DESIGN PATTERNS IN ARABIC

USING JAVA

Eng. Sami Kazah

الفهرس

مقدمة إلى Design Patterns ---------- ٣

Creational Patterns

- Singleton ---------- ٦
- Factory ---------- ٨
- Builder ---------- ١٢
- Prototype ---------- ١٦

Structural Patterns

- Adapter ---------- ١٩
- Proxy ---------- ٢٢
- Composite ---------- ٢٥
- Decorator ---------- ٢٨
- Bridge ---------- ٣١
- Fly Weight ---------- ٣٣
- Facad ---------- ٣٧

Behavioral Patterns

- Visitor ---------- ٣٩
- Observer ---------- ٤٢
- Chain of responsibility ---------- ٤٥
- Template ---------- ٤٩
- Strategy ---------- ٥١
- Command ---------- ٥٥

مقدمة إلى أنظمة الزمن الحقيقي ---------- ٥٩

المراجع ---------- ٧٣

مقدمة إلى Design Patterns

تعتبر Design Patterns من أهم المفاهيم في عالم البرمجة وتصميم البرمجيات. نماذج معمارية وأساليب مجربة ومعترف بها لحل مشكلات معينة في التصميم. تساعد المطورين على تنظيم الشفرة وتحسين قابلية الصيانة وإعادة الاستخدام.

ما هي Design Patterns؟

Design Patterns هي أفكار وتصاميم معمارية تمثل حلاً لمشاكل معينة متكررة . تم توثيق هذه الأنماط في الكتب والمقالات، وتم تطبيقها في العديد من اللغات البرمجية.

أنواع Design Patterns

هناك ثلاثة أنواع رئيسية من Design Patterns :

١. **Creational Patterns :** تتعامل مع عملية إنشاء الكائنات مثل Singleton Factory Builder ، Prototype .

٢. **Structural Patterns :** ترتبط بتركيب الكائنات والعلاقات بينها مثل Adapter Decorator، Composite ، Proxy ، Bridge ، Fly Weight .

٣. **Behavioral Patterns** : تركز على تفاعل الكائنات وتبادل المعلومات بينها. مثل Template, Observer ، Visitor ، Chain of responsibility ، Strategy .

فوائد استخدام Design Patterns

- **إعادة الاستخدام**: يمكن استخدام الأنماط المعمارية في مشاريع متعددة.
- **تبسيط التصميم**: تساعد في تنظيم الشفرة وتجنب التكرار.
- **تحسين الصيانة**: يجعل الشفرة أكثر قابلية للصيانة والتعديل.

ملاحظات :

- من الأخطاء المنتشرة اعتبار نماذج التصميم حلولاً كاملة أو جاهزة للاستخدام المباشر فهي نماذج تحتاج للتكييف و التحديد لتواجه مشكلة محددة .
- أغلب نماذج التصميم تعتمد على OOP .
- تفرعت نماذج التصميم من مفهوم يدعى بال OCP (Open Closed Principle)

أي أن النظام قابل للتوسع حيث يمكن إضافة وظائف جديدة من دون التعديل على بنية الصفوف الخاصة بالنظام.

- علاقات ال Class Diagrams :

Association

Composition

Aggregation

Implementation

Creational Patterns

Singleton Pattern

يضمن هذا النمط وجود نسخة واحدة فقط من صف (class) معين في أي وقت ، مع توفير نقطة وصول عامة لهذه النسخة.

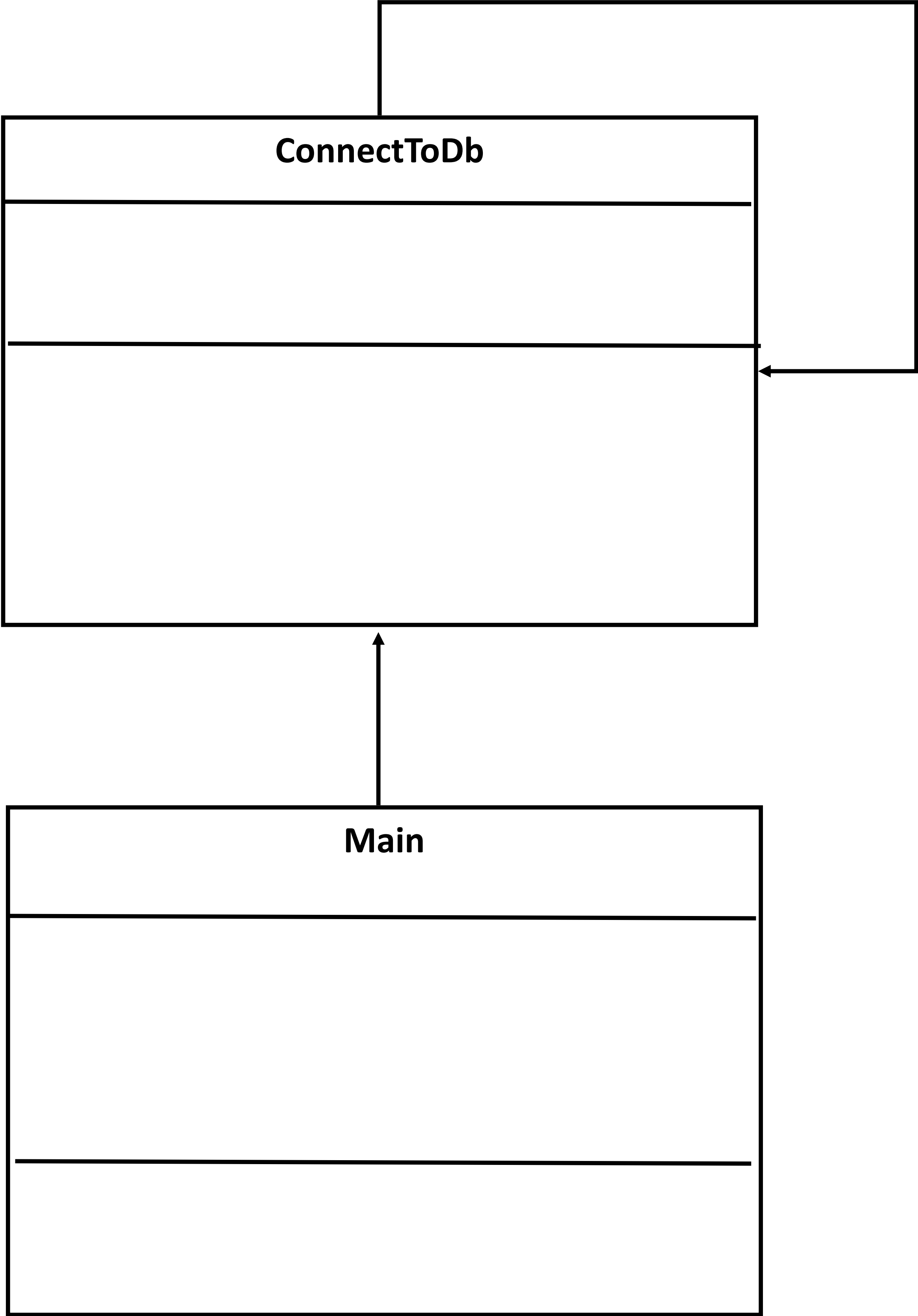
حالات الاستخدام : يُستخدم هذا النمط عادةً في الحالات التي نحتاج فيها إلى التحكم في الوصول إلى مورد مشترك، مثل الاتصال بقاعدة بيانات أو ملف تهيئة النظام أو الثوابت كأيام الأسبوع (enum). من مزايا هذا النمط أنه يمنع إنشاء أكثر من نسخة من الصف مما يساعد في توفير الموارد ويضمن تناسق البيانات عبر التطبيق.

مثال : يوجد لدينا صف يدعى ConnectToDb يقوم بالاتصال بقاعدة البيانات ، علماً أن عملية الاتصال يجب ضمان عدم تكرارها :

```
3   public class ConnectToDb {
4
5     / تم تعريف ك privat/e لمنع الإنشاء الخارجي /
6     / تم استخدام Static لضمان global access بالإضافة إلى ضمان وجودة نسخة واحدة فقط /
7
8   private static ConnectToDb instance ;
9
10      public static ConnectToDb getInstance(){
11          if(instance == null)
12          {
13              instance = new ConnectToDb();
14               System.out.println("new connection to data base");
15               return instance;
16          }
17           System.out.println("already connected to data base");
18          return instance;
19      }
20
21  }
```

```
public class Main {
    public static void main(String[] args ){
    / ستطبع new connection to data base /
    ConnectToDb single = ConnectToDb.getInstance();
    / ستطبع already connected to data base /
    ConnectToDb single1 = ConnectToDb.getInstance();
    }
}
```

Singleton UML



Factory Pattern

يوفر واجهة لإنشاء الكائنات (objects)داخل فئات رئيسية (superclasses)لكنها تسمح في نفس الوقت للفئات الثانوية (subclasses)بتغيير نوع تلك الكائنات التي سيتم إنشاؤها.

يقوم هذا النمط بتقليل التعقيد عن طريق توحيد عملية إنشاء الكائنات لتتم في مكان واحد مما يسهل إداراتها و تعديلها .

حالات الاستخدام : يُستخدم عندما يكون هناك حاجة لتفصيل أو تكوين كائنات معقدة أو عندما يكون هناك العديد من الأنواع المختلفة من الكائنات التي يمكن أن تكون لها خصائص مشتركة ولكن تفاصيلها تختلف بناءً على السياق ، مثل إنشاء واجهات مستخدم مختلفة تتناسب مع أنظمة تشغيل متعددة، أو إنشاء كائنات تمثل مختلف أنواع الدفع في نظام للتجارة الإلكترونية، حيث يمكن تغيير الكائن المُنشأ بناءً على الطريقة المختارة للدفع.

مثال : يوجد لدينا مطعم يقوم بإعداد أنواع مختلفة من الساندويش منها Cheese و Burger ، كل Sandwish لها حجم و سعر مختلف ، نريد ضمان إمكانية إضافة أنواع جديدة من الساندويش مع المحافظة على مبدأ OCP :

```
/ تم التعريف claa/s abstrac/t لجعل الصف sandwish قالب لجميع السندويشات /

public abstract class Sandwish {

    public String setSandwish_price(int price) {

        System.out.println("Sandwish price is " + price);
        return "Sandwish price is " + price;
    }
    public String setSandwishSize(int size) {
        System.out.println("Sandwish size is " + size);
        return " Sandwish size is " + size;
    }
    public void prepare(String sandwishName){
        System.out.println( sandwishName + " Sandwish is prepared");
    }
}
```

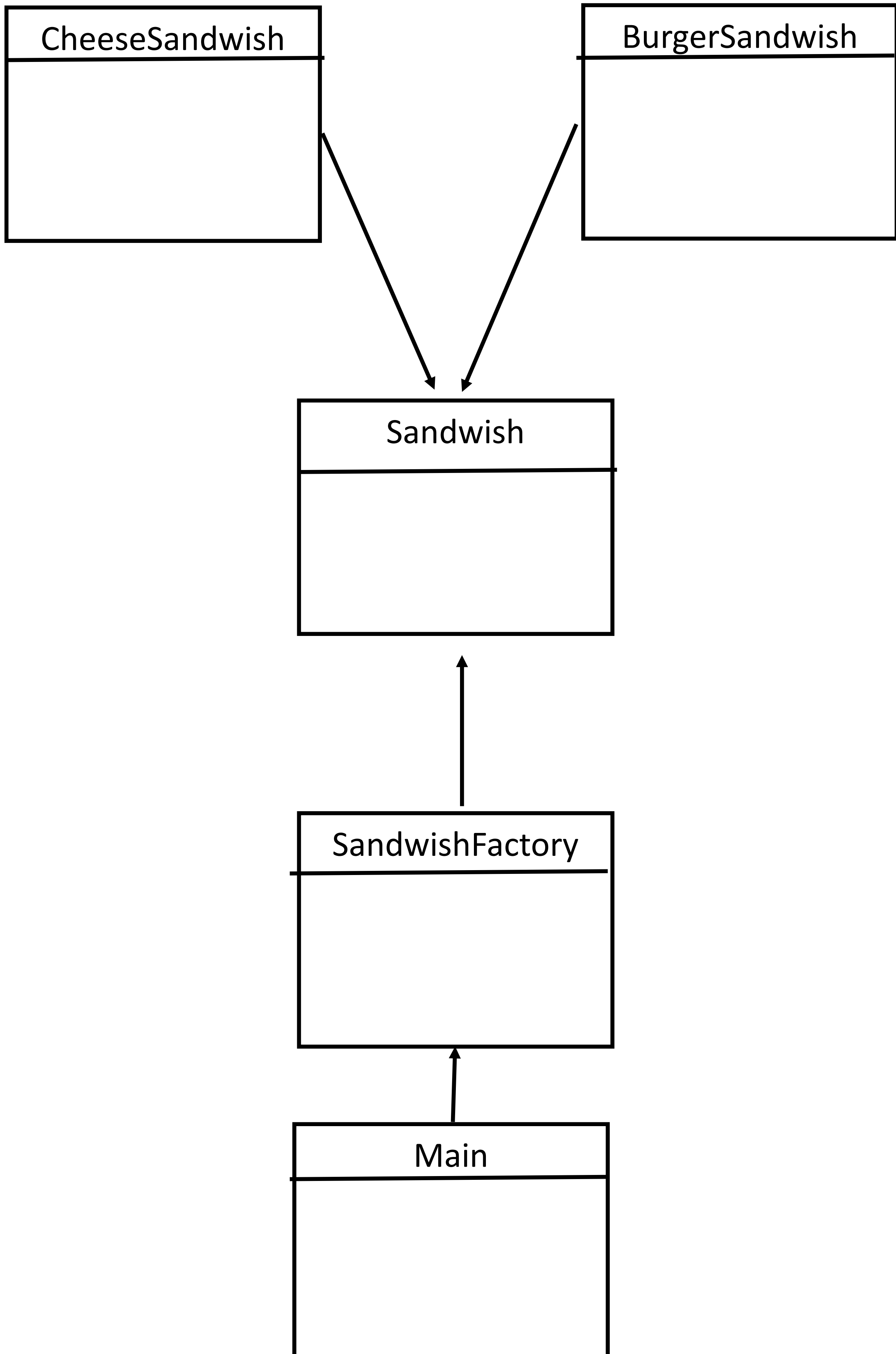
```
public class CheeseSandwish extends Sandwish{

    public CheeseSandwish(){
        setSandwish_price(10);
        setSandwishSize(2);
    }


}
```

```
public class BurgerSandwish extends Sandwish{

    public BurgerSandwish(){
        setSandwish_price(20);
        setSandwishSize(3);
    }

}
```

```java
public class SandwishFactory {
    private static Sandwish sandwish;
    / عند ادخال معرف السندويشة يتم صناعتها /
    public static Sandwish newSandwish(int id){
        if(id ==1)
        return sandwish = new CheeseSandwish();
        return sandwish = new BurgerSandwish();
    } ;
}
```

```java
public class main {
    public static void main(String[] args ) {

        Sandwish cheese = SandwishFactory.newSandwish(1);
        cheese.prepare("cheese");
        Sandwish burger = SandwishFactory.newSandwish(2);
        burger.prepare("burger");
    }
}
```

Factory UML



Builder Pattern

يُستخدم لفصل عملية بناء كائن معقد عن تمثيله ، مما يسمح بإنشاء تمثيلات مختلفة للكائن باستخدام نفس عملية البناء .يختلف عن ال Factory بأن عملية إنشاء الكائن تتم على مراحل ، يُعتبر هذا النمط مفيدًا في الحالات التي يكون فيها الكائن المراد إنشاؤه يتطلب تهيئة مرهقة وتفصيلية للكثير من الحقول والكائنات المتداخلة.

حالات استخدام :

تجنب التكاليف العالية لإنشاء الكائنات: يُستخدم عندما يكون إنشاء كائن جديد عملية مكلفة أو معقدة.

التعامل مع الكائنات ذات الحالات المعقدة: يُستخدم لإنشاء كائنات تتطلب تهيئة متعددة الخطوات ولا يمكن تمثيلها بشكل كافٍ باستخدام منشئ واحد.

تحسين قابلية الصيانة: يُساعد في تنظيم الكود وجعله أكثر قابلية للصيانة من خلال تقسيم عملية البناء إلى خطوات متعددة.

