ריצה.

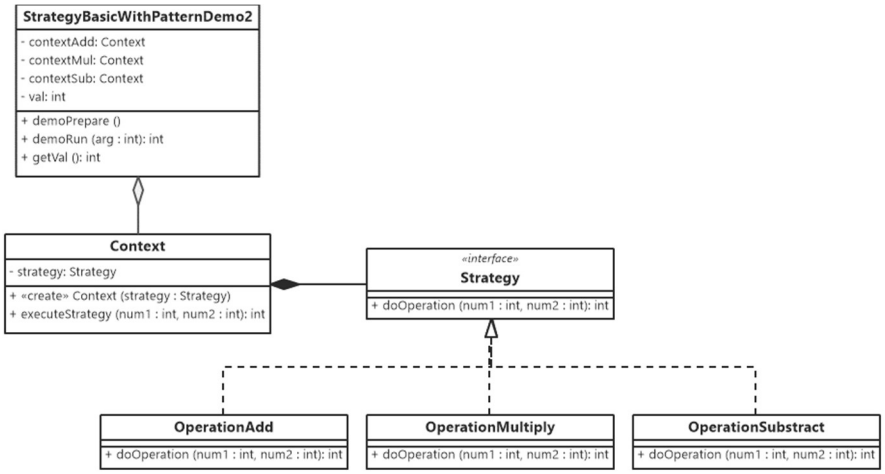
בדוגמא זו ישנם שלושה סוגי אלגוריתם - add, multiply, substract. כל אלגוריתם צריך לממש את הממשק strategy.

ה-Client מתממשק לסוגים השונים של האלגוריתם באמצעות מחלקה מסוג context אשר מכילה את ממשק strategy.



דיאגרמת מחלקות 18- Strategy SimpleAlgoWithPattern1

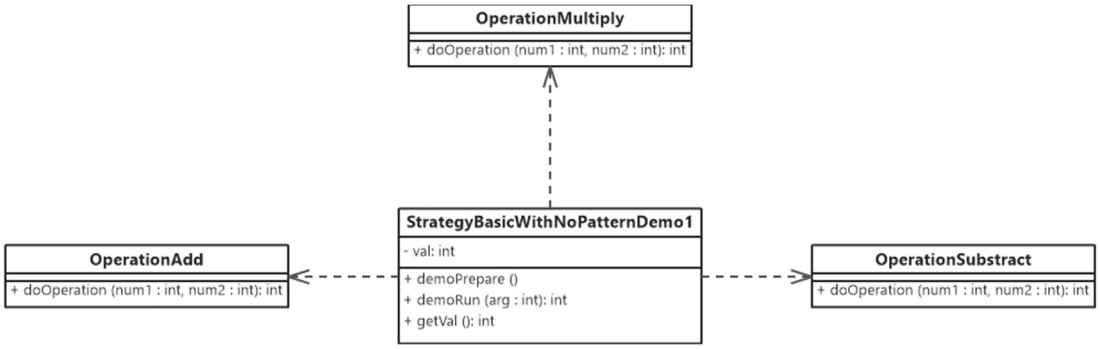
כעת נראה את אותו מימוש, כאשר האובייקטים מסוג Context וסוגי האלגוריתמים נוצרים פעם אחת בלבד בעליית מערכת. במקרה זה רק הקריאה לאלגוריתם (doOperation) מתבצע בזמן ריצה.



דיאגרמת מחלקות 19- Strategy SimpleAlgoWithPattern2

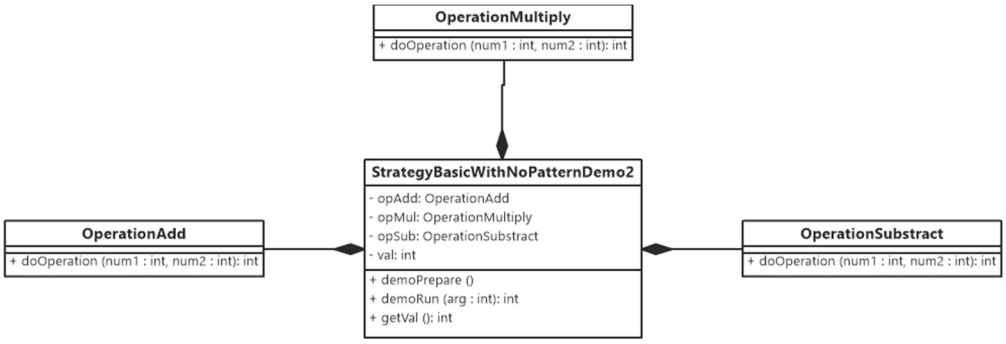
כאמור ההבדל בין הדוגמאות הינו בזמן יצירת האובייקטים. כך נוכל לבחון את עלות יצירת האובייקטים בזמן ריצה.

כעת נפשט את הדוגמא על ידי אי שימוש בתבנית העיצוב. כלומר ע״י ממשק ישיר בין ה-client לאלגוריתם.



דיאגרמת מחלקות 20- Strategy SimpleAlgoWithNoPattern1

גם כאן נשתמש בשתי דוגמאות. הראשונה יוצרת את כל האובייקטים בזמן ריצה ואילו השניה יוצרת את האובייקטים בעליית המערכת, ורק הקריאה לאלגוריתם (doOperation) מתבצעת בזמן ריצה.



דיאגרמת מחלקות 21 - Strategy SimpleAlgoWithNoPattern2

אמת מידה

מחשב מספר 1 :

Benchmark	Samples	Score	Score Error (99.9%)	Unit
StrategyAlgoWithNoPatternDemo	10	491.4739	4.770101	ns/op
StrategyAlgoWithPatternDemo	10	595.5737	12.486982	ns/op
StrategyBasicWithNoPatternDemo1	10	4.003997	0.005722	ns/op
StrategyBasicWithNoPatternDemo2	10	4.468942	0.045577	ns/op
StrategyBasicWithPatternDemo1	10	16.00978	0.064001	ns/op
StrategyBasicWithPatternDemo2	10	11.87502	0.045478	ns/op

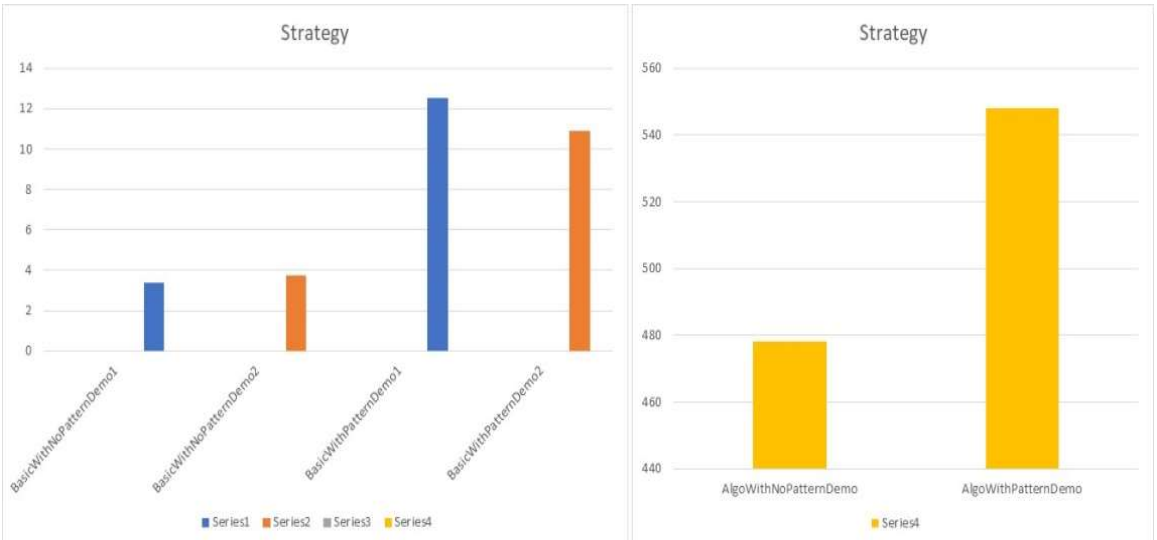
מחשב מספר 2 :

Benchmark	Samples	Score	Score Error (99.9%)	Unit
StrategyAlgoWithNoPatternDemo	10	464.53576	3.298724	ns/op
StrategyAlgoWithPatternDemo	10	500.477748	0.674091	ns/op
StrategyBasicWithNoPatternDemo1	10	2.734591	0.017615	ns/op
StrategyBasicWithNoPatternDemo2	10	3.007161	0.021789	ns/op
StrategyBasicWithPatternDemo1	10	9.048591	0.214296	ns/op
StrategyBasicWithPatternDemo2	10	9.884349	0.033836	ns/op

ניתוח התוצאות :

ניתן לראות בכל התוצאות ששימוש בתבנית עיצוב strategy עולה בביצועים. בדוגמא הראשונה העלות זניחה לעומת התועלת שתבנית עיצוב, אך בדוגמא השנייה הקפיצה די משמעותית. ההבדל נובע בגלל האלגוריתם אותו אנו מריצים. אם האלגוריתם מאוד פשוט - עלותה של תבנית העיצוב גבוהה, אך אם האלגוריתם מורכב - עלותה של תבנית העיצוב בטלה בשישים.

