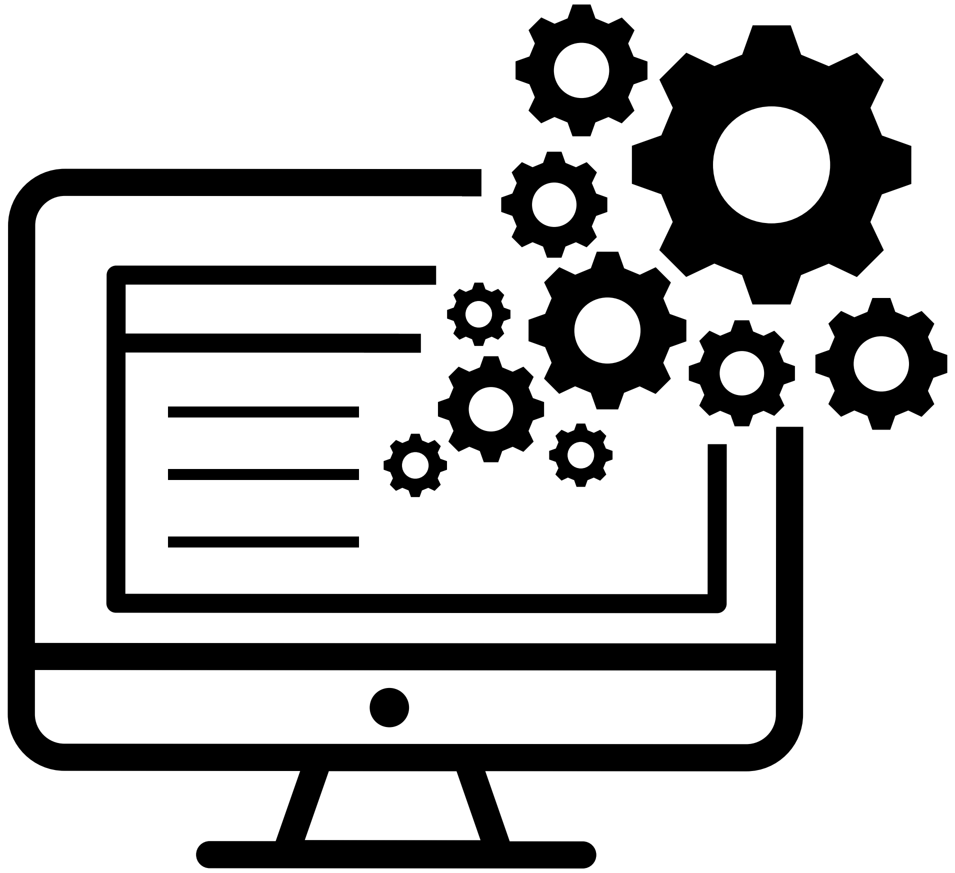
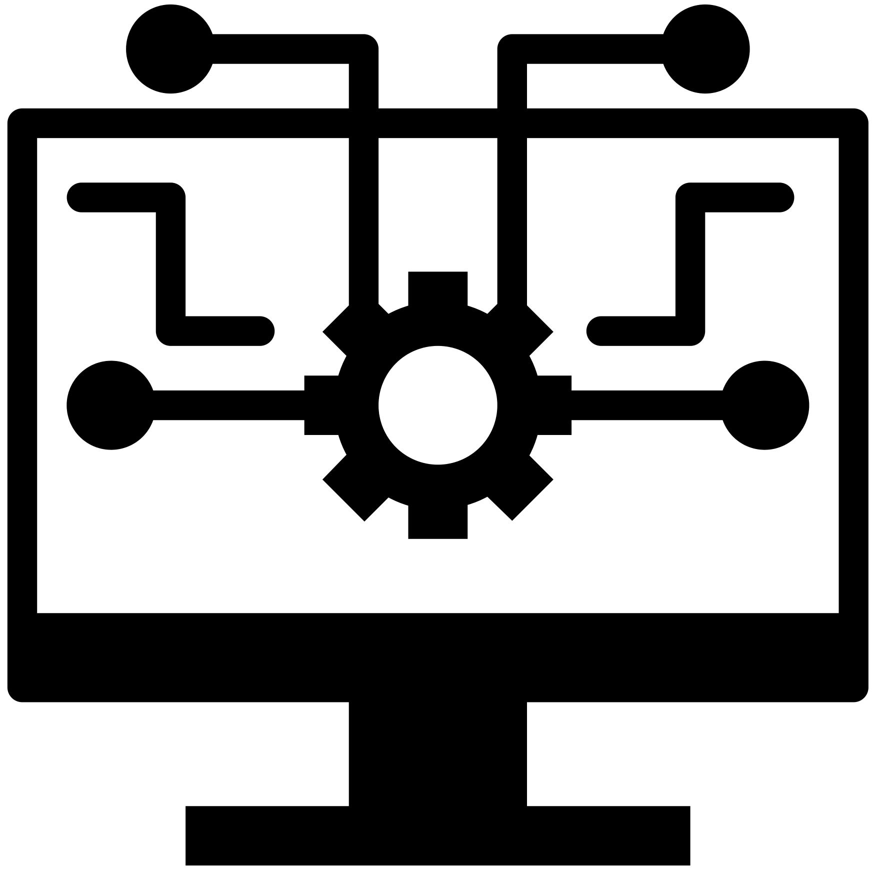