مثال : برنامج يقوم بتحويل الملفات النصية إلى ملفات Html أو ملفات Pdf تتم عملية التحويل على ثلاث مراحل ، أول مرحلة إنشاء ملف ، ثاني مرحلة بناء المحتويات ، ثالث مرحلة حفظ الملف ، بحيث تتم عملية التسليم بشكل منفصل عن البناء :

```
public interface Builder {
    public void file_type();
    public void file_size();
    public File getFile();
}
```

```
public class File {
    / اسم الملف /
    public String fileName;

    / تابع يعيد اسم الملف /

    public String getFileName(){
        return this.fileName;
    }
}
```

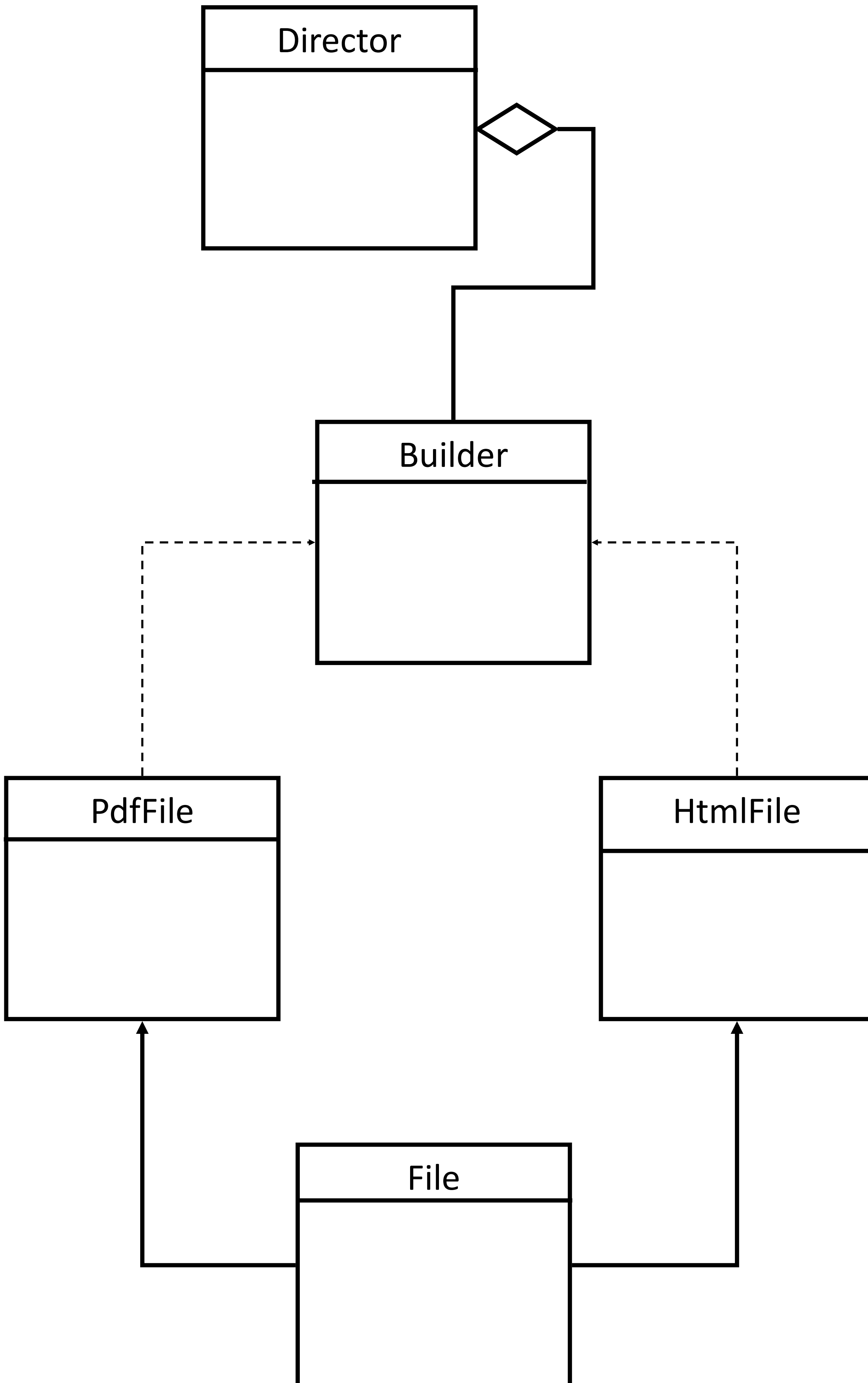
```java
public class HtmlFIle implements Builder{

    private File f ;

    public HtmlFIle(){
        this.f = new File();
        this.f.fileName = "new html";
    }
    @Override
    public void file_type() {
        System.out.println("file type is HTML");
    }

    @Override
    public void file_size() {
        System.out.println("file size is html");
    }

    @Override
    public File getFile() {
        System.out.println("getting " + this.f.fileName);
        return this.f;
    }
}
```

```java
public class PdfFile implements Builder{

    private File f ;
    public PdfFile(){
        this.f = new File();
        this.f.fileName = "new pdf";
    }

    @Override
    public void file_type() {
        System.out.println("file type is PDF");
    }

    @Override
    public void file_size() {
        System.out.println("file size is pdf");
    }

    @Override
    public File getFile() {
        System.out.println("getting " + this.f.fileName);
        return this.f;
    }
}
```

```java
public class Director {
    public Builder b;
    public void construct(Builder b) {
        / يتم بناء الملف على مراحل /
        this.b = b;
        this.b.file_type();
        this.b.file_size();
    }
}
```

```java
public class main {
    public static void main(String[] args ) {
        Director d = new Director();

        Builder pdfFile = new PdfFile();
        d.construct(pdfFile);
        //final product return
        File pdf = pdfFile.getFile();


        Builder htmlFile = new HtmlFIle();
        d.construct(htmlFile);
        //final product return
        File html = htmlFile.getFile();
    }
}
```

Builder UML



Prototype Pattern

يستخدم لإنشاء كائنات جديدة من خلال نسخ كائن موجود بالفعل ، بدلاً من إنشاء كائن جديد من الصفر. يُعتبر هذا النمط مفيدًا في الحالات التي يكون فيها إنشاء كائن جديد عملية مكلفة أو معقدة.

حالات استخدام :

تجنب التكاليف العالية لإنشاء الكائنات: عندما يكون إنشاء كائن جديد عملية مكلفة بسبب الحاجة إلى تهيئة الكائن بعد إنشائه.

التعامل مع الكائنات ذات الحالات المعقدة: يُستخدم عندما يكون للكائن حالة معينة يصعب الوصول إليها أو إعادة إنشائها.

تقليل الاعتماديات: يُفيد في تقليل الاعتماديات بين الكود وفئات الكائنات، حيث يمكن نسخ الكائن دون الحاجة إلى معرفة فئته الدقيقة.

تحسين الأداء: يُستخدم لتحسين الأداء عن طريق نسخ الكائنات بدلاً من إعادة إنشائها، خاصةً في الأنظمة التي تتطلب إنشاء كائنات متعددة بسرعة.

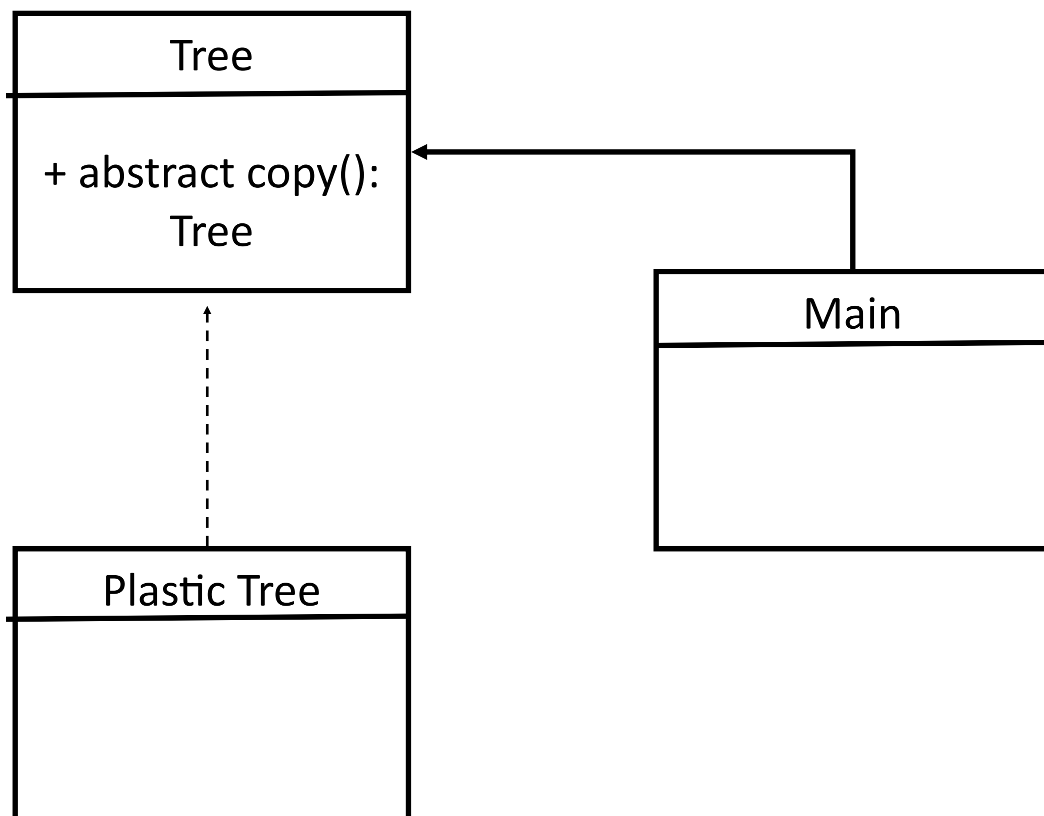
مثال : لعبة متاهة تحتوي على مجموعة كبيرة من الأشجار :

```
public abstract class Tree {
    public abstract Tree copy();
}
```

```
public class PlasticTree extends Tree {

public double mass;
public int height

public PlasticTree(double mass , int height){
  this.mass = mass;
  this.height = height;
}

@override
 public Tree copy() {
   PlasticTree p = new PlasticTree(this.getMass(),this.getHeight);
   return p;
  }

}
```

```java
public class Main {

PlasticTree T = new PlasticTree(100,10);
PlasticTree anotherT = (PlasticTree) T.copy();
anotherT.setPosition(1,2);
PlasticTree anotherT2 = (PlasticTree) T.copy();
anotherT2.setPosition(2,2);
PlasticTree anotherT3 = (PlasticTree) T.copy();
anotherT3.setPosition(3,2);
PlasticTree anotherT4 = (PlasticTree) T.copy();
anotherT4.setPosition(4,2);
PlasticTree anotherT5 = (PlasticTree) T.copy();
anotherT5.setPosition(5,2);

}
```

Prototype UML



Structural Patterns

Adapter Pattern

يُستخدم لتمكين الكائنات ذات الواجهات غير المتوافقة من العمل معًا. يُعرف أيضًا باسم المغلّف، ويُمكنه تحويل واجهة كائن إلى واجهة أخرى يتوقعها العميل.

حالات استخدام :

التكامل مع الكود القديم: يُستخدم لدمج الكود القديم أو المكتبات غير المتوافقة مع واجهات المشاريع الجديدة دون تعديل الكود المصدري.

جعل الفئات غير المتوافقة تعمل معًا: يُستخدم لتمكين التواصل بين فئات أو أنظمة مختلفة لديها واجهات متباينة.

مثال : لدينا نظام تصنيع مركبات ذو واجهة Vehicle تقدم توصيف للتسارع و المكابح و البوق الخاص بالمركبة يوجد صف Car ينفذ هذه الواجهة ، نريد إضافة صف Bike مع مراعاة أن الدراجة تدوس بدل التسارع ، و ترن الجرس بدل البوق ، دون تغيير الواجهة Vehicle :

```java
public interface Vehiacle {
    void accelerate();
    void pushBreak();
    void soundHorn();
}
```

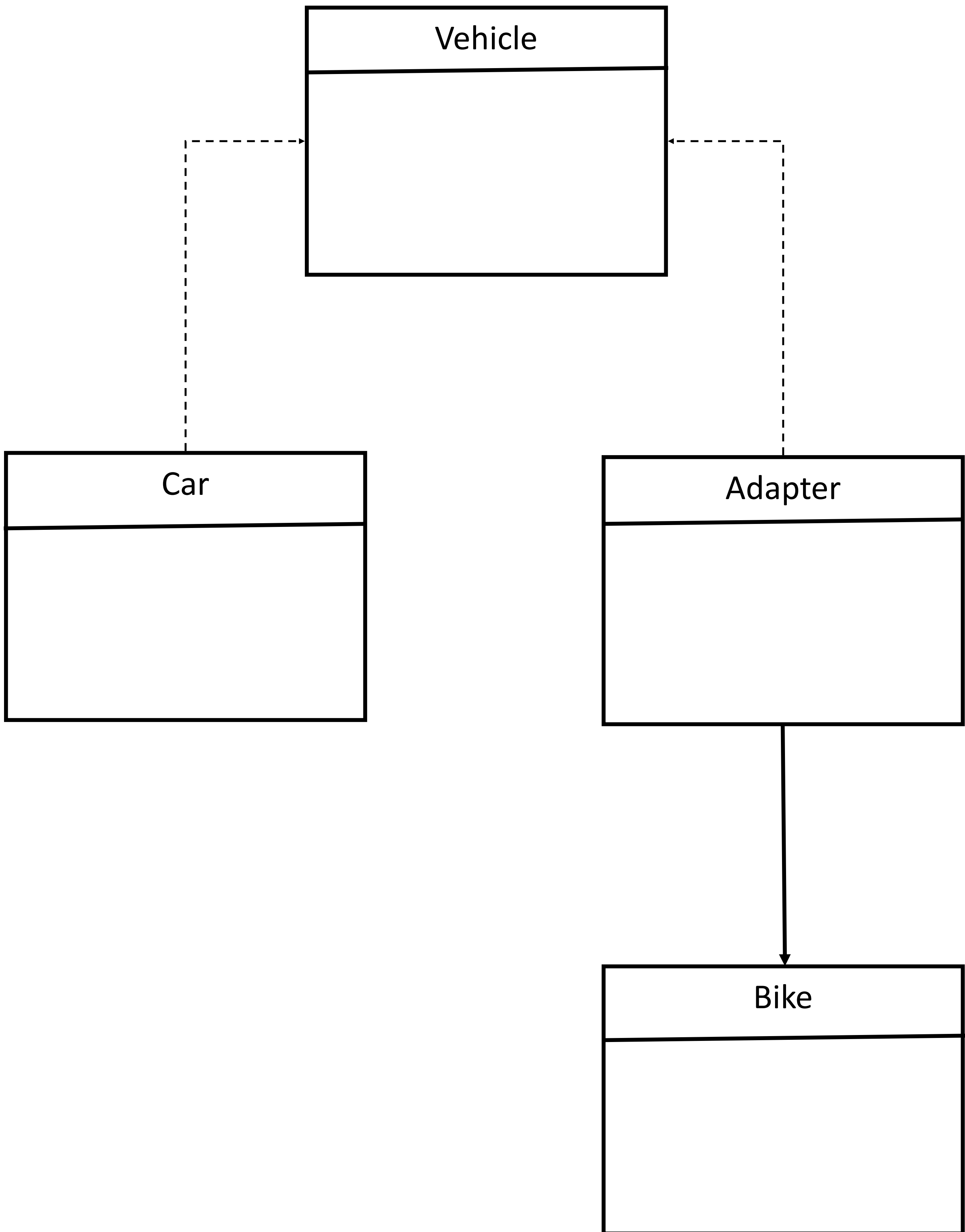
```java
public class Car implements Vehiacle{
    @Override
    public void accelerate() {
        System.out.println("car Start Moving");
    }

    @Override
    public void pushBreak() {
        System.out.println("car Stopped");
    }

    @Override
    public void soundHorn() {
        System.out.println("beeb .... beeb");
    }
}
```

```
public class Bycicle {
    public void pedal(){
        System.out.println("bike start moving");
    }
    public void ring(){
        System.out.println("ring...ring");
    }
}
```

```
public class Adapter implements Vehiacle{

     private Bycicle bycicle ;

     public Adapter (Bycicle bycicle) {
         this.bycicle = bycicle;
     }

     @Override
     public void accelerate() {
     this.bycicle.pedal();
     }

     @Override
     public void pushBreak() {
     system.out.print("bike stopped");
     }

     @Override
     public void soundHorn() {
     this.bycicle.ring();
     }
}
```

```
public class Main {
    public static void main(String[] args ) {

        Vehiacle car = new Car();
        useVehiacle(car);
        Vehiacle bike = new Adapter(new Bycicle());
        useVehiacle(bike);

    }

    private static void useVehiacle(Vehiacle car) {
        car.accelerate();
        car.pushBreak();
        car.soundHorn();
        System.out.println();
    }
}
```

Adapter UML



Proxy Pattern

يُستخدم لتوفير بديل لكائن موجود لديك، ويتحكم في الوصول إلى الكائن الأصلي. يمكنك من خلاله تنفيذ إجراءات قبل أو بعد وصول الطلب إلى الكائن الأصلي، مما يُمكن من تحسين الأداء والأمان.