ההבדל בין היישומים Demo1 ו- Demo2 הינו כאמור במועד יצירת האובייקטים. ניתן לראות שיצירת אובייקט עולה זמן, לכן כדאי במידת האפשר לעשות זאת מחוץ לזמן הקריטי של ריצת התוכנית.



גרף 4 - סיכום ממוצע strategy

Adapter

Adapter שייכת לתבניות העיצוב ממשפחת structural. ניתן להשתמש בתבנית זו כשהקוד תלוי ב-API חיצוני או במחלקה חיצונית אחרת שנוטה להשתנות לעתים קרובות.

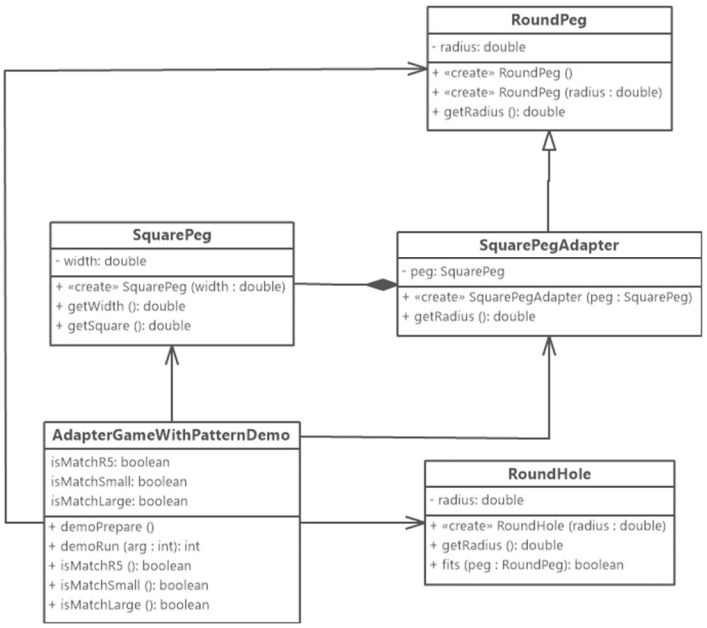
כדי לבחון באם תבנית העיצוב Adapter פוגעת בביצועים או לא, נבחן מספר דוגמאות.

Game

בדוגמא זו נבחן את יעילותו של האלגוריתם Adapter. לצורך כך ניצור שני יישומים, הראשון משתמש בתבנית עיצוב זו, והשני מיישם את אותה תוכנית ללא שימוש בתבנית העיצוב.

RoundHole הינה מחלקה חדשה אשר מספקת פונקציונליות fits. פונקציה זו מקבלת כקלט אובייקטים מסוג RoundPeg.

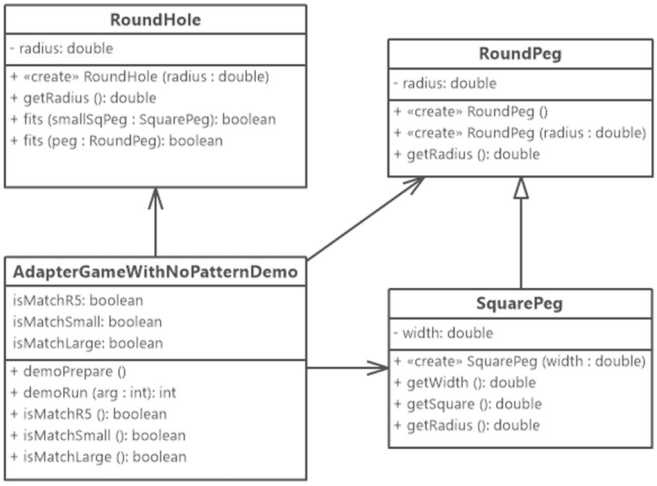
SequarePeg הינה מחלקה ישנה שאינה עונה לקריטריונים אלה, אך אנו זקוקים לה ביישום שלנו. נרצה שהיישום Demo יוכל לקבל את אותה פונקציונליות גם מהאובייקט הישן. לשם כך נשתמש בתבנית העיצוב adapter. כעת התוכנית שלנו ניגשת באמצעות adapter למחלקה הישנה.



דיאגרמת מחלקות 22 - Adapter GameWithPattern

ללא תבנית העיצוב היינו צריכים לפנות ישירות מ-SquarePeg ל-RoundPeg, ולדאוג לכך שפונקציית fits תדע לקבל אובייקט מסוג SquarePeg.

נקודת ההנחה היא שלא ניתן לשנות את האובייקט הישן.



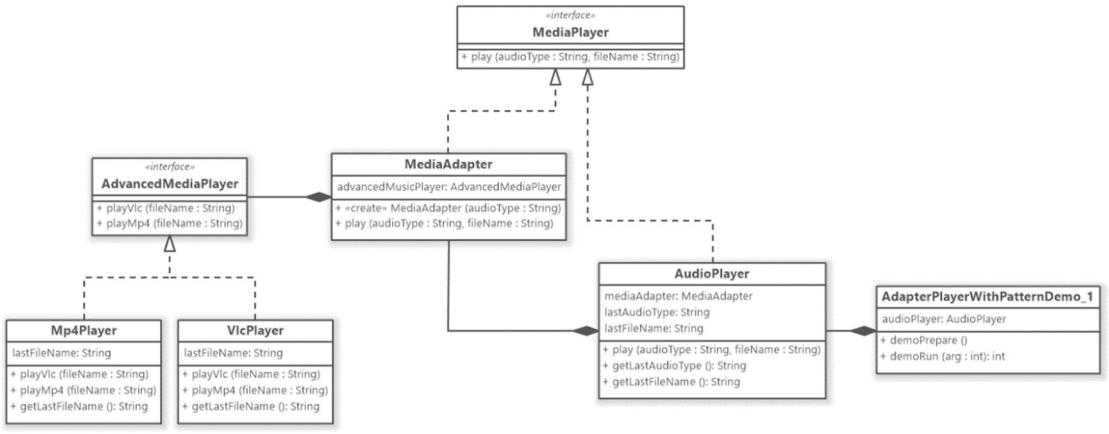
דיאגרמת מחלקות 23 - Adapter GameWithNoPattern

Player

כעת נראה דוגמא קלאסית ומוחשית יותר מעולם נגני המדיה.

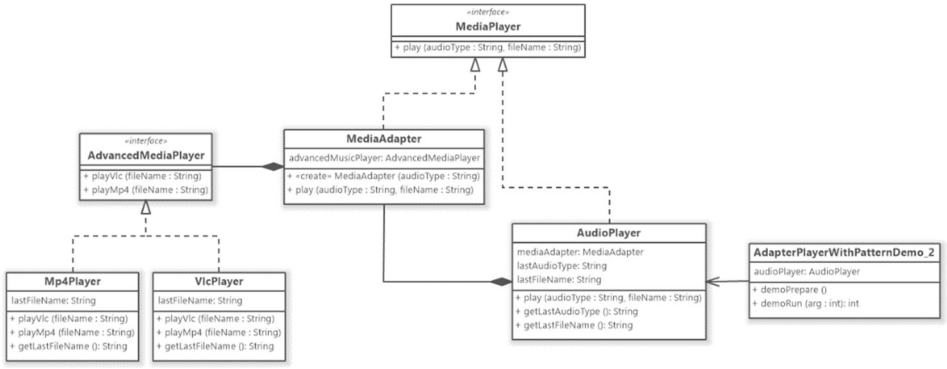
בדוגמא זו ניתן לראות שמחלקת AudioPlayer יודעת לתפעל מספר נגנים. יש לה יכולת מוטמעת של נגן mp3 ובנוסף באמצעות MediaAdater היא יכולה לתפעל עוד שני נגנים VLC ו-mp4.

מחלקת ה demo1 כלל לא מודעת להתרחשויות מאחורי הקלעים.



