Design-Patterns

	שימושים	עקרונות עיצוביים	מימוש	דוגמה

Factory	יצירה של אובייקטים החולקים עץ ירושה (מממשים את אותו הממשק או יורשים מאותו אב קדום)	• **Single-Responsibility** בכך שיוצרים הפרדה בין היצירה של האובייקטים לבין שאר הקוד
		• **Open-Closed** מכיוון שאפשר להוסיף למפעל עוד סוגים של אובייקטים מבלי לפגוע בשאר הקוד.
			מחלקה או מתודה, רגילה או סטטית, שמקבלת קלט מסוים ומחזירה בהתאם אליו אובייקט מתאים.	`public abstract class Worker{...}`
				`public class Musician extends Worker {...}`
				`public class Carpenter extends Worker {...}`
				`public class Unemployed extends Worker {...}`
				
				`public class Factory{`
				
				`public static Worker getWorker(String jobName){`
				`Worker worker;`
				`switch (jobName){`
				`case "Carpenter":`
				`worker = new Carpenter();`
				`break;`
				`case "Musician":`
				`worker = new Musician();`
				`break;`
				`default:`
				`worker = new Unemployed();`
				`}`
				
				`return worker;`
				`}`
				`}`

Singleton	הגבלה על מספר המופעים שאפשר ליצור ממחלקה, כך שיהיה קיים מופע אחד בלבד.		נשמור שדה סטטי שמחזיק את המופע היחיד של המחלקה ומתודה סטטית שמאפשרת לגשת אליו, ונגדיר את הבנאי להיות פרטי כדי למנוע יצירה של מופעים נוספים. אפשר ליצור את המופע היחיד מלכתחילה או ברגע שקוראים לו בפעם הראשונה.	`public class SingleObject{`
				`private static SingleObject singleton = new SingleObject();`
				
				`private SingleObject() {}`
				`public static SingleObject getInstance(){`
				`return SingleObject.singleton;`
				`}`
				`}`

Delegation	מימוש API של מחלקה אחרת או שימוש בפונקציונליות של מחלקה אחרת שלא באמצעות ירושה	• **חיזוק האנקפסולציה:** המימוש הפנימי של המחלקה אינו חשוף
		• **Decomposability, Composability**: מונע מחזור קוד ומאפשר לאגד יחידות נפרדות.
			נחזיק שדה מטיפוס המחלקה ונאציל אליו את הפעולות המתאימות	`public class A{`
				`  void boo(){...}`
				`}`
				
				`public class C{...}`
				
				`public class B extends C{`
				`  private A a = new A();`
				
				`  void boo(){`
				`  this.a.boo();`
				`  }`
				`}`

Decorator	מימוש ה-API של מחלקה קיימת תוך הוספת פונקציונליות (משל הוספת הרוטב לפיצה או ההרחבות לסימס וקטאן)	• **Single Responsiblity** – כל הרחבה בפני עצמה – מתחבר גם לרעיון של פירוק ליחידות קטנות.
		• **חיזוק האנקפסולציה** (היכולת להסתכל על פיצה עם רוטב קודם כל בתור פיצה או "אתה לא יכול לשחק בהרחבה של סימס בלי המשחק המקורי")
			להרחיב את המחלקה שרוצים לקשט (ע"י מימוש אותו ממשק או ירושה מאותה מחלקה קדומה) + להחזיק מופע של האובייקט המקושט בתור שדה (לרוב מקבלים אותו בקונסטרקטור)	`public class Sims2 {`
				`  public void Dress{`
				`  this.wearCasual();`
				`  }`
				`}`
				`public class Sims2BSeasons extends Sims2 {`
				`  Sims2 sims2;`
				`  public Sims2BSeasons(Sims2 sims2){`
				`  this.sims2=sims2;`
				`  }`
				`  public void Dress(){`
				`  this.wearBySeason();`
				`  }`
				`}`

Façade	כשנרצה לחשוף רק חלק מהAPI, בשל מורכבות או אי רצון בחשיפת מידע	• **API מינימלי** – חושפים רק חלק מהמידע. • **אנקפסולציה** – אפשר לחשוף רק חלקים מסויימים מה-API של המחלקה וכך להסתיר חלקים מהמימוש	כתיבה של מחלקה חדשה שמנתבת פעולות למחלקות הקיימות.	

```java
/* Complex parts */
class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}
class Memory {
    public void load(long position, byte[] data) { ... }
}
class HardDrive {
    public byte[] read(long lba, int size) { ... }
}

/* Facade */
class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR,
SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}
/* Client */
class You {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start();
    }
```

Strategy	כשנרצה לאפשר מספר התנהגויות ולהחליף ביניהן בזמן הריצה	• **Open-Closed** – ניתן להוסיף אסטרטגיות נוספות מבלי לשנות את הקיימות.
		• **Single-Choice** – הבחירה באסטרטגיה מתבצעת במקום אחד.
		• **פירוק לחלקים קטנים** – כל אסטרטגיה מתנהלת באופן עצמאי.
			ניצור מחלקות שונות בהתאם לאסטרטגיות, שכולן מממשות את אותו ממשק. באמצעות Factory נבצע את הבחירה של כל אחד מהאסטרטגיות, לפי תנאי מסוים.	`public interface transportationToAirPort{`
				`void drive();`
				`}`
				`public class AirportShuttle implements transportationToAirPort{...}`
				`public class CityBus implements transportationToAirPort{...}`
				`public class Helicopter implements transportationToAirPort{...}`
				`public class Subway implements transportationToAirPort{...}`
				`public class Taxi implements transportationToAirPort{...}`
				`public class Strategy{`
				`public static transportationToAirPort select(){...}`
				`}`