حالات استخدام :

التحكم في الوصول: يُستخدم للتحكم في الوصول إلى كائن، مثل تحميل كائن ثقيل فقط عند الحاجة إليه.

تأخير التهيئة (Lazy Initialization): يُستخدم لتأخير إنشاء كائن حتى يُطلب فعليًا، مما يُحسن الأداء.

توفير الحماية: يُستخدم لإضافة طبقة حماية للكائنات التي تُعتبر حساسة أو معقدة.

التسجيل والتدقيق: يُستخدم لتسجيل الطلبات التي تُرسل إلى الكائن الأصلي.

مثال : في مزود انترنت نريد حجب الخدمة في حال تم الاتصال بأحد المواقع المحظورة :

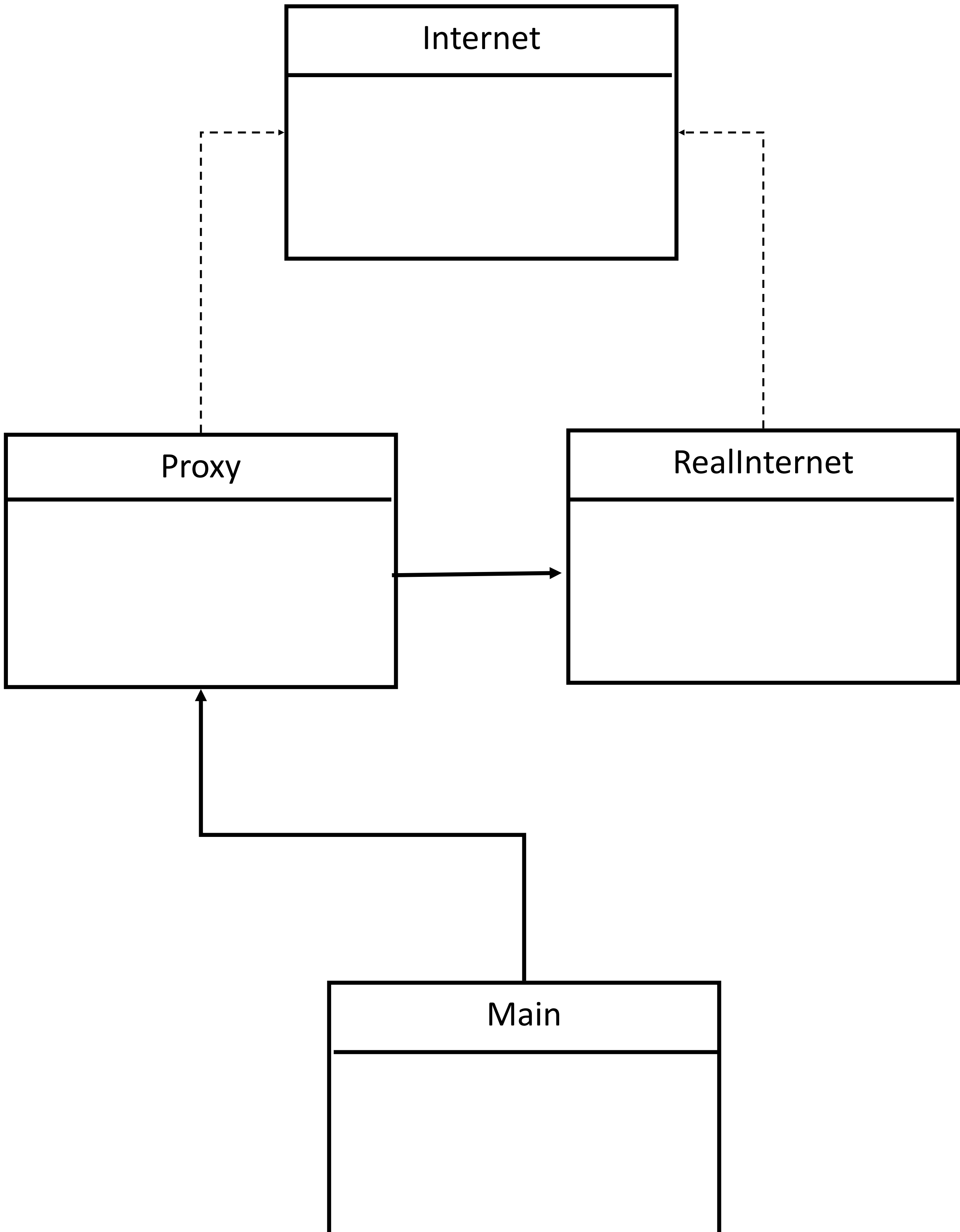
```
public interface Internet {
    public void connectTo(String server);
}
```

```
public class RealInternet implements Internet{
    @Override
    public void connectTo(String server) {
        System.out.println("connecting to " + server);
    }
}
```

```java
public class Proxy implements Internet{

    private Internet internet = new RealInternet();
    private static List<String>bannedSite ;
    static {
        bannedSite = new ArrayList<>();
        bannedSite.add("abc.com");
        bannedSite.add("bbc.com");
        bannedSite.add("dbc.com");
    }

    @Override
    public void connectTo(String server) {

        if(bannedSite.contains(server))
        {
            System.out.println(server + " is banned");
        }
        else
        {System.out.println(server + " is accepted");
         internet.connectTo(server);
        }

    }
}
```

```java
public class Main {
     public static void main(String[] args ) {
    Internet internet = new Proxy();
    internet.connectTo("bbc.com");
    internet.connectTo("abbc.com");
     }
}
```

Proxy UML



Composite Pattern

يُستخدم لتنظيم الكائنات في هياكل شجرية لتمثيل العلاقات الجزئية/الكلية. يسمح بتركيب الكائنات في هياكل شجرية ليعاملها على أنها وحدة واحدة أو كائن واحد . مما يُسهل إدارة التعقيدات.

حالات استخدام :

عندما تريد تمثيل هياكل جزئية/كلية: يُستخدم لتمثيل وإدارة هياكل البيانات التي تحتوي على كائنات يمكن أن تكون مركبة من كائنات أخرى.

عندما تريد تبسيط الكود: يُساعد في تبسيط الكود عن طريق السماح بمعاملة المجموعات من الكائنات ككائن واحد.

عندما تريد إضافة أو إزالة الكائنات ديناميكيًا: يُمكن أن يُسهل إضافة وإزالة الكائنات من الهيكل الشجري دون تأثير كبير على الكود.

مثال : لدينا نظام خاص بشركة برمجيات يحتوي على واجهة موظفين Employee تقدم خدمة عرض بيانات الموظف ، يوجد نوعين من الموظفين المطورين Developers ، و المدراء Admins ، كل مدير مسؤول عن قائمة من المطورين :

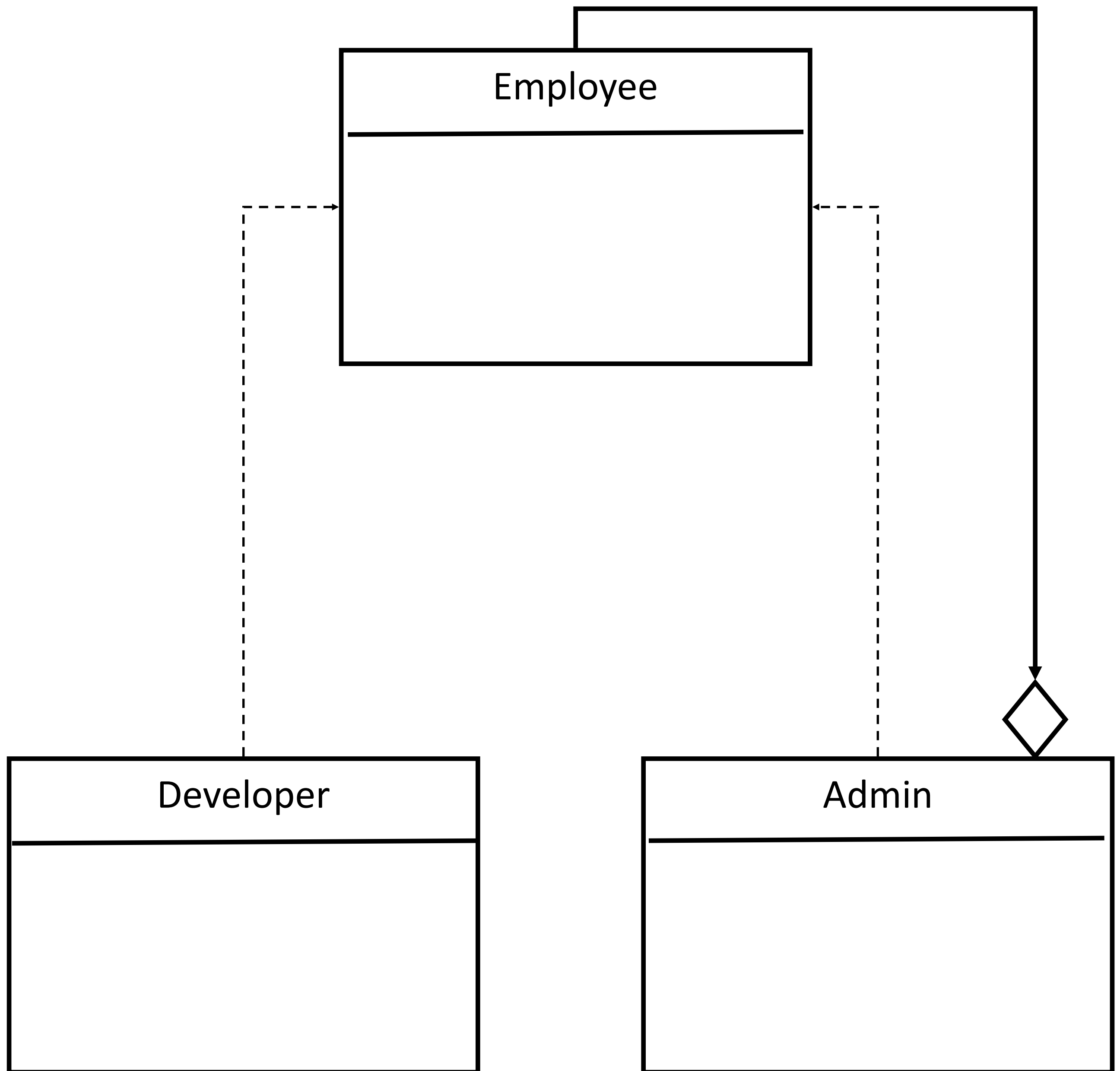
```
public interface Employee {
    public void showEmployessDetails();
}
```

```
public class Developer implements Employee {

    String name;
    String Id;
    String postion;
    public Developer(String name,String Id ,String postion){
        this.Id = Id;
        this.name = name;
        this.postion = postion;
    }
    @Override
    public void showEmployessDetails() {
        System.out.println("empId: " + this.Id + " empname: " + this.name + " empPos: " + this.postion);
    }
}
```

```java
public class Admin implements Employee{
    String name;
    String Id;
    String postion;
    List <Employee> employees = new ArrayList<>();

    @Override
    public void showEmployessDetails() {
        System.out.println("empId: " + this.Id + " empname: " + this.name + " empPos: " + this.postion);
    }
    public Admin(String name,String Id ,String postion){
        this.Id = Id;
        this.name = name;
        this.postion = postion;
    }
    public void add(Developer dev){
        employees.add(dev);
    }
    public void remove(Developer dev){
        employees.remove(dev);
    }
    public void showDev(){
        for (int i = 0; i < employees.size(); i++) {
            if(employees.get(i)!=null){
                employees.get(i).showEmployessDetails();
            }
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args ) {
        Developer dev1 = new Developer("sami","1","eng");
        Developer dev2 = new Developer("sami1","2","eng");
        Admin admin1 = new Admin("sami3" , "3" , "eng");

        admin1.add(dev1);
        admin1.add(dev2);

        admin1.showDev();
    }
}
```

Composite UML



Decorator Pattern

يستخدم لإضافة سلوكيات جديدة إلى الكائنات دون تغيير الكود الموجود في الفئات الأخرى. يتم ذلك عن طريق إنشاء فئة مزخرفة تُحيط الكائن الأصلي وتُضيف وظائف إضافية.

حالات استخدام :

عندما تريد إضافة سلوكيات جديدة أو مسؤوليات إضافية إلى كائنات فردية: يمكن إضافة سلوكيات بشكل ديناميكي.

عندما تحتاج إلى إضافة سلوكيات يمكن إزالتها: نظرًا لأن السلوكيات المضافة يمكن إزالتها، يوفر هذا مرونة أكبر في إدارة السلوكيات.

عندما تريد تجنب إنشاء فئات فرعية متعددة: بدلاً من إنشاء العديد من الفئات الفرعية لتوسيع الوظائف، يمكن استخدام نمط المزخرف لإضافة وظائف محددة بطريقة أكثر مرونة.

مثال: مطعم بيتزا يقوم بتصنيع عدة أنواع من البيتزا ، كل نوع عبارة عن مقادير إضافية أو تحسينات :

```
public interface Pizza {
    public void cooke();
}
```

```
4
5   public class BasicPizza implements Pizza{
6       @Override
7       public void cooke() {
8           System.out.println("cooking basic pizza");
9       }
0   }
```

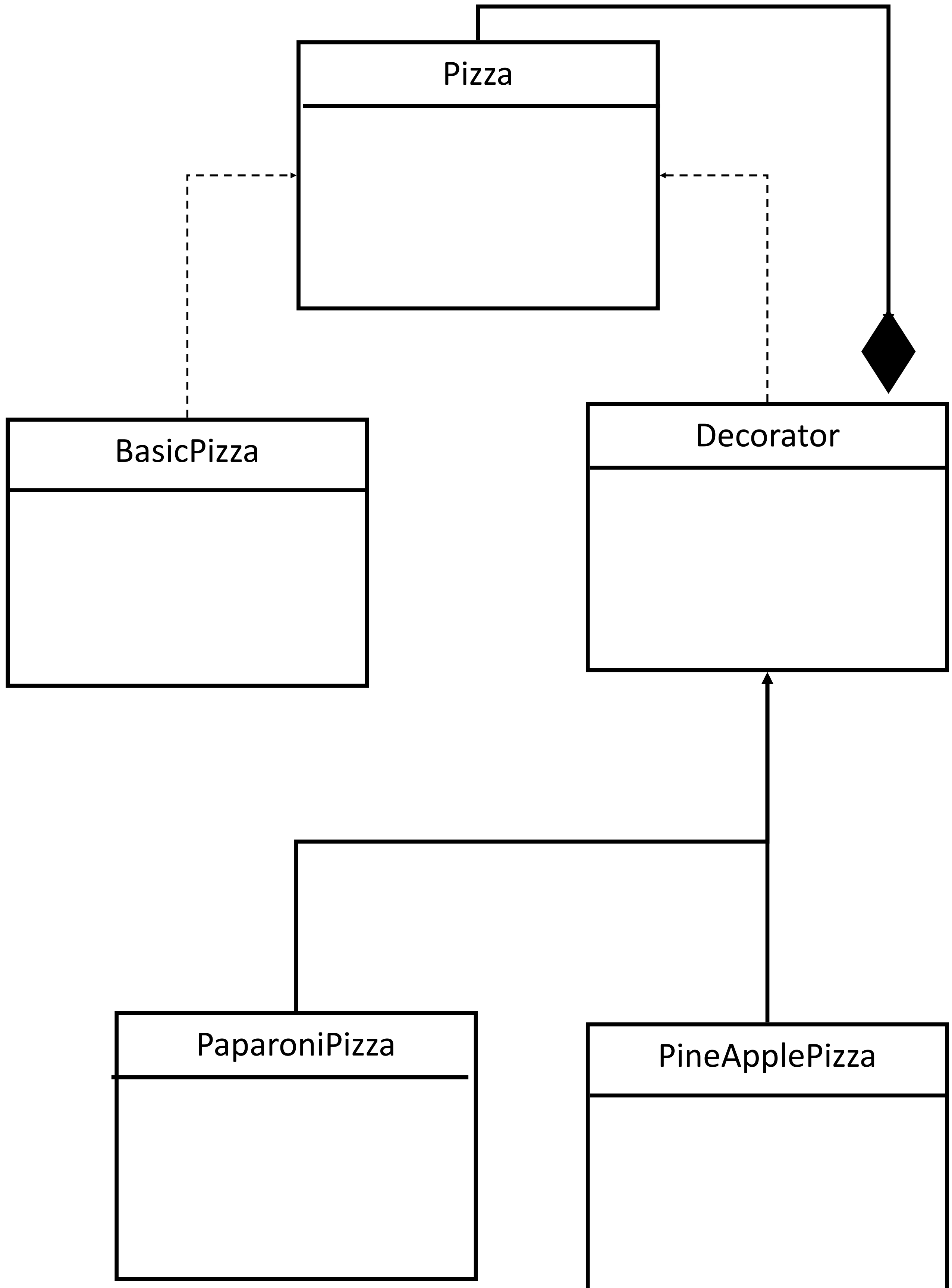
```java
public abstract class Decorator implements Pizza{

    Pizza pizza;
    public  Decorator(Pizza pizza){
        this.pizza =pizza;
    }
    @Override
    public void cooke() {
      this.pizza.cooke();
    }
}
```

```java
public class PaparoniPizza extends Decorator{

    public  PaparoniPizza(Pizza pizza){
        super(pizza);
    }
    @Override
    public void cooke() {
        super.cooke();
        System.out.println("Cooking paparoniPizza");
    }
}
```

```java
public class PineApplePizza extends Decorator{

    public  PineApplePizza(Pizza pizza){
    super(pizza);
   }

    @Override
    public void cooke() {
        super.cooke();
        System.out.println("cooking pineApplePizza");
    }
}
```

```java
public class main {
    public static void main(String[] args ) {
    Pizza pizza = new PineApplePizza(new PaparoniPizza(new BasicPizza()));
    pizza.cooke();
    }
}
```

Decorator UML



Bridge Pattern

يُستخدم لفصل التجريد (Abstraction) عن تنفيذه (Implementation) بحيث يمكن للاثنين أن يتطوروا بشكل مستقل. يُعتبر هذا النمط مفيدًا بشكل خاص عندما يكون هناك فئات متعددة تتشارك في واجهات أو تنفيذ معين، ولكن نريد تجنب ربطها بشكل ثابت لتسهيل المرونة و إتاحة إمكانية إعادة الاستخدام.