דיאגרמת מחלקות 24 - Adapter PlayerWithPattern1

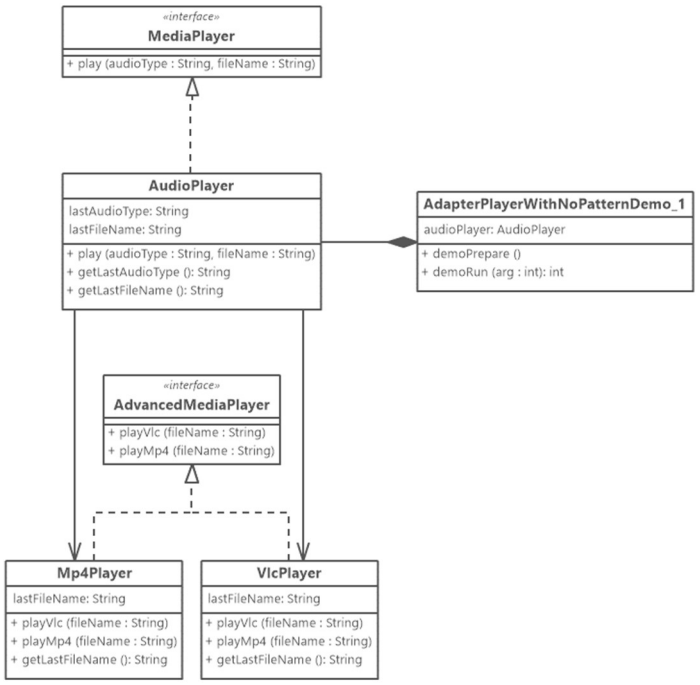
נבחן גם את demo2 אשר ההבדל היחידי בינה לבין demo1 הינו היחס ל-AudioAdapter.



דיאגרמת מחלקות 25 - Adapter PlayerWithPattern2

ב-demo2 בניגוד ל-demo1 תהליך יצירת כל האובייקטים מתבצע בזמן הריצה, כלומר ב-demoRun. ולהלן נרצה לבדוק כיצד משפיעה היצירה על זמני הריצה.

לשתי הדוגמאות הנ"ל ניצור יישומים אשר לא משתמשים Adapter. גם כאן ההבדל בין demo1 ל- demo2 הינו מועד יצירת האובייקטים.



דיאגרמת מחלקות 26 - Adapter PlayerWithNopattern1

אמת מידה

מחשב מספר 1 :

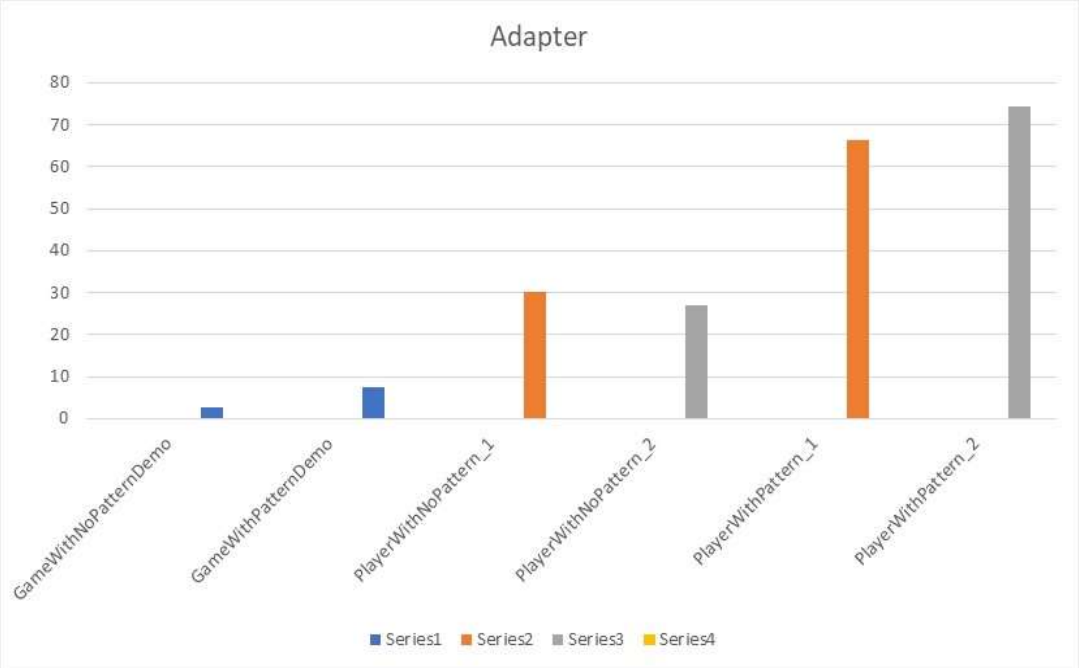
Benchmark	Samples	Score	Score Error (99.9%)	Unit
AadapterGameWithNoPatternDemo	10	3.032291	0.045506	ns/op
AdapterGameWithPatternDemo	10	7.666983	0.055294	ns/op
AdapterPlayerWithNoPattern_1	10	34.567581	0.535122	ns/op
AdapterPlayerWithNoPattern_2	10	31.678668	0.561797	ns/op
AdapterPlayerWithPattern_1	10	87.120311	1.597821	ns/op
AdapterPlayerWithPattern_2	10	87.471446	2.035368	ns/op

מחשב מספר 2 :

Benchmark	Samples	Score	Score Error (99.9%)	Unit
AadapterGameWithNoPatternDemo	10	2.521831	0.032868	ns/op
AdapterGameWithPatternDemo	10	7.334349	0.053111	ns/op
AdapterPlayerWithNoPattern_1	10	25.734676	0.361807	ns/op
AdapterPlayerWithNoPattern_2	10	22.402711	0.234978	ns/op
AdapterPlayerWithPattern_1	10	45.758062	0.405975	ns/op
AdapterPlayerWithPattern_2	10	61.342042	0.408552	ns/op

ניתוח התוצאות :

ברור מכל התוצאות והדוגמאות ששימוש בתבנית עיצוב adapter עולה בביצועים. כמו כן נראה שכדאי לקחת בחשבון את העלות המתלווה לשימוש בתבנית עיצוב זו, ואם מחליטים לעשות שימוש בתבנית העיצוב - כדאי שלא לבצע את יצירת האובייקטים בזמן ריצה.



גרף 5 - סיכום ממוצע Adapter

Prototype

תבנית עיצוב זו שייכת לתבניות עיצוב ממשפחת creational שנועדה למקרים בהם מעוניינים ליצור אובייקט כהעתק של אובייקט נתון, אשר טיפוסו ידוע רק בזמן ריצה.

לטובת בדיקת הביצועים של תבנית עיצוב זו נבחן שלוש דוגמאות.

Employee

בתבנית יישום זו כל מה שנבחן הינו שיכפול מבנה של אובייקט מסוג employee, וזאת בשני אופנים:

1. באמצעות יצירה מחדש.
2. באמצעות clone פשוט, כלומר shallow.

```
public class PrototypeWithPatternDemo{
	public void demoPrepare() {

	}

	public int demoRun(int arg) throws CloneNotSupportedException
	{
		for (int i=0; i < arg; i++)
		{
			Employees emps;
			emps = new Employees();
			emps.loadData();
			Employees empsNew1 = (Employees) emps.clone();
			List<String> list = emps.getEmpList();
			list.add("John");
			List<String> list1 = empsNew1.getEmpList();
			list1.remove("Pankaj");
		}
		return arg;
	}
```

קוד 5 - employeeWithPatternDemo - Prototype

הפעם נראה קוד:

אובייקט emps נוצר לפני בדיקת הביצועים, כלומר בפונקציית demoPrepare.

ניצור אובייקט חדש באמצעות clone. נעשה את אותה פעולה בדיוק - יצירת אובייקט חדש, אך ללא שימוש ב-clone אלא באמצעות הבנאי. ולבסוף נבחן את הביצועים.

חשוב לציין שדוגמא זו מייצגת שיכפול מלא של האובייקטים בפונקציית ה-clone. כלומר אובייקטים של מחלקת empoyees אינם משותפים בין שני האובייקטים (deep clone). בנוסף יש להדגיש שאנו לא משתמשים כאן ביכולת של java לבצע את ה-clone אלא אנו דורסים יכולת זו.

```
public class PrototypeWithNoPatternDemo{

	Employees emps;
	public void demoPrepare() {
		emps = new Employees();
	}
	public int demoRun(int arg)
	{
		for (int i=0; i < arg; i++)
		{
			Employees empsNew1 = new Employees();
			List<String> list = emps.getEmpList();
			list.add("John");
			List<String> list1 = empsNew1.getEmpList();
			list1.remove("Pankaj");
		}
		return arg;
	}
}
```

קוד 6 - Prototype EmployeeWithNoPatternDemo

שיכפול רגיל ללא clone:

```
public class Employees implements Cloneable{

	private List<String> empList;

	public Employees(){
		empList = new ArrayList<String>();
		loadData();
	}

	public Employees(List<String> list){
		this.empList=list;
	}
	private void loadData(){
		//read all employees from database and put into the list
		for (int i=0 ; i < 1; i++) {
			empList.add("Pankaj" + Integer.toString(i));
			empList.add("Raj"+ Integer.toString(i));
			empList.add("David"+ Integer.toString(i));
			empList.add("Lisa"+ Integer.toString(i));
		}
	}
	@Override
	public Object clone() throws CloneNotSupportedException{
			List<String> temp = new ArrayList<String>();
			for(String s : this.getEmpList()){
				temp.add(s);
			}
			return new Employees(temp);
	}
```

קוד 7 - Prototype EmpoyeeWithPattern

Department

נעבור לדוגמא נוספת, בעזרתה נדגים באמצעות קוד את אותה תוכנית עם ובלי תבנית עיצוב, עבור deep clone ו-shallow clone. הפעם נעזר ביכולת ה-clone של java.