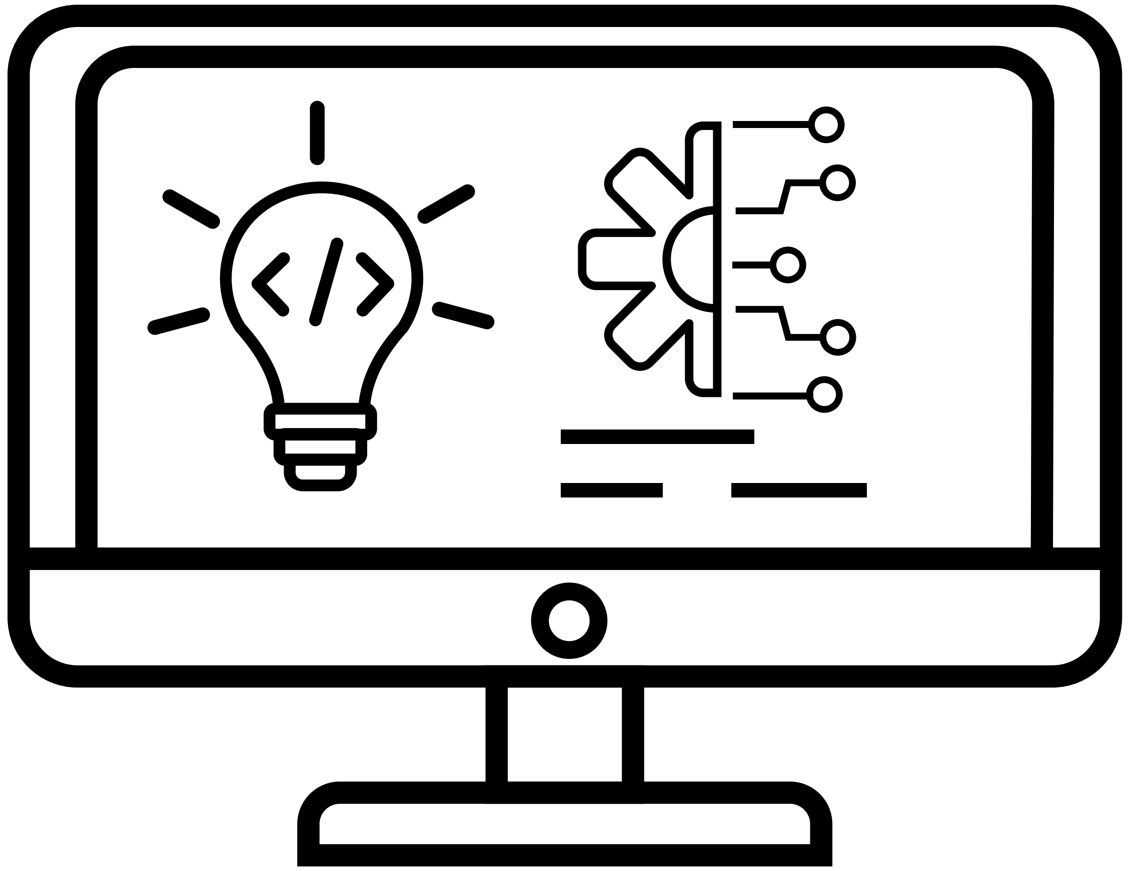