حالات استخدام :

عندما تريد تجنب ربط التجريد بتنفيذه: يسمح بتغيير التنفيذ دون التأثير على العملاء.

عندما تحتاج إلى توسيع الفئات في عدة أبعاد مستقلة: يساعد في تنظيم الكود بشكل أفضل ويجعله أكثر قابلية للصيانة.

عندما تريد مشاركة تنفيذ بين عدة فئات: بدلاً من نسخ الكود، يمكن فصل التنفيذ إلى فئة منفصلة واستخدام مرجع في التجريد.

مثال : أداة تقوم برسم نوعين من الأشكال هما Circle , Square باستخدام واجهة Shape (تمثل التجريد) ، نريد تلوين هذه الأشكال باستخدام واجهة منفصلة Color (تمثل التنفيذ) بلونين هما Black , Red ، بحيث نتمكن من تغيير ألوان الأشكال ديناميكيا من دون الحاجة لتعديل الأشكال :

```java
public abstract class Shape {
    Color color;
    public Shape(Color color){
        this.color = color;
    }
    public abstract String draw();
}
```

```java
public interface Color {
    public String fill();
}
```

```java
public class Red implements Color{
    @Override
    public String fill() {
        return "color filled red";
    }
}
```

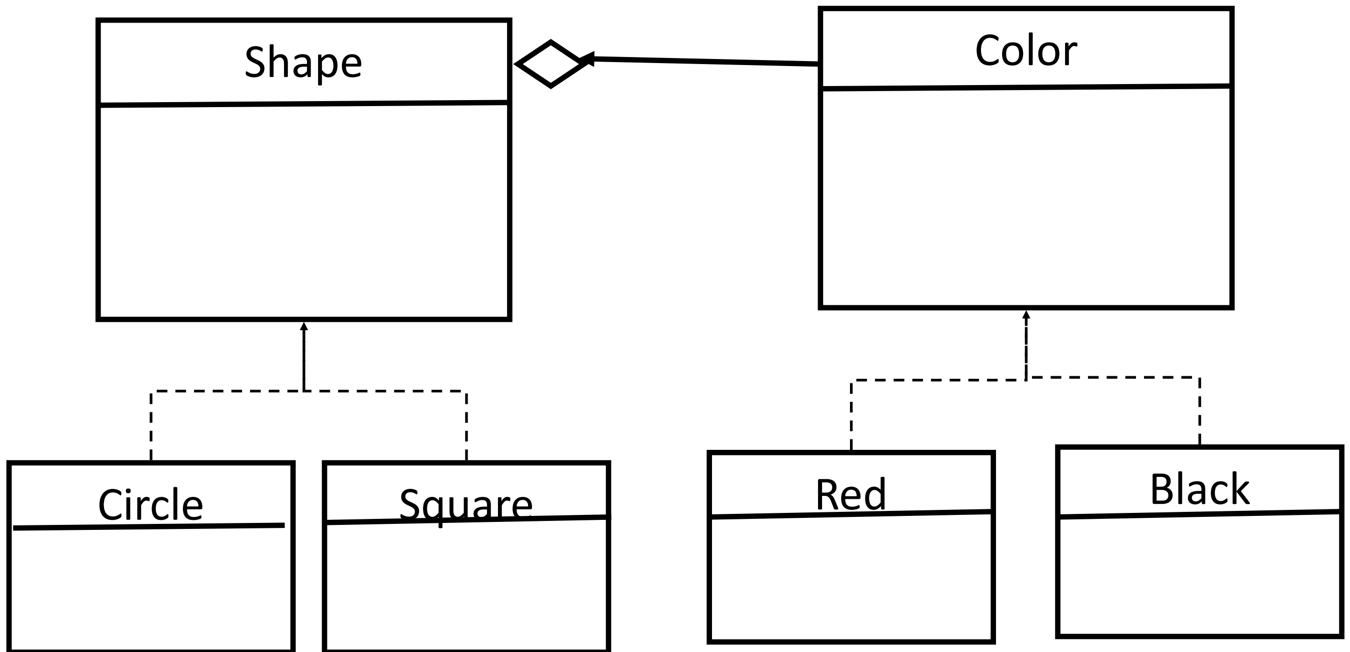
```java
public class Black implements Color{
    @Override
    public String fill() {
        return "color filled black";
    }
}
```

```java
public class Circle extends Shape{
    public Circle(Color color) {
        super(color);
    }

    @Override
    public String draw() {
        System.out.println("circle drawn" + color.fill());
        return "cr";
    }
}
```

```java
public class Sqware extends Shape{
    public Sqware(Color color) {
        super(color);
    }

    @Override
    public String draw() {
        System.out.println("sqware drawn " + color.fill());
        return "sq";
    }
}
```

```java
public class main {
    public static void main(String[] args ) {
        Shape s =new Sqware(new Red());
        s.draw();
    }

}
```

Bridge UML



Fly Weight Pattern

يهدف إلى تقليل استهلاك الذاكرة للبرامج التي تتعامل مع كميات كبيرة من الكائنات. يتم ذلك عن طريق مشاركة الحالة المشتركة بين الكائنات بدلاً من الاحتفاظ بنسخة من البيانات في كل كائن.

حالات استخدام :

تقليل استهلاك الذاكرة: يُستخدم عندما تحتاج إلى إدارة عدد كبير من الكائنات ذات الحالة المشتركة، مما يقلل من الحاجة إلى الذاكرة.

تحسين الأداء: يُساعد في تحسين أداء البرامج التي تستخدم كميات كبيرة من الكائنات خاصةً في البيئات ذات الموارد المحدودة.

تجنب الأخطاء المتعلقة بالذاكرة: يُقلل من فرص حدوث أخطاء الذاكرة مثل java.lang.OutOfMemoryErrorعن طريق تقليل عدد الكائنات المستخدمة.

مثال : مصنع سيارات ينتج سيارات وباصات بألوان مختلفة. في الطريقة التقليدية، سيتم إنشاء كائن جديد لكل سيارة أو باص بلون معين، مما يؤدي إلى استهلاك كبير للذاكرة إذا كان عدد السيارات والباصات كبيرًا جدًا.

باستخدام نمط (Flyweight Pattern)، يمكن للمصنع تقليل استهلاك الذاكرة بشكل كبير. بدلاً من إنشاء كائن جديد لكل سيارة أو باص، يُنشأ كائن لكل نوع ولون مرة واحدة فقط. عندما يحتاج المصنع إلى سيارة أو باص بلون معين، يتم استخدام الكائن الموجود بالفعل بدلاً من إنشاء واحد جديد.

إذا كان هناك طلب على ١٠٠ سيارة حمراء و٥٠ باص أسود، بدلاً من إنشاء ١٥٠ كائنًا، يُنشأ كائن واحد للسيارة الحمراء وآخر للباص الأسود، ويتم مشاركتهما لتلبية جميع الطلبات. هذا يقلل من الحاجة إلى الذاكرة ويحسن الأداء، خاصةً في الأنظمة التي تتعامل مع كميات كبيرة من الكائنات.

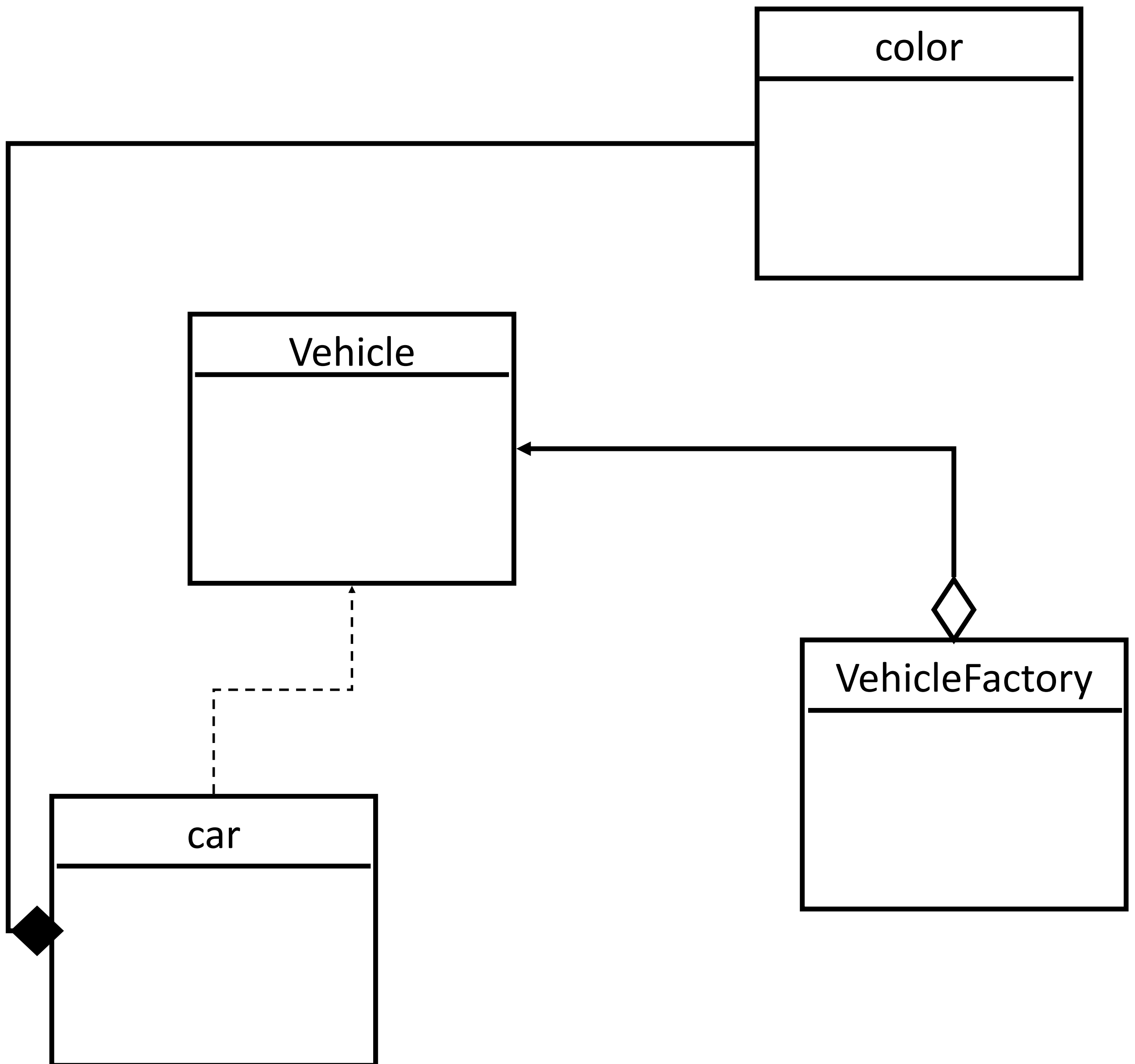
```java
public interface Veihcle {
        public void start();
        public void stop();
        public Color getColor();
}
```

```java
public class Color {
        public String Color ;
        public void set_color(String color){
                this.Color = color;
        }
        public String getColor(){
                return this.Color;
        }
}
```

```java
public class VeihcleFactory {
    private Map<Color , Veihcle> vehiclesCache = new HashMap<>();
    / يتم التحقق من وجود السيارة ضمن ال map في حال لم تكن موجودة ينشا كائن جديد باللون الجديد /
    public Veihcle create(Color color){
        Veihcle n = vehiclesCache.computeIfAbsent(color,newColor->{
            return new Car(newColor);
        });
        return n;
    }
}
```

```java
public class main {
        public static void main(String[] args ) {
            VeihcleFactory v = new VeihcleFactory();
            Color red = new Color();
            Color blue = new Color();
            red.set_color("red");
            blue.set_color("blue");
            v.create(red);
            v.create(red);
            v.create(blue);
        }
}
```

Fly Weight UML



Facad Pattern

يُستخدم عادةً عندما يكون هناك حاجة إلى واجهة بسيطة للوصول إلى نظام معقد ، أو عندما يكون النظام صعب الفهم، أو عندما يكون هناك حاجة إلى نقطة دخول لكل مستوى من مستويات البرمجيات المتعددة الطبقات، أو عندما تكون الاستخدامات و التنفيذات لنظام فرعي مرتبطة بشكل وثيق.

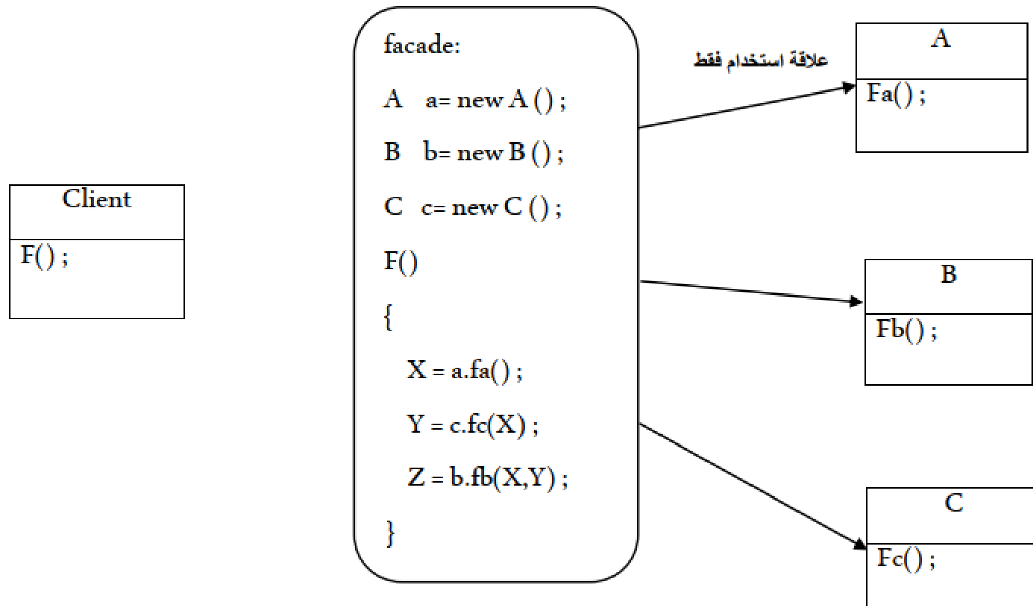
حالات استخدام :

تبسيط الأنظمة الخارجية المعقدة: يُبسط استخدام قواعد البيانات وتنفيذ الاستعلامات ومعالجة النتائج، مقدمًا واجهة نظيفة للتطبيق.