חשוב לציין שאין הבדל בפונקציה המיישמת את יצירת האובייקטים, כלומר אין הבדל בקוד של Shallow ו-Deep במחלקת Demo. לכן נסתפק בדוגמא אחת.

```
public class PrototypeDepartmentDeepWithPatternDemo {

	Department dept1;
	Employee emp1;

	public void demoPrepare() {
		dept1 = new Department ("1", "A", "AVP");
		emp1 = new Employee (111, "John", dept1);
	}
	public int demoRun(int arg) throws CloneNotSupportedException
	{
		Employee emp2 = null;
		for (int i=0; i < arg; i++)
		{
			try {
			// Creating a clone of emp1 and assigning it to emp2
				emp2 = (Employee) emp1.clone();
			} catch (CloneNotSupportedException e) {
				e.printStackTrace();
			}
			emp2.dept.designation = "Director";
		}
		return arg;
	}
```

קוד 8 - Prototype DepartmentDeepWithPattern

```
public class PrototypeDepartmentDeepWithNoPatternDemo {
	Department dept1;
	Employee emp1;

	public void demoPrepare() {
		dept1 = new Department ("1", "A", "AVP");
		emp1 = new Employee (111, "John", dept1);
	}
	public int demoRun(int arg)
	{
		Employee emp2 = null;
		for (int i=0; i < arg; i++)
		{
			dept1 = new Department ("1", "A", "AVP");
			emp2 = new Employee (111, "John", dept1);
			emp2.dept.designation = "Director";
		}
		return arg;
	}
}
```

אמת מידה

מחשב מספר 1 :

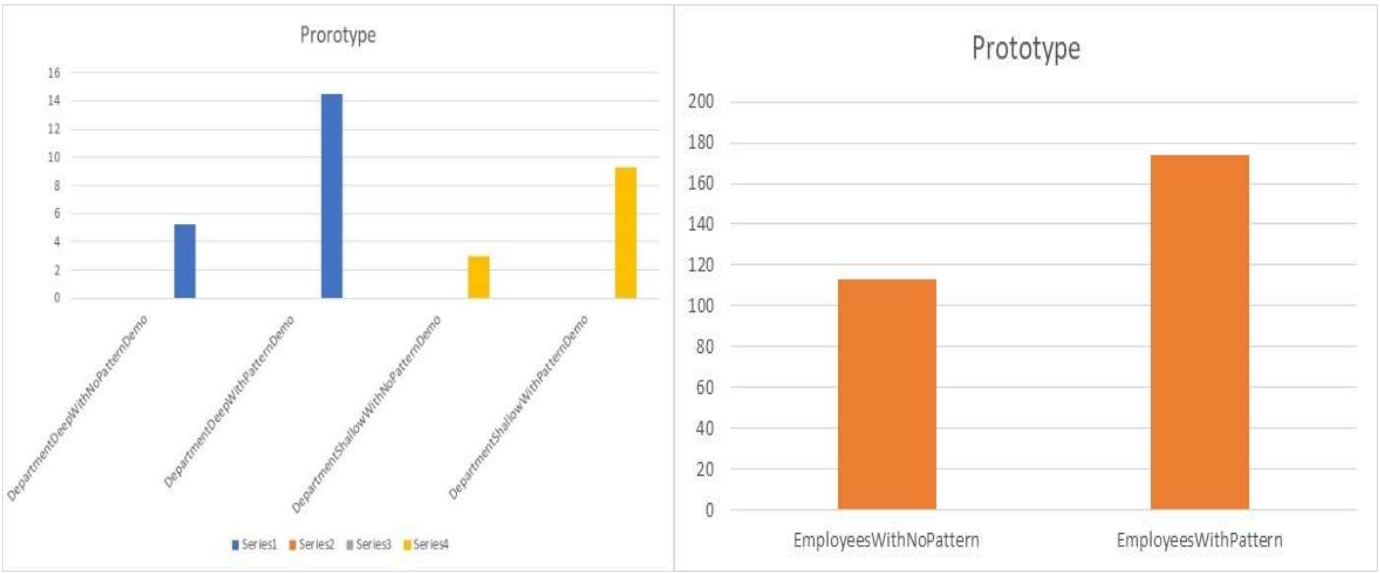
Benchmark	Samples	Score	Score Error (99.9%)	Unit
PrototypeDepartmentDeepWithNoPatternDemo	10	5.503294	0.01971	ns/op
PrototypeDepartmentDeepWithPatternDemo	10	13.811348	0.143677	ns/op
PrototypeDepartmentShallowWithNoPatternDemo	10	3.134209	0.070152	ns/op
PrototypeDepartmentShallowWithPatternDemo	10	10.288007	0.032716	ns/op
PrototypeEmployeesWithNoPattern	10	124.289961	1.108235	ns/op
PrototypeEmployeesWithPattern	10	169.533308	0.551619	ns/op

מחשב מספר 2 :

Benchmark	Samples	Score	Score Error (99.9%)	Unit
PrototypeDepartmentDeepWithNoPatternDemo	10	4.92363	0.047765	ns/op
PrototypeDepartmentDeepWithPatternDemo	10	15.24952	1.305976	ns/op
PrototypeDepartmentShallowWithNoPatternDemo	10	2.880845	0.025088	ns/op
PrototypeDepartmentShallowWithPatternDemo	10	8.276641	0.400288	ns/op
PrototypeEmployeesWithNoPattern	10	102.341446	0.548126	ns/op
PrototypeEmployeesWithPattern	10	178.777581	26.536093	ns/op

ניתוח התוצאות :

ניתן לראות שישנה עלות לתבנית העיצוב prototype. כמצופה, עלותו של deep clone גבוהה יותר מאשר זו של shallow clone. כמו כן, העלות היחסית זהה בין העתקה רדודה והעתקה עמוקה, וזאת ניתן לראות בברור בשתי הריצות על המחשבים השונים.



גרף 6 - סיכום ממוצע Prototype

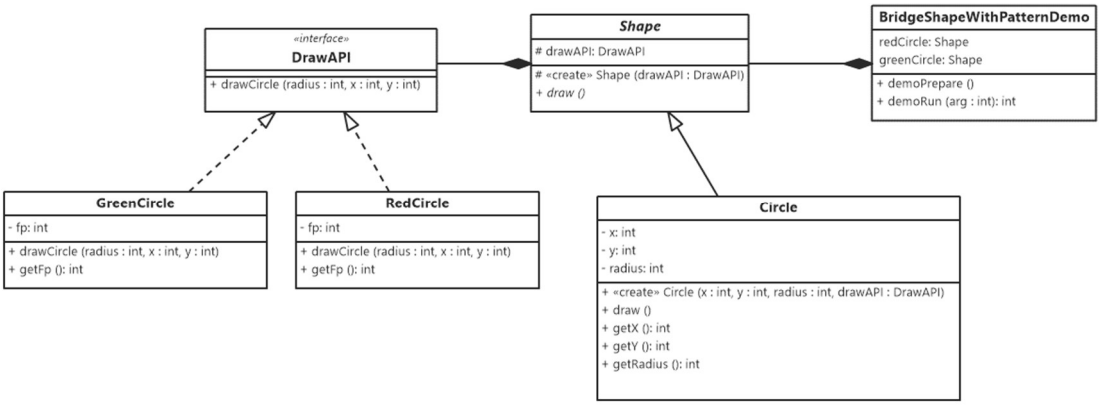
Bridge

תבנית עיצוב זו שייכת לקטגוריה Structural. באמצעות Bridge ניתן לנטרל את התלות הקיימת בין classes ששייכים להיררכיה מסוימת לבין ה-interfaces שהם מיישמים, וששייכים להיררכיה אחרת.

לטובת בדיקת הביצועים של תבנית העיצוב זה נבחן דוגמאות.