إخفاء تعقيدات واجهات البرمجة التطبيقية الخارجية: يُبسط استخدام واجهات البرمجة التطبيقية الخارجية من خلال إخفاء تعقيدات المصادقة وتنسيق الطلبات وتحليل الاستجابات .

توفير واجهة مستخدم سهلة لأنظمة معقدة: يُقدم واجهة مستخدم بسيطة وسهلة الاستخدام للأنظمة المعقدة، مما يُسهل على المستخدمين التفاعل مع النظام دون الحاجة إلى فهم التفاصيل المعقدة .

مثال : لدينا عدة توابع من عدة صفوف وبدلاً من التواصل مع جميع الصفوف نتواصل مع صف واحد يرتبط مع هذه الصفوف .



Behavioral Patterns

Visitor Pattern

يُستخدم لفصل الخوارزميات عن الكائنات التي تشغلها ، مما يسمح بإجراء عمليات مختلفة على هذه الكائنات دون تعديل تعريفاتها. يُعتبر هذا النمط مفيدًا عندما تحتاج إلى إجراء عمليات متنوعة ومعقدة على مجموعة من الكائنات دون الحاجة إلى تغيير الكود الخاص بهذه الكائنات.

حالات استخدام :

عندما تحتاج إلى إجراء عمليات مختلفة على كائنات دون تغييرها: إضافة وظائف جديدة إلى الكائنات دون تعديلها.

عندما تريد فصل الخوارزميات عن الكائنات: يُساعد في الحفاظ على الكود نظيفًا ومنظمًا من خلال فصل الخوارزميات عن الكائنات التي تستخدمها.

عندما تحتاج إلى إضافة وظائف جديدة بشكل متكرر: يُفيد في البيئات التي تتطلب تحديثات وإضافات مستمرة للوظائف دون الحاجة إلى تغيير الكائنات الموجودة.

مثال : في نظام متجر يبيع منتجين Fruit , Book نريد حساب الضريبة على كل منتج من دون التأثير على الكائنات ، علماً أن حجم الضريبة يختلف باختلاف المنتج :

```
public interface Visitor {
    public double visit(Book book);
    public double visit(Fruit fruit);
}
```

```
public interface visitable {
    public double accept(Visitor visitor);
}
```

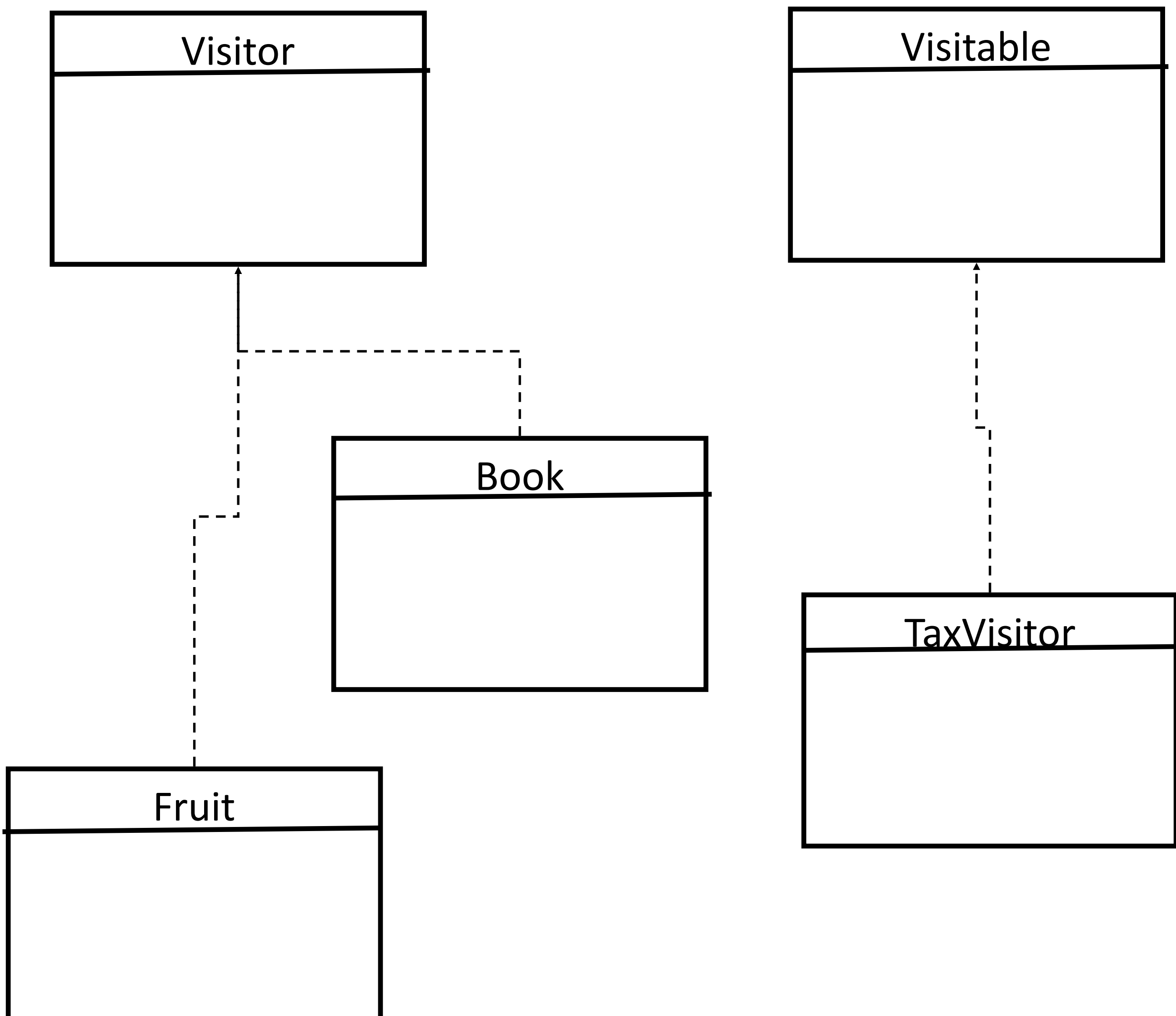
```java
public class Book implements visitable {
    public double price ;
    public Book(double price){
        this.price= price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }
}
```

```java
public class Fruit implements visitable{
    public double price ;
    public Fruit(double price){
        this.price = price;
    }
    @Override
    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }
}
```

```java
public class TaxVisitor implements  Visitor{
    @Override
    public double visit(Book book) {
        return book.price +1 ;
    }

    @Override
    public double visit(Fruit fruit) {
        return fruit.price +2 ;
    }
}
```

```java
public class Main {
    public static void main(String[] args ) {
     Visitor visitor = new TaxVisitor();
     visitable[] xvisitable = {new Book(10) , new Fruit(20)};
     visitable xvisitable2 = new Book(10);
        System.out.println(xvisitable2.accept(visitor));
     for (visitable visitable : xvisitable) {
         System.out.println(visitable.accept(visitor));
     }
    }
}
```

Visitor UML



Observer Pattern

يُسمح بتحديد آلية اشتراك لتنبيه عدة كائنات بأي أحداث تقع للكائن الذي يراقبونه بحيث عندما تتغير حالة كائن ما (الموضوع)، يتم إخطار جميع الكائنات المعتمدة (المراقبين) وتحديثها تلقائيًا.

حالات استخدام :

إدارة الاشتراكات: يُستخدم لتحديد آلية اشتراك تسمح للكائنات بالاشتراك في إشعارات حول أحداث معينة.

التحديثات الديناميكية: يُفيد في الحالات التي تحتاج فيها إلى تحديث عدة كائنات استجابةً لتغيير في كائن آخر.

فصل الوظائف: يُساعد في فصل الكود الذي يدير الكائنات عن الكود الذي يُخطر بالتغييرات، مما يُحسن التنظيم والصيانة.

مثال : في نظام تعليم الكتروني ، يقوم الطلاب بالاشتراك ، أو إلغاء اشتراكهم في مادة معينة يتم إعلام جميع الطلاب المشتركين بتوفر الكورس :

```java
public interface Observer {
    void update(String msg);
}
```

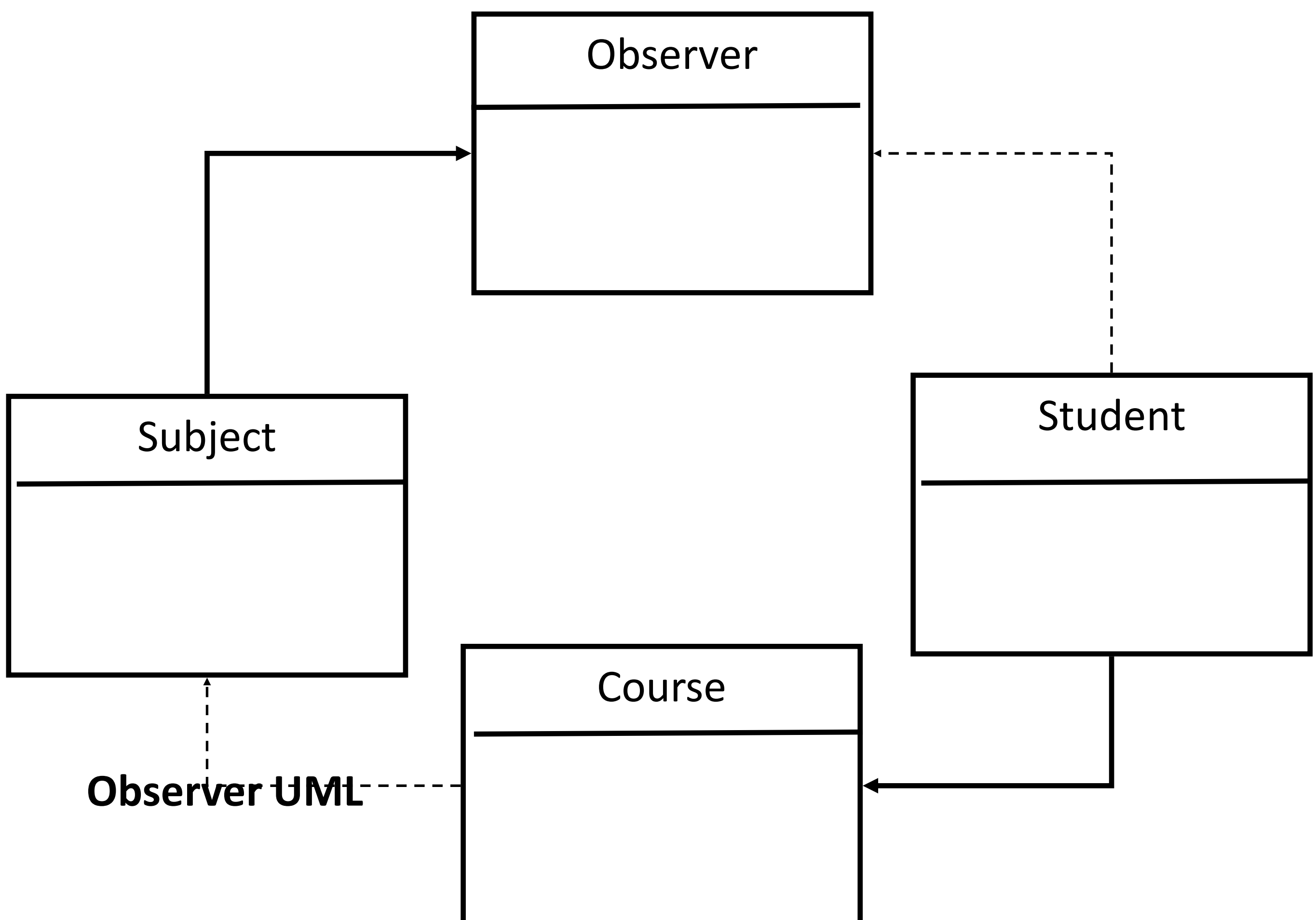
```java
public class Student implements Observer{
   private String name;
    public Student(String name){
        this.name = name;
    }

    @Override
    public void update(String msg) {
        System.out.println(this.name + " has new notification = " + msg);
    }
}
```

```java
public interface Subject {
    void Subscribe (Observer observers);
    void unSubscribe(Observer observers);
    void notifyAllSubscriber();
}
```

```java
public class Course implements Subject{
    private String name;
    private String availability;
    private List<Observer> observers = new ArrayList<>();
    @Override
    public void Subscribe(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void unSubscribe(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyAllSubscriber() {
        for (Observer observer : observers){
            observer.update(availability);
        }
    }
    public void setAvailability(boolean availability){
        this.availability = this.name + (availability ? " Available" : " Not Available");
        notifyAllSubscriber();
    }
    public Course(String name){
        this.name =name;
    }
```

```java
public class Main {
    public static void main(String[] args ) {

        Student sami  = new Student("sami");
        Student sami1  = new Student("sami1");
        Student sami2  = new Student("sami2");
        Student sami3  = new Student("sami3");
        Student sami4  = new Student("sami4");

        Course flutter = new Course("flutter course");

        flutter.Subscribe(sami);
        flutter.Subscribe(sami1);
        flutter.Subscribe(sami2);
        flutter.Subscribe(sami3);
        flutter.Subscribe(sami4);

        flutter.unSubscribe(sami);
        flutter.setAvailability(true);
    }
}
```

Observer UML



Chain Of Responsibility Pattern

يُستخدم لتمرير الطلبات عبر سلسلة من المعالجين (Handlers)، حيث يقرر كل مداوِل ما إذا كان سيعالج الطلب أو يمرره إلى المداوِل التالي في السلسلة .يُمكّن هذا النمط من تقليل الاقتران بين المرسل والمستقبل ويسمح بمرونة أكبر في توزيـع المسؤوليات بين الكائنات.

حالات استخدام :

التحكم في الوصول: يُستخدم لإدارة الوصول إلى نظام معين، مثل تقييد الوصول للمستخدمين الموثقين فقط.

التحقق والتصفية: يُمكن استخدامه لإضافة خطوات تحقق متعددة، مثل التحقق من صحة البيانات وترشيح الطلبات الفاشلة.

المرونة في التعامل مع الطلبات: يُفيد في توفير القدرة على إضافة معالجين جدد أو تغيير ترتيبهم دون تغيير الكود الأساسي.

مثال : لدينا تطبيق للتحقق من صحة روابط ، تتم عملية التحقق على ثلاث مراحل أول مرحلة يتم التحقق من أن الرابط ليس رابط فيسبوك ، ثم ينتقل للمرحلة الثانية و يتحقق من أن الرابط ليس رابط يوتيوب ، ثم ينتقل للمرحلة الثالثة التي تعطي نتيجة افتراضية :

```java
public interface Ihander {
    public void setNextHander(Ihander ihander);
    public String handelLink(String link);
}
```