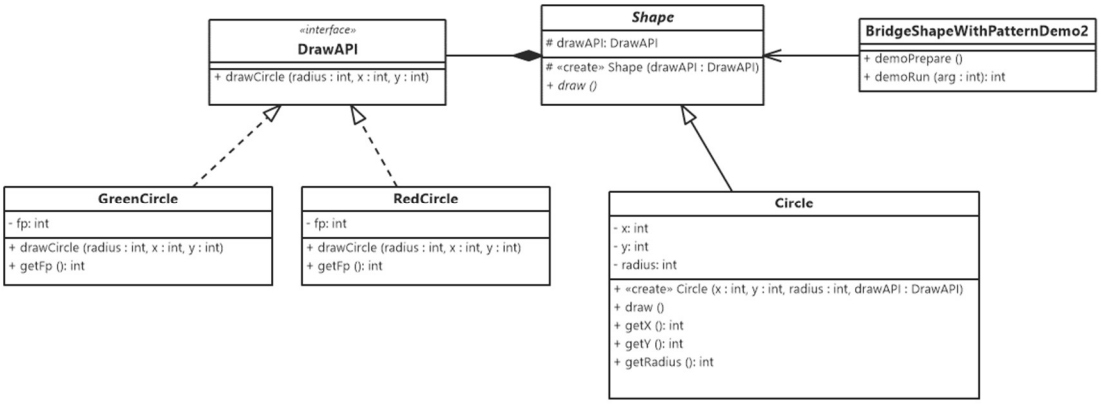
Shape

בדוגמא זו אנו ניצור צורה ונצבע אותה. הואיל וכל צורה ניתן לייצר בכל צבע, מספר האופציות האפשריות במימוש פשוט הינו מכפלה של כמות האפשרויות. באמצעות תבנית עיצוב זו נעשה פיצול בין אובייקט צבע לאובייקט צורה.



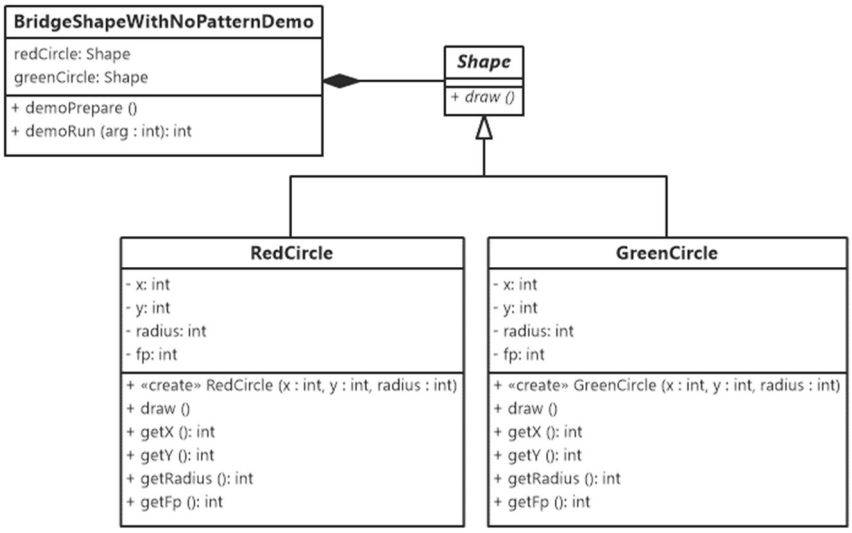
דיאגרמת מחלקות 27 Bridge ShapwWithPattern1

ההבדל בין שתי הדוגמאות הוא בזמן יצירת אובייקט shape. בדוגמא הראשונה ניצור את האובייקטים בזמן init, כלומר באתחול. ואילו בדוגמא השנייה ניצור את האובייקט תוך כדי ריצה. פעולת drawCircle מתבצעת בזמן ריצה בשני המקרים.



דיאגרמת מחלקות 28- Bridge ShapeWithPattern2

כעת, נראה את אותה פונקציונליות ללא תבנית עיצוב. כלומר נשלב את יצירת האובייקט עם יצירת צבע לאובייקט אחד.

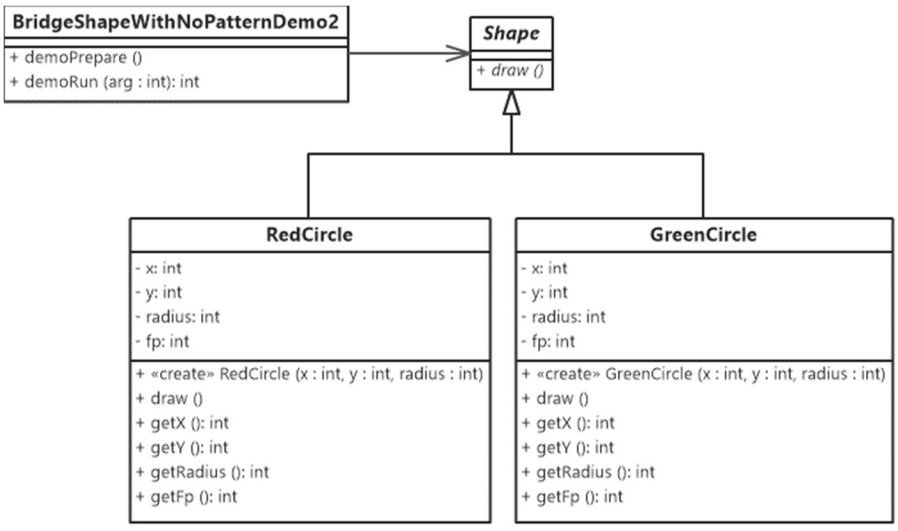


דיאגרמת מחלקות 29 - Bridge ShapeWithNoPattern1

כאשר מורידים את תבנית העיצוב, אותו אובייקט אחראי על יצירת הצורה והצבע.

גם כאן, ההבדל היחיד בין שני הדוגמאות זה מועד יצירת האובייקטים. בדוגמא הראשונה האובייקטים נוצרים בזמן האיתחול ובדוגמא השנייה בזמן הריצה.

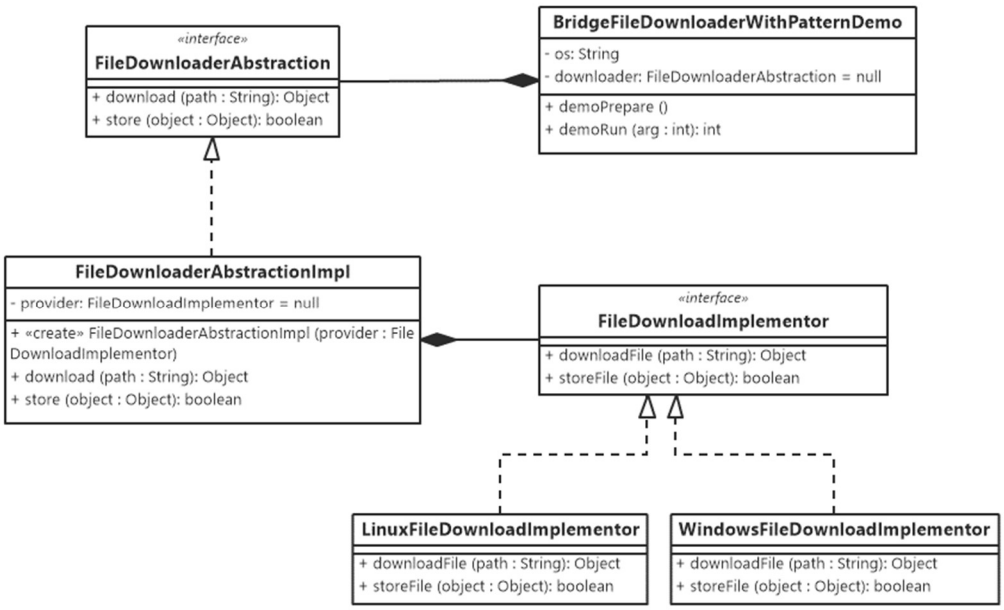
לצורך ההשוואה חשוב שנשווה כל אפשרות בנפרד.



דיאגרמת מחלקות 30 - Bridge ShapeWithNoPattern2

File downloader

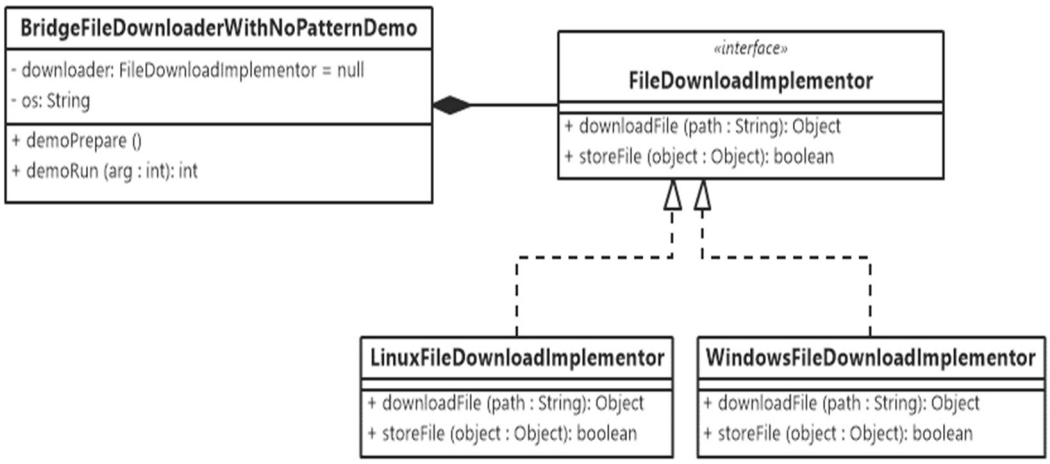
בדוגמא זו נראה bridge בין שני אובייקטים. האובייקט הראשון אחראי על הורדה ושמירת הקבצים. והאובייקט השני אחראי על בחירת מערכת הפעלה מתאימה עם נתיב לקבצים ואופן קריאה ושמירת הקבצים.