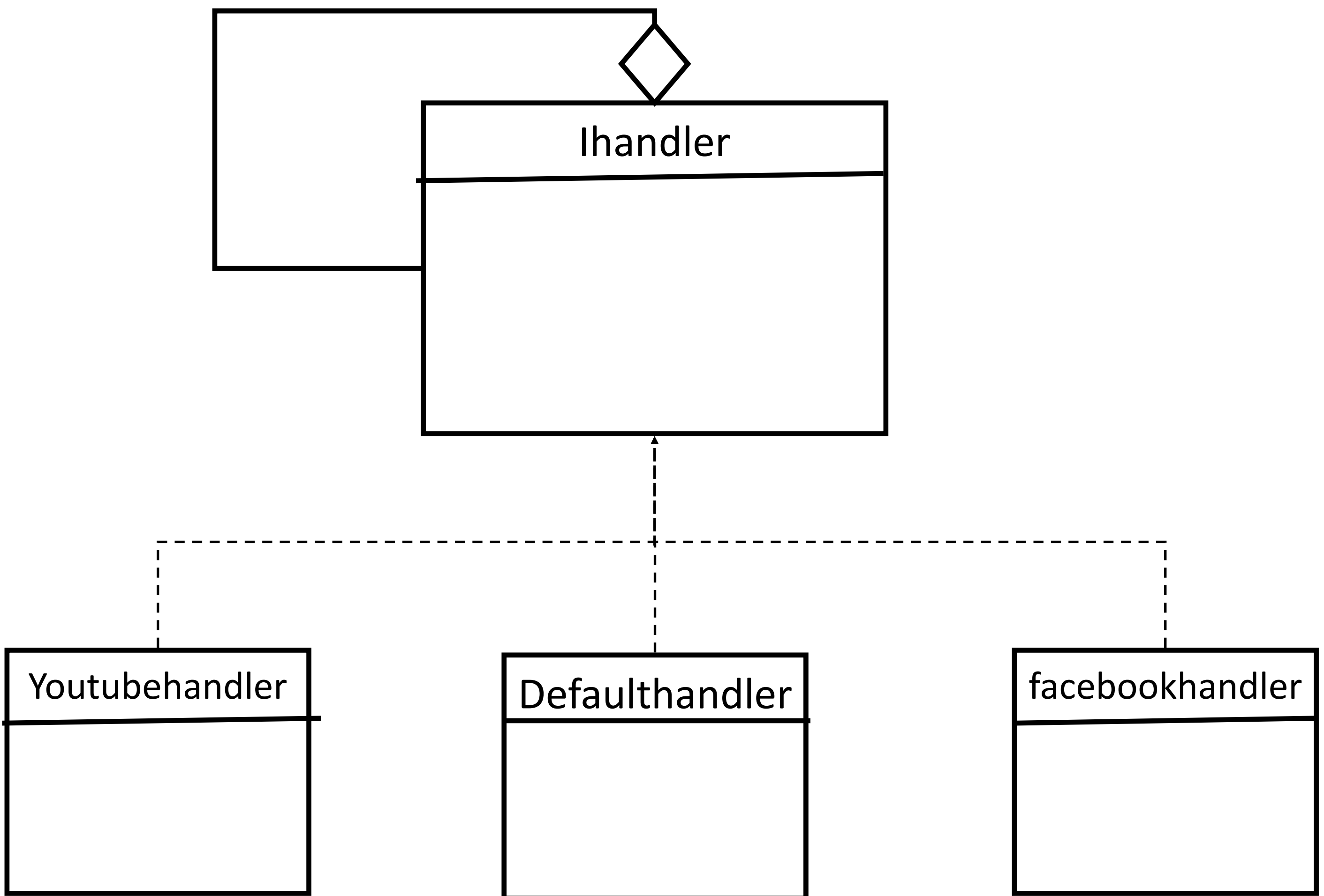
```
public class YouTubeLink implements Ihander{
    private Ihander ihander;
    public YouTubeLink(){}

    @Override
    public void setNextHander(Ihander ihander) {
        this.ihander = ihander;
    }

    @Override
    public String handelLink(String link) {
        if(!link.equals("y")){
            System.out.println("thats not youtube link");
            return this.ihander.handelLink(link);


        }
        System.out.println("this is youtube link");
        return "ok";
    }
}
```

```
public class FacebookLink implements Ihander{
    private Ihander ihander;

    @Override
    public void setNextHander(Ihander ihander) {
        this.ihander = ihander;
    }

    @Override
    public String handelLink(String link) {
        if(!link.equals("f")){
            System.out.println("thats not facebook link");
            return this.ihander.handelLink(link);


        }
        System.out.println("this is facebook link");
        return "ok";
    }

}
```

```java
public class DefaultLink implements Ihander{
    @Override
    public void setNextHander(Ihander ihander) {
        System.out.println("no handler");
    }

    @Override
    public String handelLink(String link) {
        System.out.println("default link not y or f");
        return "default handler";
    }
}
```

```java
public class main {
    public static void main(String[] args ) {
        FacebookLink facebookLink = new FacebookLink();
        YouTubeLink youTubeLink = new YouTubeLink();
        DefaultLink defaultLink = new DefaultLink();
        facebookLink.setNextHander(youTubeLink);
        youTubeLink.setNextHander(defaultLink);
        facebookLink.handelLink("s");
    }
}
```

Chain Of Responsibility UML



Template Pattern

يُستخدم لتحديد هيكل خوارزمية في فئة أساسية ولكنه يسمح للفئات الفرعية بتجاوز خطوات معينة من هذه الخوارزمية دون تغيير هيكلها العام .يُعزز هذا النمط إعادة استخدام الكود من خلال تغليف الهيكل الخوارزمي المشترك في الفئة الأساسية، مع السماح للفئات الفرعية بتوفير تنفيذ محدد لخطوات معينة، مما يُمكّن من التخصيص والمرونة.

حالات استخدام :

عندما تريد توحيد وتنظيم خوارزمية: يُستخدم لتحديد الخطوات الأساسية لخوارزمية وترك التفاصيل للفئات الفرعية لتنفيذها.

عندما تحتاج إلى توفير هيكل مرن للخوارزميات: يُمكن للفئات الفرعية تغيير بعض الخطوات دون التأثير على الهيكل العام للخوارزمية.

عندما ترغب في تجنب تكرار الكود: يُساعد في تقليل التكرار من خلال توحيد الخطوات المشتركة في فئة أساسية.

مثال : في معمل صناعة مركبات ، يقوم بتصنيع سيارات ، وشاحنات ، تتم عملية البناء على مرحلتين ، المرحلة الأولى هي التجميع و الثانية هي التشغيل ، تختلف عملية التجميع باختلاف المركبة ، بينما عملية التشغيل تكون متطابقة مع جميع المركبات :

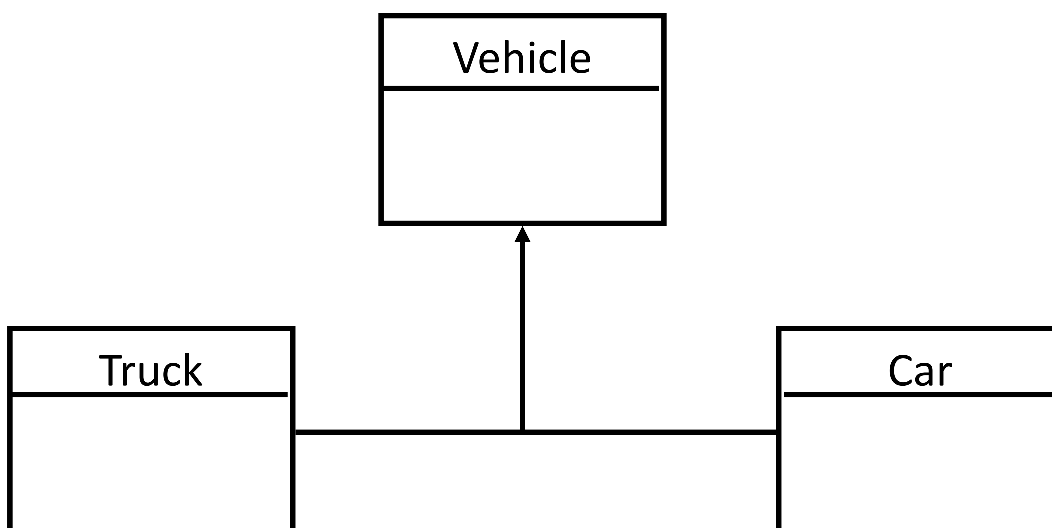
```java
public abstract class Vehicle {
    public final void build(){
        assemble();
        start();
    }
    protected abstract void assemble();
    public void start(){
        System.out.println("start the vehicle");
    }
}
```

```java
public class Car extends Vehicle{
    @Override
    protected void assemble() {
        System.out.println("car assembled");
    }

}
```

```java
public class Truck  extends Vehicle{
    @Override
    protected void assemble() {
        System.out.println("Truck assembled");
    }
}
```

```java
public class main {
    public static void main(String[] args ) {
        Vehicle car = new Car();
        car.build();
        Vehicle Truck = new Truck();
        Truck.build();
    }
}
```

Template UML



Strategy Pattern

يُمكّن من تعريف عائلة من الخوارزميات، وتغليف كل خوارزمية في فئة منفصلة، وجعل الخوارزميات قابلة للتبادل داخل الكائنات .يسمح هذا النمط بتغيير سلوك الكائن ديناميكيًا أثناء تشغيل البرنامج، مما يُتيح المرونة في اختيار الخوارزمية المناسبة.

حالات استخدام :

تغيير السلوك ديناميكيًا: يُستخدم عندما نحتاج إلى تغيير سلوك الكائن أثناء تشغيل البرنامج.

تجنب الشيفرة الشرطية: يُفيد في تقليل الشيفرة الشرطية المعقدة والمتعددة التي تعتمد على نوع الخوارزمية.

إضافة خوارزميات جديدة: يُسهل إضافة خوارزميات جديدة دون تغيير الكود القائم.

فصل الخوارزميات عن العملاء: يُمكن العملاء من التفاعل مع الكائنات دون معرفة الخوارزميات المستخدمة.

مثال : لعبة قتال يمكن للاعب اختيار مقاتلين مختلفين، كل منهم لديه أساليب قتالية متنوعة مثل الهجوم العنيف، أو الدفاع الصلب، في بداية اللعبة، يمكن للاعب اختيار أسلوب القتال الذي يفضله، ولكن خلال المعركة، يمكنه تغيير استراتيجيته بناءً على تحركات الخصم :

```
public interface Ibehaviour {
    public void moveCommand();
}
```

```java
public class Context {
    protected Ibehaviour ibehaviour;
    public Context(Ibehaviour ibehaviour){
        this.ibehaviour=ibehaviour;
    }

    public Context() {
    }

    public void moveC(){
        ibehaviour.moveCommand();
    }
}
```

```java
public class AggersiveBehaviour implements Ibehaviour{
    @Override
    public void moveCommand() {
        System.out.println("Attack");
    }
}
```

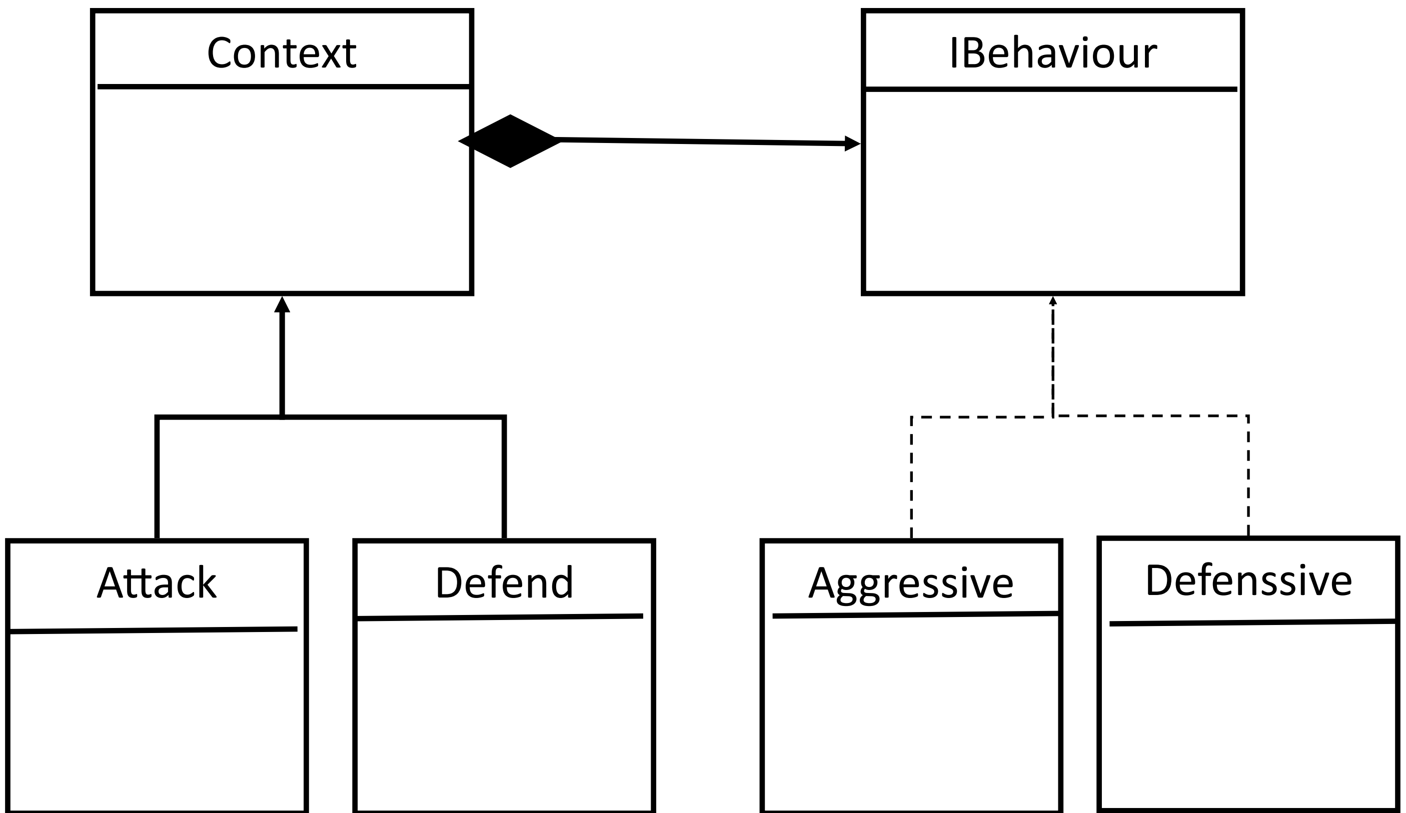
```java
public class DefeciveBeh implements Ibehaviour{
    @Override
    public void moveCommand() {
        System.out.println("Defend");
    }
}
```

```
public class Attack extends Context{
    public Attack() {
        ibehaviour  = new AggersiveBehaviour();

    }


}
```

```
public class Defend extends Context{
    public Defend(){
        ibehaviour = new DefeciveBeh();
    }
}
```

```
public class Main {
    public static void main(String[] args ) {

        Context context = new Context();
        System.out.println("enemy up ahead");
        context = new Attack();
        context.moveC();
        System.out.println("enemy attacking us");
        context = new Defend();
        context.moveC();

    }
}
```

Strategy UML



Command Pattern

يتيح تحويل الطلبات إلى كائنات مستقلة تحتوي على كل المعلومات اللازمة لتنفيذ الطلب.

هذا يعني أنه يمكن تخزين أو تنفيذ الطلبات في أوقات مختلفة.

حالات استخدام :

إضافة أنواع جديدة من الأوامر: يمكن إضافة أنواع جديدة من الأوامر دون الحاجة إلى تغيير الكود القائم.

تخزين الأوامر لتنفيذها لاحقًا: يمكن تخزين الأوامر في قائمة انتظار لتنفيذها في وقت لاحق.

التراجع عن الأوامر: يمكن تنفيذ عملية التراجع إذا كانت الأوامر تحتوي على معلومات حول كيفية القيام بذلك.

تسلسل الأوامر: يمكن تنفيذ الأوامر بشكل متسلسل أو متوازي، وهذا يعتمد على قدرات اللغة البرمجية المستخدمة.