דיאגרמת מחלקות 31 - Bridge FileDownloaderWithPattern

נציג שתי דוגמאות - בראשונה שמירה של המשתנים במחלקה, ובשנייה שמירה של המשתנים בפונקציה היוצרת.

כאשר לא נשתמש בתבניות עיצוב, אותה מחלקה אחראית על שתי הפונקציונליות.



דיאגרמת מחלקות 32 - Bridge FileDownloaderWithNoPattern

אמת מידה

מחשב מספר 1:

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
BridgeFileDownloaderWithNoPatternDemo	10	16.103647	0.072878	2
BridgeFileDownloaderWithNoPatternDemo2	10	16.118695	0.077628	2
BridgeFileDownloaderWithPatternDemo	10	24.040437	0.321433	2
BridgeFileDownloaderWithPatternDemo2	10	20.53958	0.483309	2
BridgeShapeWithNoPatternDemo	10	4.109481	0.018541	2
BridgeShapeWithNoPatternDemo2	10	2.693864	0.009063	2
BridgeShapeWithPatternDemo	10	4.340204	0.057461	2
BridgeShapeWithPatternDemo2	10	5.776019	0.154885	2

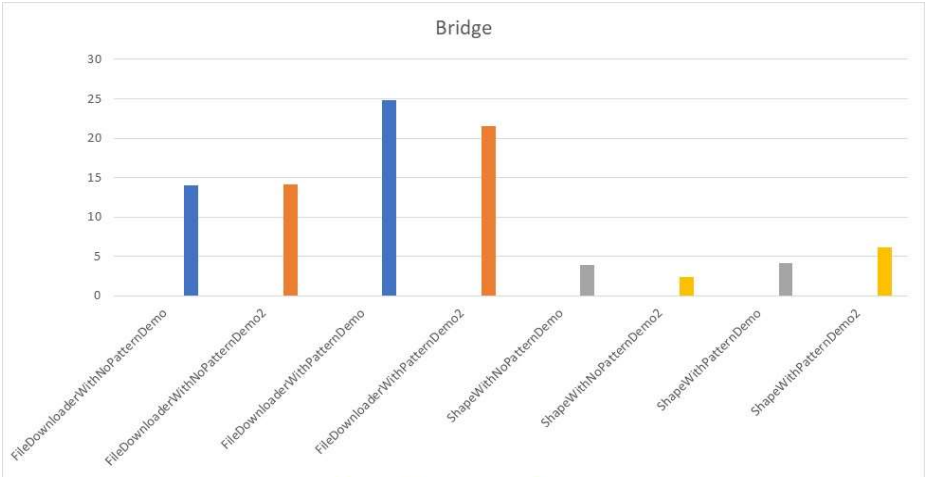
מחשב מספר 2:

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
BridgeFileDownloaderWithNoPatternDemo	10	11.989213	0.029732	2
BridgeFileDownloaderWithNoPatternDemo2	10	12.035578	0.094441	2
BridgeFileDownloaderWithPatternDemo	10	25.639563	6.084821	2
BridgeFileDownloaderWithPatternDemo2	10	22.485891	0.044099	2
BridgeShapeWithNoPatternDemo	10	3.668069	0.004051	2
BridgeShapeWithNoPatternDemo2	10	2.157869	0.004382	2
BridgeShapeWithPatternDemo	10	3.859886	0.003319	2
BridgeShapeWithPatternDemo2	10	6.567419	1.337945	2

ניתוח התוצאות:

ניתן לראות שישנה עלות כאשר משתמשים בתבנית עיצוב bridge. הדבר המעניין הוא שכאשר משתמשים בפונקציונליות, כלומר ללא יצירת האובייקטים, העלות בשימוש בתבנית עיצוב זו זניחה. ניתן לראות זאת בברור בדוגמא - BridgeShapeWithNoPatternDemo לעומת BridgeShapeWithPatternDemo.

כמסקנה ניתן לומר שבמידה ואפשר להמעיט ביצירה של אובייקטים בקטע קריטי של ריצת המערכת, תבנית זו יכולה להיות שימושית מאוד.



גרף 7 - סיכום ממוצע Bridge

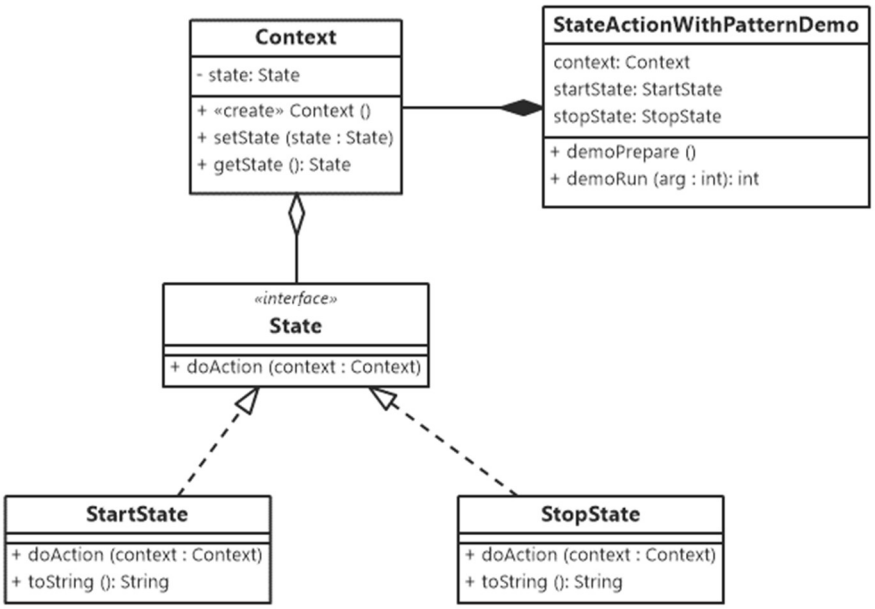
State

תבנית עיצוב זו השייכת לתבניות עיצוב מסוג Behavioral, מאפשרת לאובייקט לשנות את התנהגותו כאשר מצבו הפנימי משתנה, ע"י שינוי המחלקה שבה הוא משתמש. תבנית עיצוב זו הינה אחת הפשוטות למימוש, והיא שימושית ביותר. נשתמש בתבנית כאשר התנהגות האובייקט תלויה במצבו, והוא חייב לשנותה בזמן ריצה.

הדוגמא שלהלן תמחיש לנו את ההבדלים בין מימוש עם ובלי תבנית עיצוב.

Action

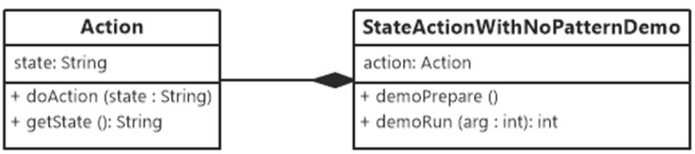
בדוגמא זו נרצה להראות פעולה של אובייקט בהתאם למצבו. כמו כן נראה שמחלקת demo מחזיקה אובייקטים ומפעילה אותם בהתאם למצב הרצוי. במקרה שלנו, העברה בין מצב start ו-stop כאשר ה-context מחזיק את המצב.



דיאגרמת מחלקות 33 - State ActionWithPattern

כאשר נרצה שלא להשתמש בתבנית העיצוב, נצטרך ליישם את כל פונקציונליות של מעבר בין המצבים באותה מחלקה. לכן נקרא למחלקה החדשה Action, והיא תיישם באמצעות if-else את המעברים הרצויים.