مثال : يوجد لدينا Remote Control مقسم لعدة Slot كل واحدة تتحكم بجهاز معين في Slot 0 المتحكم بجهاز التلفاز يحتوي على زرين On , Off ، يقوم التلفاز بعمليتين turnOn , trunOff :

```java
public interface Command {
    void execute();
}
```

```java
public class Television {

    public void turnOn() {
        System.out.println("TV is ON");
    }
    public void turnOff(){
        System.out.println("TV is OFF");
    }
}
```

```java
public class TurnOnTelevision implements Command {

    private Television tv;

    public TurnOnTelevision(Television tv) {
        this.tv = tv;
    }

    @Override
    public void execute() {
    this.tv.turnOn();
    }
}
```

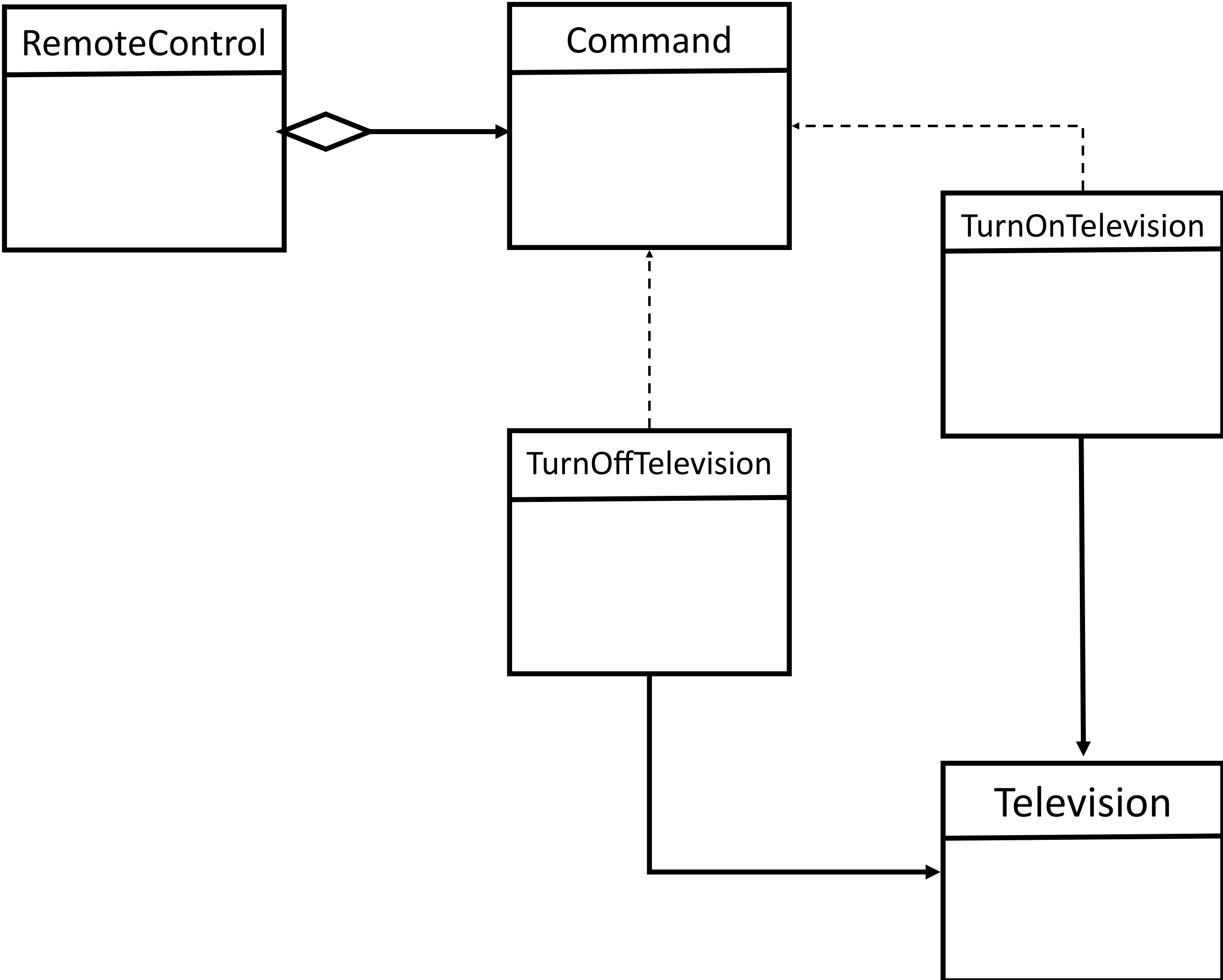
```java
public class TurnOffTelevision implements Command {
    private Television tv;

    public TurnOffTelevision(Television tv) {
        this.tv = tv;
    }

    @Override
    public void execute() {
        this.tv.turnOff();
    }
}
```

```java
public class RemoteControl {

    private Command[] onCommand;
    private Command[] offCommand;

    public RemoteControl(){
        onCommand = new Command[3];
        offCommand = new Command[3];
    }

    public void addCommand(int slot, Command onCommand, Command offCommand){
        this.onCommand[slot] = onCommand;
        this.offCommand[slot]= offCommand;
    }

    public void onButtonPressed(int slot){
        onCommand[slot].execute();
    }
    public void offButtonPressed(int slot){
        offCommand[slot].execute();
    }
}
```

عدد ال Slots

```java
    final static int TV_SLOT = 0;

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();


        Television tv = new Television();

        TurnOnTelevision turnOnTelevision = new TurnOnTelevision(tv);
        TurnOffTelevision turnOffTelevision = new TurnOffTelevision(tv);

        remoteControl.addCommand(TV_SLOT,turnOnTelevision,turnOffTelevision);

        remoteControl.onButtonPressed(TV_SLOT);
      // remoteControl.offButtonPressed(TV_SLOT);
    }
}
```

Command UML



Real time systems Additional Section

مقدمة إلى أنظمة الزمن الحقيقي

تعتبر أنظمة الزمن الحقيقي من أهم مجالات هندسة البرمجيات. إنها تتعامل مع تصميم وتطوير البرمجيات التي تحتاج إلى استجابة فورية للأحداث في الوقت الحقيقي. سنستعرض بعض المفاهيم الأساسية حول هذا الموضوع:

ما هي أنظمة الزمن الحقيقي؟

أنظمة الزمن الحقيقي هي الأنظمة التي تتفاعل مع البيئة في الوقت الفعلي. يعني ذلك أنها تتعامل مع الأحداث والمهام بشكل فوري دون تأخير. مثلًا، أنظمة التحكم في المصاعد وأنظمة مراقبة الطاقة، والألعاب الفيديو، والتطبيقات الطبية.

تحديات أنظمة الزمن الحقيقي :

- **الاستجابة الفورية** : يجب أن تكون النظام قادرًا على التفاعل مباشرة مع الأحداث.
- **التزامن والتوازن** : يجب أن يتم التنسيق بين المهام المتعددة بشكل صحيح.
- **الاعتمادية** : يجب أن يكون النظام موثوقًا وقادرًا على التعامل مع الأخطاء.

أنواع أنظمة الزمن الحقيقي

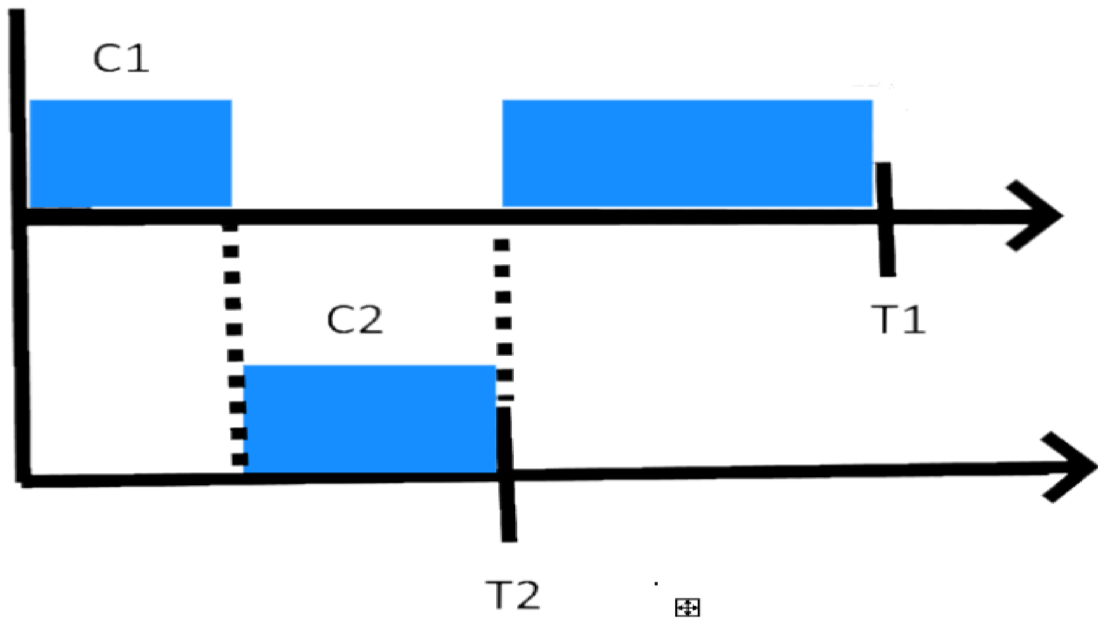
هناك نوعان رئيسيان من أنظمة الزمن الحقيقي:

1. **النظم الصلبة (Hard Real-Time Systems)** : تتطلب استجابة فورية ومحددة للأحداث. مثل أنظمة التحكم في الطائرات.
2. **النظم الناعمة (Soft Real-Time Systems)** : تتطلب استجابة سريعة، ولكن يمكن التسامح مع بعض التأخير. مثل أنظمة الصوت والفيديو.

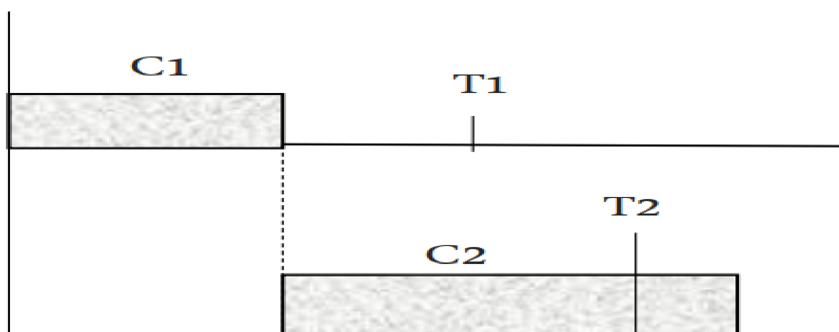
ملاحظة : ليس من الضروري أن يكون نظام الزمن الحقيقي سريعاً ، و إنما يجب أن يتم تنفيذ مهامه بحيث يحقق متطلبات التوقيت المتزامن أي أنه من أجل كل مهمة أو حدث يجب أن يتم اتمامها عند الزمن **T** بحيث (**T**max =< **T** <= **T**min)

المهام المتسايرة :

تدعى المهام التي تنفذ بشكل متساير بالمهام المتسايرة ، ليكن لدينا المهمتين P1 , P2 زمن معالجة كل منهما هو C1 , C2 دور تنفيذهما هو T1 , T2 نفترض أن P2 ذات أولوية أعلى من P1، نلاحظ أنه عندما تعمل P2 لا تعمل P1 و ذلك لأن أنظمة الزمن الحقيقي هي أنظمة Primitive schedule (أي في حال ورود مهمة ذات أولوية أعلى ستقاطع المهمة ذات الأولوية الأدنى و ستعمل بدلاً منها) .

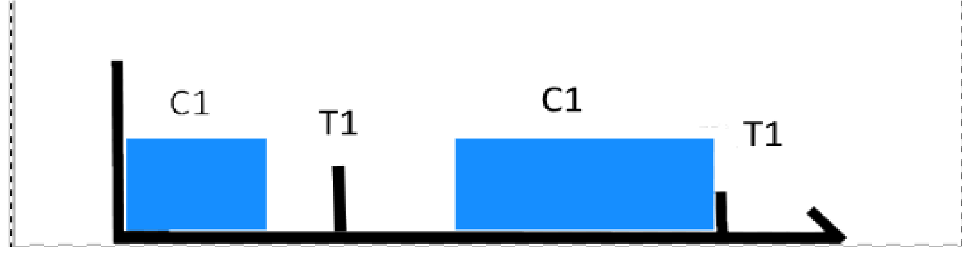


فالشرط الأساسي لكون النظام نظام زمن حقيقي هو تحقيق جميع فعالياته (مهماته) لمحدداتها الزمنية (أي تتم من دون تجاوز الدور المخصص لها)



(مثال لمهمتين ضمن نظام لا يمثل نظام زمن حقيقي)

الشرط المخصص : النظام هو نظام زمن حقيقي إذا كانت كل فعالياته قد حققت موعدها الأول .



أي أن المهمة حققت الموعد الأول T1 (انتهت من المعالجة من دون تجاوز أول دور)

في حال عدم تحقيق المهام لموعدها الأول نعمل على تصحيح الخلل من خلال :

١. توسيع الدور T

٢. تقليل زمن المعالجة C

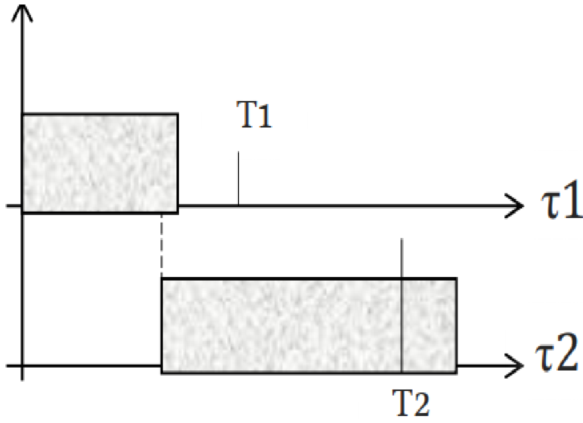
٣. في حال لم نتمكن من توسيع الدور أو تقليل زمن المعالجة نتجه للبرمجة التفرعية (يتم توزيع المهام على عدة معالجات مع مراعاة حساب زمن الاتصال بين الفعاليات)

المعادلات الرياضية لنظام الزمن الحقيقي

الشروط اللازمة ليكون النظام نظام زمن حقيقي :

الشرط الأول : (لازم غير كافي)

زمن معالجة كل مهمة أصغر من دورها **Ci <= Ti** حيث **i** هي رقم المهمة .



مثال: ليكن لدينا المهمتين τ2, τ1

	P	T	C
τ1	1	5	3
τ2	2	10	8

حيث *p* هي *priority* (أولوية المهمة)

لاحظ أن الشرط محقق : **C1 = 3 < T1 = 5 && C2 = 8 < T2= 10**

و مع ذلك فالنظام ليس نظام زمن حقيقي حيث أن المهمة t2 تبدأ بعد انتهاء t1 بسبب الأولوية P فيكون زمن حساب t2 هو C2 = 8 + 3 = 11 > T2 = 10 نلاحظ تجاوز زمن معالجة المهمة للدور المخصص لها فالنظام ليس نظام زمن حقيقي و منه فالشرط لازم لكنه غير كافي للإثبات .

الحمولة الوسطية للنظام : ∑ Ci/Ti

لحساب الحمولة الوسطية للنظام السابق :

∑ Ci/Ti = 3/5 + 8/10

∑ Ci/Ti = 6/10 + 8/10

∑ Ci/Ti = 14/10

∑ Ci/Ti > 1

فلا يكون النظام نظام زمن حقيقي اذا تجاوزت حمولته الوسطية الواحد .

الشرط الثاني : (لازم غير كافي)

$$\sum Ci/Ti \le 1$$

من شروط نظم الزمن الحقيقي أن تكون حمولة النظام الوسطية أصغر أو تساوي الواحد .

مثال: ليكن لدينا المهام $\tau1, \tau2, \tau3$

	P	T	C
$\tau1$	1	6	3
$\tau2$	2	9	2
$\tau3$	3	11	2

نقوم بحساب الحمولة الوسطية للنظام :

$$\sum Ci/Ti = 3/6 + 2/9 + 2/11$$

$$\sum Ci/Ti = 3/6 + 2/9 + 2/11$$

$$\sum Ci/Ti = 9/18 + 4/18 + 2/11$$

$$\sum Ci/Ti = 13/18 + 2/11$$

$$\sum Ci/Ti = 143/198 + 36/198$$

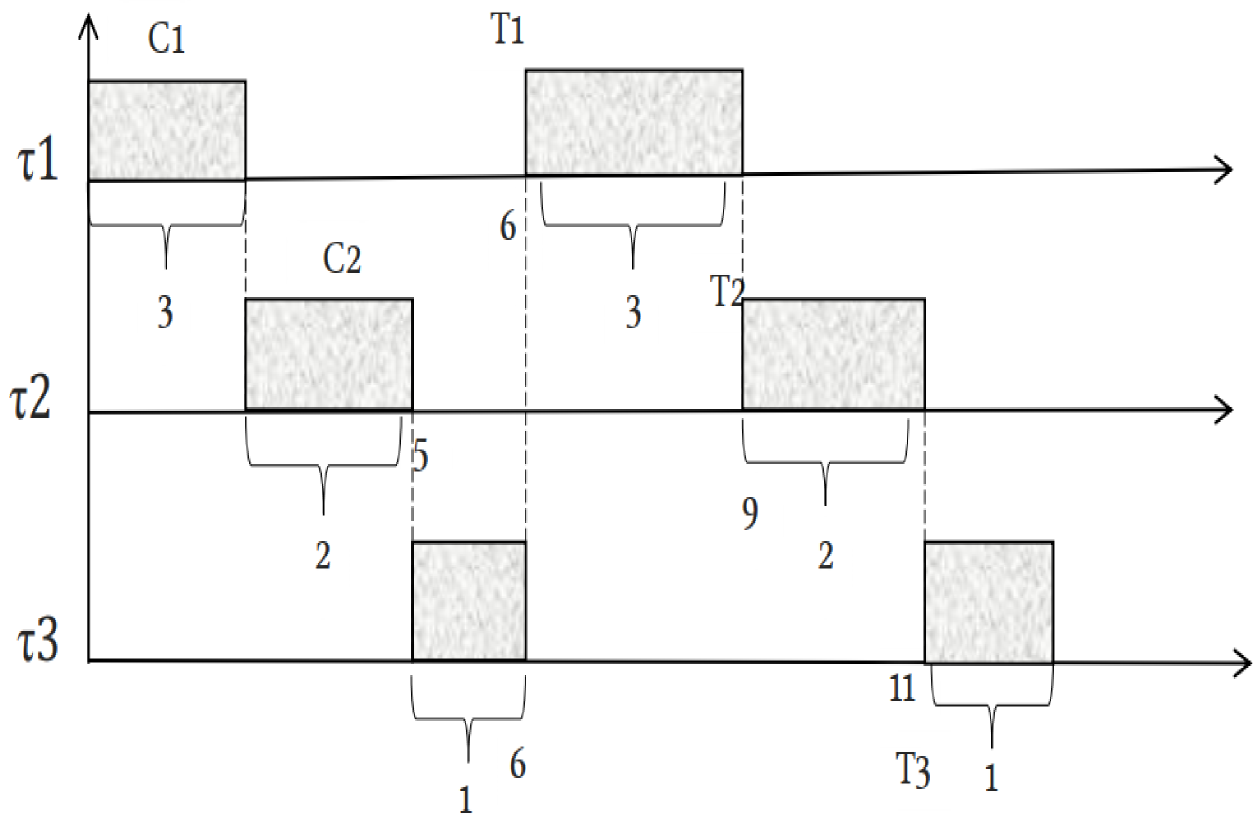
$$\sum Ci/Ti = 179/198 < 1$$

الشرط محقق !