ללא תבנית עיצוב, היישום יראה כך:



דיאגרמת מחלקות 34 - State ActionWithNoPattern

אמת מידה

מחשב מספר 1 :

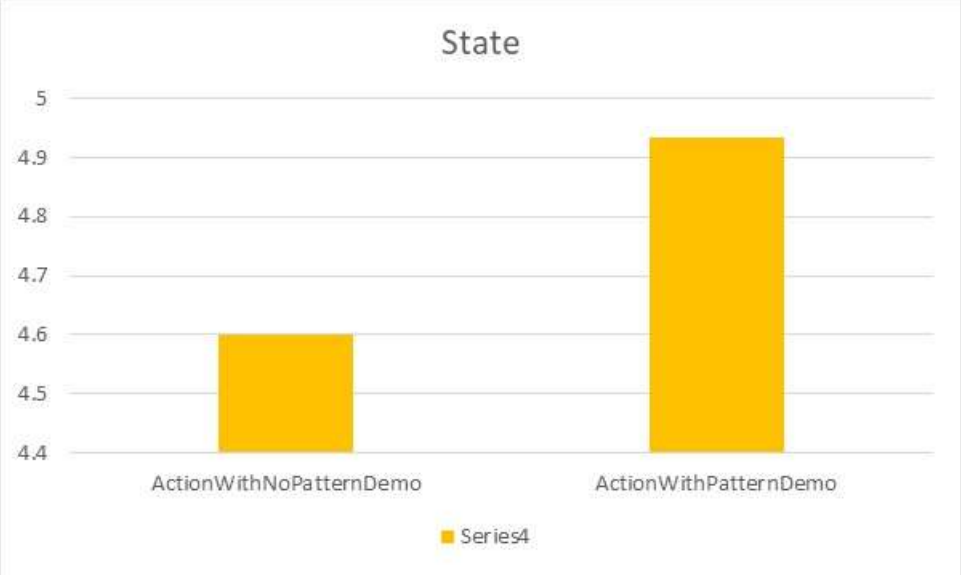
Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
StateActionWithNoPatternDemo	10	5.387879	0.026288	10
StateActionWithPatternDemo	10	5.423527	0.094535	10

מחשב מספר 2 :

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
StateActionWithNoPatternDemo	10	3.811982	0.028121	10
StateActionWithPatternDemo	10	4.442211	0.027971	10

ניתוח התוצאות :

מהניתוח תוצאות רואים בבירור שישנה הרעה מינימלית כאשר משתמשים בתבנית העיצוב State. תבנית עיצוב זו באה לתת פתרון לשינויי מצבים של האובייקט, לכן לפי הניתוח עולה שהעלות בזמן ריצה מצדיקה את הסדר אשר תבנית עיצוב זו נותנת.



גרף 8 - סיכום ממוצע State

Iterator

תבנית עיצוב זו שייכת לתבניות עיצוב מסוג Behavioral.

איטרטור הוא הפשטה של מעבר בסדר מוגדר מראש על מבנה נתונים כלשהו. כדי לבצע פעולה ישירה על מבנה הנתונים, צריך לדעת כיצד הוא מיוצג. אך שימוש באיטרטור מאפשר זאת ללא הכרה מוקדמת של מבנה הנתונים.

כדי לבחון את יעילותו של האיטרטור נביא כמה דוגמאות. בחלקן ניישם את האיטרטור בעצמנו ובחלקן נשתמש באיטרטור של השפה.

```
public class IteratorArrayListWithNoPattern implements DemoFunctions{
	private ArrayList<Integer> numbers;
	public void demoPrepare() {
		numbers = new ArrayList<Integer>();
	}
	public int demoRun(int arg) {
		for (int i = 0; i < arg; i++)
		{
			numbers.add(i);
		}
		for (int i = 0; i < arg; i++)
		{
			if(!numbers.isEmpty())
				numbers.remove(0);
		}
		return arg;
	}
}
```

קוד 10 - Iterator ArrayWithNoPattern

```
public class IteratorArrayListWithPattern implements DemoFunctions {
	private ArrayList<Integer> numbers;
	public void demoPrepare() {
		numbers = new ArrayList<Integer>();

	}
	public int demoRun(int arg) {
		for (int i = 0; i < arg; i++)
		{
				numbers.add(i);
		}
		Iterator<Integer> it = numbers.iterator();
		while(it.hasNext()) {
			Integer i = it.next();
			if (i<arg)
			  it.remove();
		}
		return arg;
	}
}
```

קוד 11 - Iterator ArrayWithPattern

בשתי הדוגמאות אנו יוצרים בפונקציית demoPrepare מערך (Array), מוסיפים לו ארגומנטים ולאחר מכן מורידים ממנו ארגומנטים. נבצע את אותה פונקציונליות גם בעזרת vector במקום array, כלומר נשתמש ב- Vector<Integer> במקום ב- ArrayList<Integer> .

בדוגמא נוספת ניצור ArrayList<Employee> ונשתמש באיטרטור כדי לאכסן ולקרא אובייקטים. כאן נרצה לראות באם יש הבדל בין ריצת האיטרטור על אובייקטים מסוג ידוע (integer) לבין ריצתו על אובייקטים מסוג לא ידוע (employee).

אמת מידה

מחשב מספר 1:

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
IteratorEmployeesWithNoPattern	20	5878.914019	73.643602	10
IteratorEmployeesWithPattern	20	4537.471863	74.936712	10
IteratorIteratorArrayListWithNoPattern	20	137.129627	1.6765	10
IteratorIteratorArrayListWithPattern	20	132.813047	1.903217	10
IteratorVectorWithNoPattern	20	731.427522	9.84362	10
IteratorVectorWithPattern	20	880.293131	8.638476	10

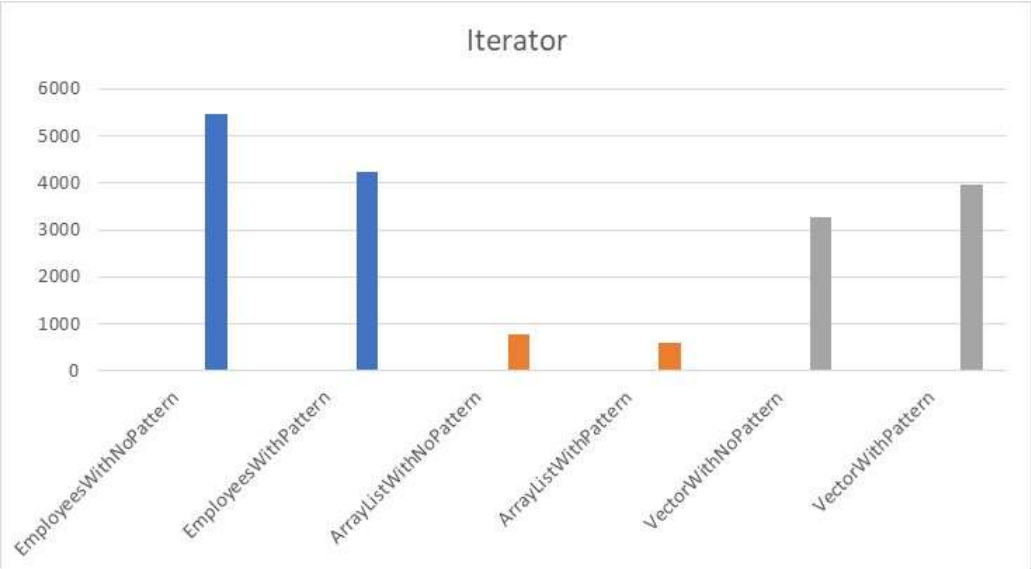
מחשב מספר 2:

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
IteratorEmployeesWithNoPattern	10	5066.432538	225.036541	100
IteratorEmployeesWithPattern	10	3915.072462	26.319769	100
IteratorIteratorArrayListWithNoPattern	10	1391.476652	156.09756	100
IteratorIteratorArrayListWithPattern	10	1090.72459	4.868505	100
IteratorVectorWithNoPattern	10	5828.490394	29.387842	100
IteratorVectorWithPattern	10	7038.753458	11.334661	100

ניתוח התוצאות:

לטובת ניתוח התוצאות של iterator הרצנו את היישום על שני המחשבים מספר פעמים, ובכולן קיבלנו באופן מובהק שכדאי להשתמש באיטרטור במקרים של ArrayList ופחות כדאי כאשר אנו משתמשים ב-Vector.

ניתן גם לראות שהשימוש באיטרטור כאשר משתמשים ב Vector אינו עולה הרבה יותר מהפתרון ללא תבנית עיצוב. כמו כן, הטמעת תבנית עיצוב זו בשפת java משפרת את ביצועים.



גרף 9 - סיכום ממוצע iterator

סיכום ומסקנות

סקרנו את יעילותן של מספר תבניות עיצוב המופיעות בספר GoF [3]. בחרנו לחקר תבניות נפוצות בתעשיה, משלושה תחומים (יצירה, מבנה והתנהגות).

הראינו שישנן תבניות עיצוב מאוד יעילות בביצועים, ואף כאלו שעולות בביצועיהן על אותו יישום ללא תבנית עיצוב. יש תבניות עיצוב אשר יעילותן פחותה מעט מאותו יישום ללא תבנית העיצוב, אך הקוד הנקי אשר הן מקנות מפצה על הירידה הקלה בביצועים. וישנן תבניות עיצוב אשר יעילותן פחותה בהרבה מאותו יישום ללא תבנית עיצוב, ויש לשקול היטב אם נכון וכדאי להשתמש בהן.