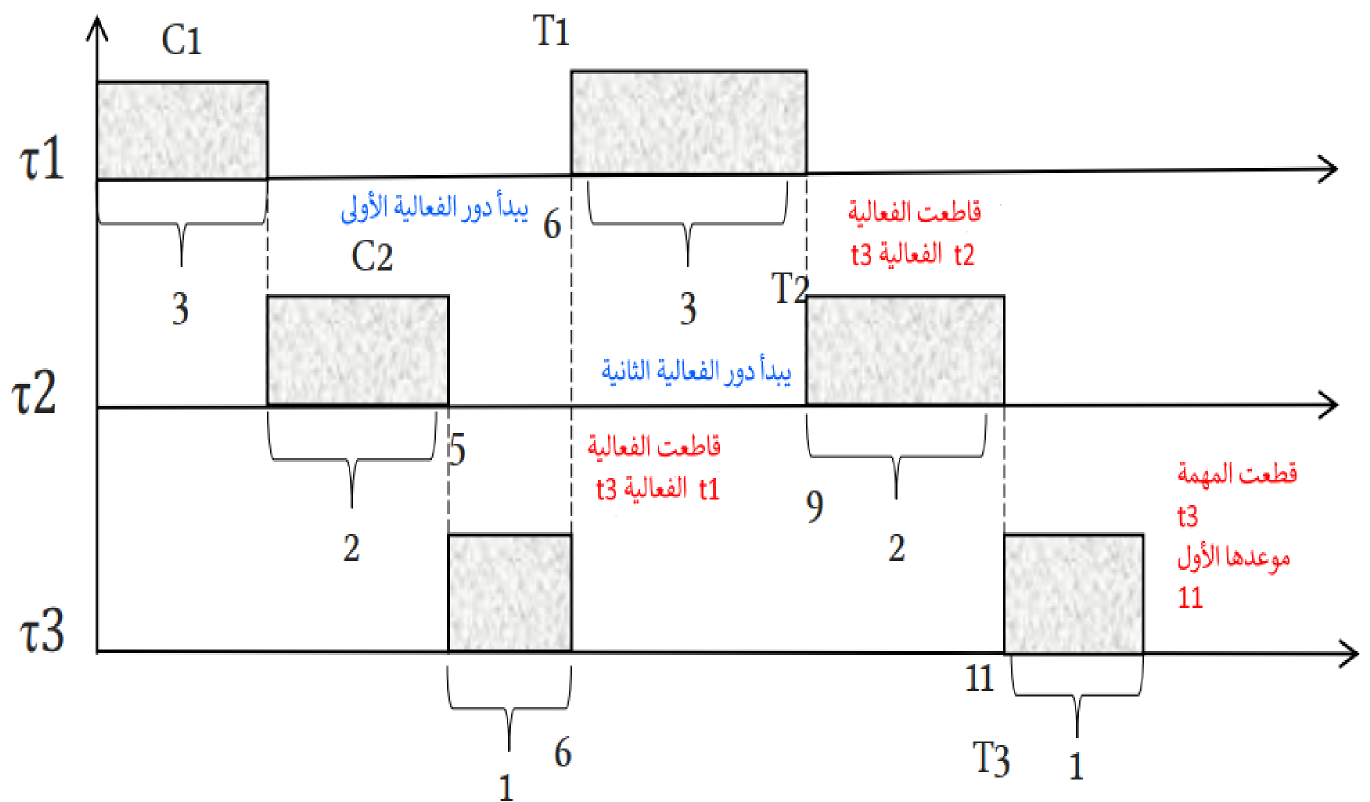
إن الشرط الثاني لازم لكنه غير كافي لإثبات أن النظام هو نظام زمن حقيقي فالنظام السابق **ليس نظام زمن حقيقي** لأن الفعالية t3 فشلت بتحقيق موعدها الأول ، دعونا نفحص الأمور بشكل أكثر تفصيلاً :

- **المهمة الأولى :(t1)**
- تنتهي من التنفيذ في زمن مقداره 3 وتحقق موعدها الأول (3 < 6).
- **المهمة الثانية :(t2)**
- تبدأ بعد المهمة الأولى وتنتهي في زمن مضاف إليه زمن المهمة السابقة (2 + 3 < 9).
- تحقق موعدها الأول أيضًا.
- **المهمة الثالثة :(t3)**
- تبدأ في العمل بعد مرور ثانية من الزمن.
- يكون زمن التنفيذ مساويًا لأحد مضاعفات دور المهمة الأولى (6) فيدخل دور المهمة الأولى .
- باعتبار أن المهمة الأولى ذات أولوية أعلى من المهمة الثالثة، تقاطع عملهما قبل انتهائها.
- تنقذ المهمة الأولى بزمن قدره 3، ليصبح زمن التنفيذ 6 + 3= 9.
- هذا الزمن يعد من مضاعفات العدد 9 فيدخل دور المهمة الثانية .
- **استكمال المهمة الثانية :(t2)**
- باعتبار المهمة الثانية ذات أولوية أعلى من الثالثة تقاطعها أيضاً فتبدأ المهمة الثانية بالتنفيذ مرة أخرى قبل انقضاء المهمة الثالثة.
- تنفذ مهمتها بزمن مقداره 2.
- **استكمال المهمة الثالثة :(t3)**
- تعود المهمة الثالثة لإكمال تنفيذها.
- يكون زمن بداية التنفيذ 9 + 2 = 11.
- بعد نهاية التنفيذ، يضاف إليه الثانية المتبقية ليصبح الزمن 11 + 1 = 12.
- هذا الزمن أكبر من الدور الأول للمهمة الثالثة (12> 11).
- لذلك، المهمة الثالثة تجاوزت موعدها الأول.

بهذا، يمكن القول إن النظام السابق لا يعتبر نظام زمن حقيقي.



C1
T1
τ1
6
3
3
C2
T2
τ2
5
9
2
2
τ3
11
6
1
T3
1



C1
T1
τ1
يبدأ دور الفعالية الأولى
6
قاطعة الفعالية
t2 الفعالية t3
C2
3
3
T2
يبدأ دور الفعالية الثانية
τ2
5
قاطعة الفعالية
t1 الفعالية t3
9
2
2
قطعت المهمة
t3
موعدها الأول
11
τ3
11
6
1
T3
1

الشرط الثالث : (لازم غير كافي)

$$\sum_{j=1}^{i-1}\left(\frac{Ti}{Tj}\times Cj\right)\le Ti-Ci$$

حيث يرمز j إلى الفعاليات ذات الأولوية الأعلى من i

حيث يرمز $\frac{Ti}{Tj}$ إلى عدد المرات التي تقطع فيها الفعالية j الفعالية i

مثال: ليكن لدينا المهام $\tau1, \tau2, \tau3$

	P	T	C
$\tau1$	1	6	3
$\tau2$	2	9	2
$\tau3$	3	11	2

لحساب كم مرة تقطع الفعالية t1 الفعالية t2

T2 / T1 = 9/6

T2/T1 =~ 2

نقوم بتقريب الناتج لأكبر عدد صحيح و نستنتج أن الفعالية T1 تقطع الفعالية T2 مرتين .

لنتأكد من صحة الجواب :

إن زمن تنفيذ الفعالية t2 يبدأ من المجال من [0,9] ، فعند النقطة الزمنية 0 تبدأ الفعالية t1 وهذه أول مرة تقطع فيها الفعالية t2 ، و عند النقطة الزمنية 6 المساوية لأحد مضاعفات دور الفعالية الأولى الواقعة ضمن المجال من [0,9] تقطعها للمرة الثانية و لا تقطعها مرة أخرى ، و ذلك لأن المضاعف التالي لدور الفعالية الأولى هو 12 و هو يقع خارج مجال زمن الفعالية الثانية .

لنقم الآن بتطبيق الشرط الثالث لاختبار كون النظام نظام زمن حقيقي :

$$\sum_{j=1}^{i-1} \left(\frac{Ti}{Tj} \times Cj\right) \leq Ti - Ci$$

نقوم بمعالجة جميع المقاطعات

نبدأ من الفعالية t2 :

T2 / T1 = 9/6

T2/T1 =~ 2

2 * 3 ≤ 9 – 2

6 ≤ 8

محقق

الفعالية التالية t3 :

T3 / T1 = 11/6

T3/T1 =~ 2

T3/T1 * C1 = 2*3 = **6**

بالنسبة لتقاطعها مع الفعالية الثانية

T3 / T2 = 11/9

T2/T1 =~ 2

T3/T2 * C2 = 2*2 =**4**

نحسب

T3 - C3 = 11 - 2 = **9**

فيكون مقدار التأخير

6 + 4 > 9

فالنظام خالف الشرط و منه فهو ليس نظام زمن حقيقي

إن الشروط الثلاثة السابقة هي شروط لازمة لكنها غير كافية أي أن تحققها لا يكفي لإثبات أن النظام هو نظام زمن حقيقي ، لكن ، و بالمقابل إن مخالفة إحداها يعني بشكل جازم أن النظام لا يعد نظام زمن حقيقي .

الشرط الكافي : (غير اللازم)

$$n(2^{1/n}-1) \geq \sum_{i=1}^{n} \frac{Ci}{Ti}$$

حيث n هو عدد الفعاليات، أي إذا كان مثلاً عدد الفعاليات 3 ففي حال كان $3(2^{1/3}-1)$ أكبر أو تساوي الحمولة الوسطية للنظام عندها يكون النظام نظام زمن حقيقي.

إن تحقق الشرط **يعني أن النظام حتماً** هو نظام زمن حقيقي ، ولكن **عدم تحققه لا يعني العكس،** فقد يكون النظام نظام زمن حقيقي فلذلك هو غير لازم .

الشرط الأكيد :

$$Ri^{n+1} = Ci + \sum_{j \in hp} \left\lceil \frac{Ri^{n}}{Tj} \right\rceil \times Cj$$

زمن الاستجابة الحرج

زمن الحسابات

- إن زمن الاستجابة الحرج هو زمن الإنهاء الفعلي .
- تسمى هذه المعادلة معادلة تراجعية لوجود Ri في طرفيها .
- Ri^{n} هو الزمن الحرج في المرحلة السابقة .
- يتغير الشرط في حال وجود اعتمادية بين الفعاليات .
- الاعتمادية بين الفعاليات تعني أن الفعالية X لا تتنفذ حتى انتهاء تنفيذ الفعالية Y المعتمدة عليها .

مثال : هل النظام التالي يمثل نظام زمن حقيقي ؟

	P	T	C
τ1	1	10	1
τ2	2	12	2
τ3	3	30	8
τ4	4	60	20

الحل :

١- الشرط الأول محقق إذ أن زمن معالجة جميع الفعاليات أصغر من الدور المخصص لها .

٢- نقوم بحساب الحمولة الوسطية للنظام **$\sum Ci/Ti$**

$$\sum Ci/Ti = 1/10 + 2/12 + 8/30 + 20/60$$

$$\sum Ci/Ti = 0.86 \leq 1$$

الشرط الثاني محقق

٣- نقوم بحساب زمن الاستجابة الحرج للفعاليات :

حساب زمن الاستجابة الحرج للفعالية الأولى

$$R1 = 1$$

إن زمن الاستجابة الحرج لأول فعالية

هو زمن حسابها لأنها الأعلى أولوية

وبالتالي لا يوجد مقاطعة من قبل فعالية ثانية

$R1 < T1$ نكمل حساب زمن الاستجابة لباقي الفعاليات

نقوم بحساب زمن الاستجابة الحرج للفعالية الثانية :

$$R2 = R([0,2[) = (1*1)$$

زمن البداية

زمن معالجة الفعالية الثانية

عدد مرات ورود الفعالية الأولى ضمن المجال

زمن معالجة الفعالية الأولى

نلاحظ ورود الفعالية الأولى ضمن المجال R([0,2[) مرة واحدة عند ال 0 (نقطة البداية والتي ترد عندها جميع الفعاليات) .

بما أن عدد مرات الورود لا يساوي الصفر نقوم بجمع قيمة زمن الاستجابة الحرج مع أخر قيمة ضمن المجال أي 1+ 2 ، ليصبح المجال الجديد R([2,3[)

$$R2 = R([2,3[) = (0*1)$$

أخر قيمة ضمن المجال السابق

أخر قيمة ضمن المجال مضفاً إليها قيمة زمن الاستجابة الحرج السابق

عدد مرات ورود الفعالية الأولى ضمن المجال

زمن تنفيذ الفعالية الأولى

لم ترد الفعالية الأولى في المجال R([2,3[) إذ أن المجال لا يحتوي على 0 أو على أحد مضاعفات دور الفعالية الأولى 10 .

و باعتبار أن الفعالية لم ترد مرة أخرى ، فتكون قيمة زمن الاستجابة الحرج الخاص بها مساوياً للقيمة ضمن المجال المفتوح $R2 = 3$

$R2 < T2$ نكمل حساب زمن الاستجابة لباقي الفعاليات

$$R3 = R([0,8[) = (1*1) + (1*2)$$

$$R3 = R([8,11[) = (1*1)$$

نلاحظ ورود أحد مضاعفات دور الفعالية الأولى 10 ضمن المجال $R([8,19[)$.

$$R3 = R([11,12[) = 0$$

$$R3 = 12$$

وبما أن $R3 < T3$ نتابع حساب زمن الاستجابة الحرج لباقي الفعاليات .

$$R4 = R([0,20[) = (2*1) + (2*2) + (1*8)$$

نلاحظ ورود الفعالية الأولى مرتين ضمن المجال $R([0,20[)$ عند (0 ,10).

نلاحظ ورود الفعالية الثانية مرتين ضمن المجال $R([0,20[)$ عند (0 ,12).

نلاحظ ورود الفعالية الثالثة مرة ضمن المجال $R([0,20[)$ عند (0).

$$R4 = R([20,34[) = (2*1) + (1*2) + (1*8)$$

نلاحظ ورود الفعالية الأولى مرتين ضمن المجال $R([20,34[)$ عند (20 , 30) مضاعفات دورها الأصلي 10 .

نلاحظ ورود الفعالية الثانية مرة ضمن المجال $R([20,34[)$ عند (24) مضاعفات دورها الأصلي 12.

نلاحظ ورود الفعالية الثالثة مرة ضمن المجال $R([20,34[)$ عند (30) مضاعفات دورها الأصلي 30.

$$R4 = R([34,46[) = (1*1) + (1*2)$$

نلاحظ ورود الفعالية الأولى مرة ضمن المجال R([34,46[) عند (40) مضاعفات دورها الأصلي 10 .

نلاحظ ورود الفعالية الثانية مرة ضمن المجال R([34,46[) عند (36) مضاعفات دورها الأصلي 12.

$$R4 = R([46,49[) = (1*2)$$

نلاحظ ورود الفعالية الثانية مرة ضمن المجال R([46,49[) عند (48) مضاعفات دورها الأصلي 12 .

$$R4 = R([49,51[) = (1*1)$$

نلاحظ ورود الفعالية الأولى مرة ضمن المجال R([49,51[) عند (50) مضاعفات دورها الأصلي 10.

$$R4 = R([51,52[) = 0$$

و منه يكون زمن الاستجابة الحرج للفعالية الرابعة $R4 = 52$

<== $R4 < T4$ ، و بما أن جميع المهام تحقق موعدها الأول فهذا يعني أن النظام هو نظام زمن حقيقي .

المراجع

https://en.wikipedia.org/wiki/Design_Patterns

https://wiki.hsoub.com/Design_Patterns

https://www.baeldung.com/design-patterns-series

دروس مفيدة

https://www.youtube.com/playlist?list=PLZeX1aIDYSn8qNDPcqwyvOtGZkMQwPjCg

https://www.youtube.com/watch?v=yAqyWnUReis&list=PLbw0NwfOlr_xT3L0xnEr1c3KH4KzDHtXf&pp=iAQB

The End