הטבלה שלהלן מציגה בתמצות את המסקנות אליהן הגענו עבור כל אחת מהתבניות.

Design Pattern	סיכום
Builder	יעילות פחותה מעט בהשוואה ליישום ללא תבנית עיצוב (לריבוי בנאים), אך מקנה יתרון גדול בקריאות ותחזוקה של הקוד.
Observer	יעילות טובה כאשר רוצים לשלוח הודעות קצרות. יעילות פחות טובה כאשר ההודעה מורכבת וארוכה.
Visitor	תבנית עיצוב יעילה.
Strategy	יעילות פחותה בהשוואה לשימוש ללא תבנית עיצוב, אך העלות זניחה כאשר לוקחים בחשבון את הסדר שתבנית עיצוב זו מקנה. העלות מול תועלת משתפרת עבור אלגוריתמים מורכבים.
Adapter	עלות גבוהה בביצועים. כדאי לשקול היטב אם יש צורך ממשי להשתמש בתבנית עיצוב זו.
Prototype	עלות גבוהה בביצועים. כדאי לשקול היטב אם יש צורך ממשי להשתמש בתבנית עיצוב זו.
Bridge	עלות גבוהה בביצועים בעת יצירת אובייקטים. כדאי לשקול היטב אם יש צורך ממשי להשתמש בתבנית עיצוב זו. אם משתמשים - כדאי לייצר את האובייקטים לא בזמן קריטי.
State	יעילות פחותה מעט בהשוואה ליישום ללא תבנית עיצוב, אך העלות זניחה כאשר לוקחים בחשבון את הסדר שתבנית עיצוב זו מקנה.
Iterator	תבנית עיצוב יעילה.

תבניות עיצוב מקנות קריאות ותחזוקתיות, ואף משפרות את זמני פיתוח הקוד. כל תכנת צריך לקחת בחשבון את ה'עלות מול תועלת' שתבנית העיצוב מקנה ליישום אותו הוא מממש, ולהימנע במידת האפשר משימוש בתבניות עיצוב 'זוללות' זמן CPU (כלומר להימנע מ"הכנות למזגן").
בנוסף, מאוד חשוב היכן וכיצד מאוחסנים האובייקטים בהם משתמשים בזמן ריצה.
המסקנות אליהן הגענו הינן אותן המסקנות המופיעות ב- [4], כלהלן:

- “Using design patterns is mainly advocated because of their positive effect on the software structure and their potential to speed up the design process and improve its quality.”
- “A standard solution may have positive or negative performance effects depending on the characteristics of the application area and the intended workload. The difference between a “good” pattern and a “bad” one may be in the parameters of the system, and storing this type of information in the pattern library can aid the designer in predicting the performance of the software.”

<u>דגשים</u>:

1. ניתן להשתמש בחלק ראשון של העבודה לביצוע אמת מידה עבור כל פרויקט JAVA שהוא.
2. ניתן להוסיף לחלק השני של העבודה מספר רב של תבניות עיצוב. אנו התמקדנו בתבניות עיצוב GoF, אך אין מניעה מלהשתמש בתבניות עיצוב מסוג אחר.
3. בנספח א׳ היצגנו תוצאות כאשר מגבילים את הזיכרון של ה- JVM ל- 16M. הניתוח שנעשה עבור זיכרון של 2G נכון גם עבור זיכרון הרבה יותר קטן. מכאן ניתן להסיק שגודל הזיכרון אינו משפיע על הבדיקות שנעשו.

References

[1] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Boston: Prentice Hall, 2018.

[2] C. Alexander, The Timeless Way of Building, California: Center for Environmental structure, 1970.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley, 1994.

[4] V. Inkeri, J. Gustafsson, L. Nenonen and J. Paakki, "Design patterns in performance prediction," vol. 2000, pp. 143-144, 2000.

[5] S. Kamil, J. Shalf and E. Strohmaier, "Power efficiency in high performance computing," *EEE International Symposium on Parallel and Distributed Processing,* pp. 1-8, 2008.

[6] A. Carroll and H. Gernot , "An analysis of power consumption in a smartphone," *USENIX annual technical conference,* vol. 14, p. 21, 2010.

[7] E. Freeman, E. Robson, B. Bates and K. Sierra, Head First Design Patterns, Culemborg, Netherlands: Van Duuren Media, 2008.

[8] F. Buschmann and D. C. Schmidt, Pattern oriented software architecture, Chichester: Wiley, 2011.

[9] V. Sarcar, A Hands-On Experience with Real-World Examples, Apress; 2nd editionau, 2018.

נספח א'

תוצאות ריצה: Xmx16m, Xms16m

Builder

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
StudentBuilderDemoWithNoPattern	10	30.120804	0.185435	1
StudentBuilderDemoWithPattern	10	32.67859	0.23732	1
UserBuilderDemoWithNoPattern1	10	52.808549	0.755455	1
UserBuilderDemoWithNoPattern2	10	53.940639	0.642685	1
UserBuilderDemoWithPattern	10	78.373934	0.793491	1

Observer

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
ObserverCalcWithNoPattern	10	93.929839	0.670419	1
ObserverCalcWithPatternDemo	10	112.076309	3.207111	1
ObserverMessageWithNoPattern	10	7.26804	0.118423	1
ObserverMessageWithPatternDemo	10	23.824975	0.417082	1

Visitor

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
VisitorShoppingCartClientWithNoPattern1	10	6.925998	0.02132	1
VisitorShoppingCartClientWithNoPattern2	10	6.020379	0.040484	1
VisitorShoppingCartClientWithPattern	10	8.039587	0.048929	1
VisitorTaxWithNoPattern	10	4.984029	0.025366	1
VisitorTaxwithPattern1	10	5.519906	0.422597	1
VisitorTaxwithPattern2	10	5.820035	0.131826	1

Strategy

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
StrategyAlgoWithNoPatternDemo	10	695.471275	4.934688	1
StrategyAlgoWithPatternDemo	10	738.140344	4.2736	1
StrategyBasicWithNoPatternDemo1	10	2.562645	0.021527	1
StrategyBasicWithNoPatternDemo2	10	2.821064	0.010145	1
StrategyBasicWithPatternDemo1	10	21.908887	0.258094	1
StrategyBasicWithPatternDemo2	10	9.97238	0.47575	1

Adapter

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
AadapterGameWithNoPatternDemo	10	2.300993	0.011849	1
AdapterGameWithPatternDemo	10	10.095848	0.053376	1
AdapterPlayerWithNoPattern_1	10	23.692933	0.064757	1
AdapterPlayerWithNoPattern_2	10	20.887526	0.225774	1
AdapterPlayerWithPattern_1	10	58.106862	0.60464	1
AdapterPlayerWithPattern_2	10	53.457159	0.592241	1

Prototype

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
PrototypeDepartmentDeepWithNoPatternDemo	10	7.009444	0.158803	1
PrototypeDepartmentDeepWithPatternDemo	10	18.851966	0.338659	1
PrototypeDepartmentShallowWithNoPatternDemo	10	2.679542	0.0043	1
PrototypeDepartmentShallowWithPatternDemo	10	12.410764	0.610901	1
PrototypeEmployeesWithNoPattern	10	154.452812	1.761076	1
PrototypeEmployeesWithPattern	10	235.653878	3.212793	1

Bridge

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
BridgeFileDownloaderWithNoPatternDemo	10	23.532199	0.228059	2
BridgeFileDownloaderWithNoPatternDemo2	10	18.596684	0.424194	2
BridgeFileDownloaderWithPatternDemo	10	32.35025	1.011397	2
BridgeFileDownloaderWithPatternDemo2	10	23.409096	0.099976	2
BridgeShapeWithNoPatternDemo	10	3.512642	0.019604	2
BridgeShapeWithNoPatternDemo2	10	2.07609	0.008843	2
BridgeShapeWithPatternDemo	10	3.761247	0.253477	2
BridgeShapeWithPatternDemo2	10	7.436565	0.067973	2

State

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
StateActionWithNoPatternDemo	10	3.863548	0.03139	10
StateActionWithPatternDemo	10	4.551018	0.072072	10

Iterator

Benchmark	Samples	Score	Score Error (99.9%)	Param: arg
IteratorEmployeesWithNoPattern	10	4973.962948	41.224576	10
IteratorEmployeesWithPattern	10	4491.25014	42.686361	10
IteratorIteratorArrayListWithNoPattern	10	120.436456	1.166211	10
IteratorIteratorArrayListWithPattern	10	102.325206	0.516977	10
IteratorVectorWithNoPattern	10	583.629615	1.188583	10
IteratorVectorWithPattern	10	726.477081	1.554294	10