Software Design and Architecture

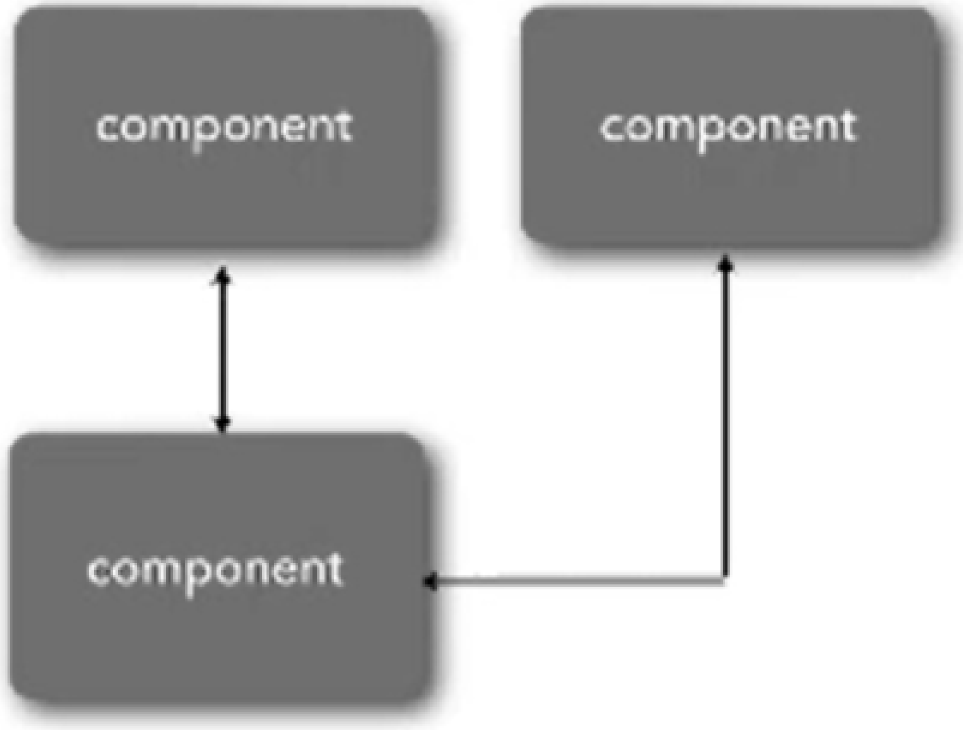
Lab 2

Task 1

Component and Service Coupling

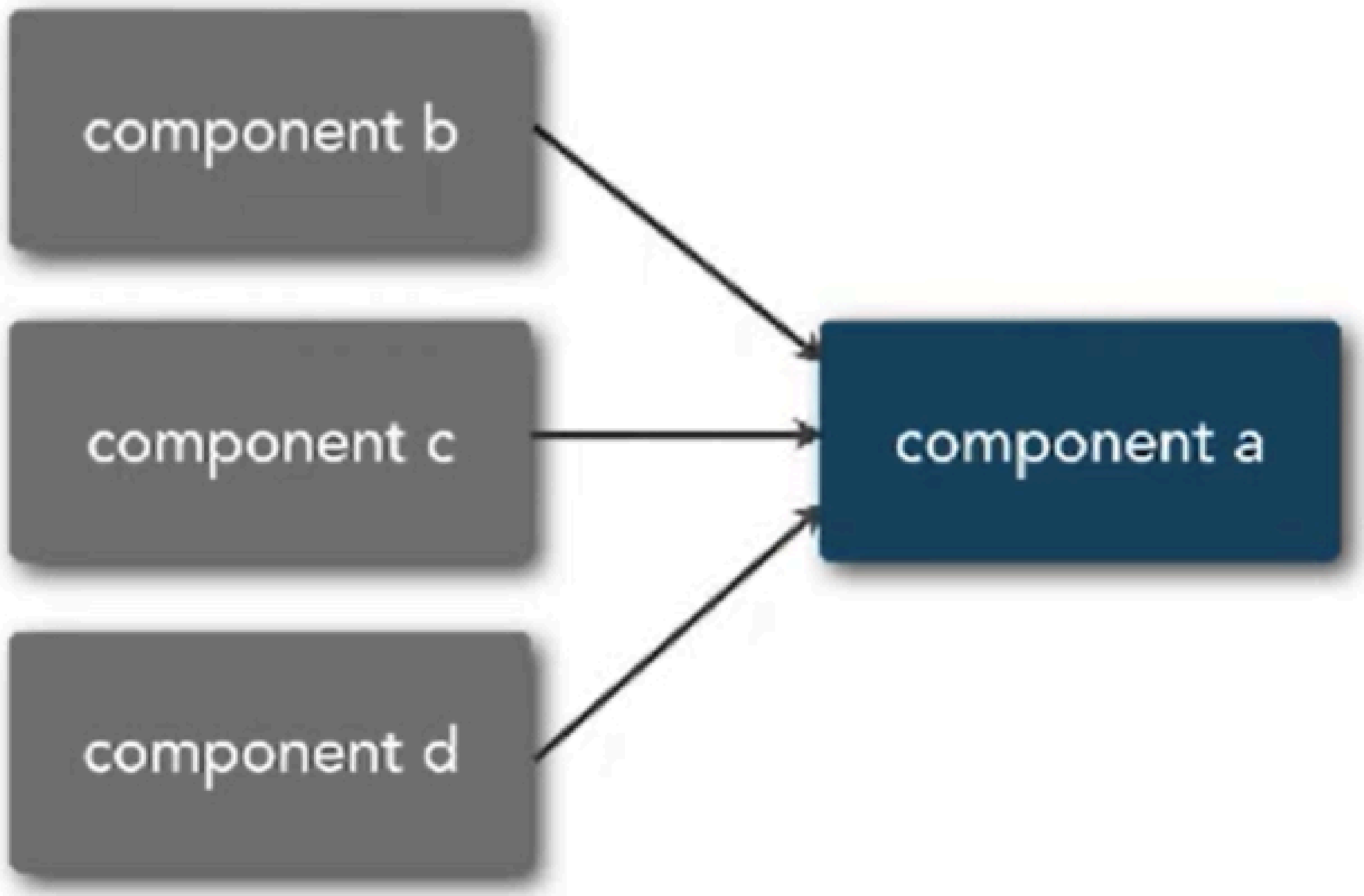
Component Coupling
the extent to which components know about each other

There are :
3 type of Coupling
4 level of Coupling



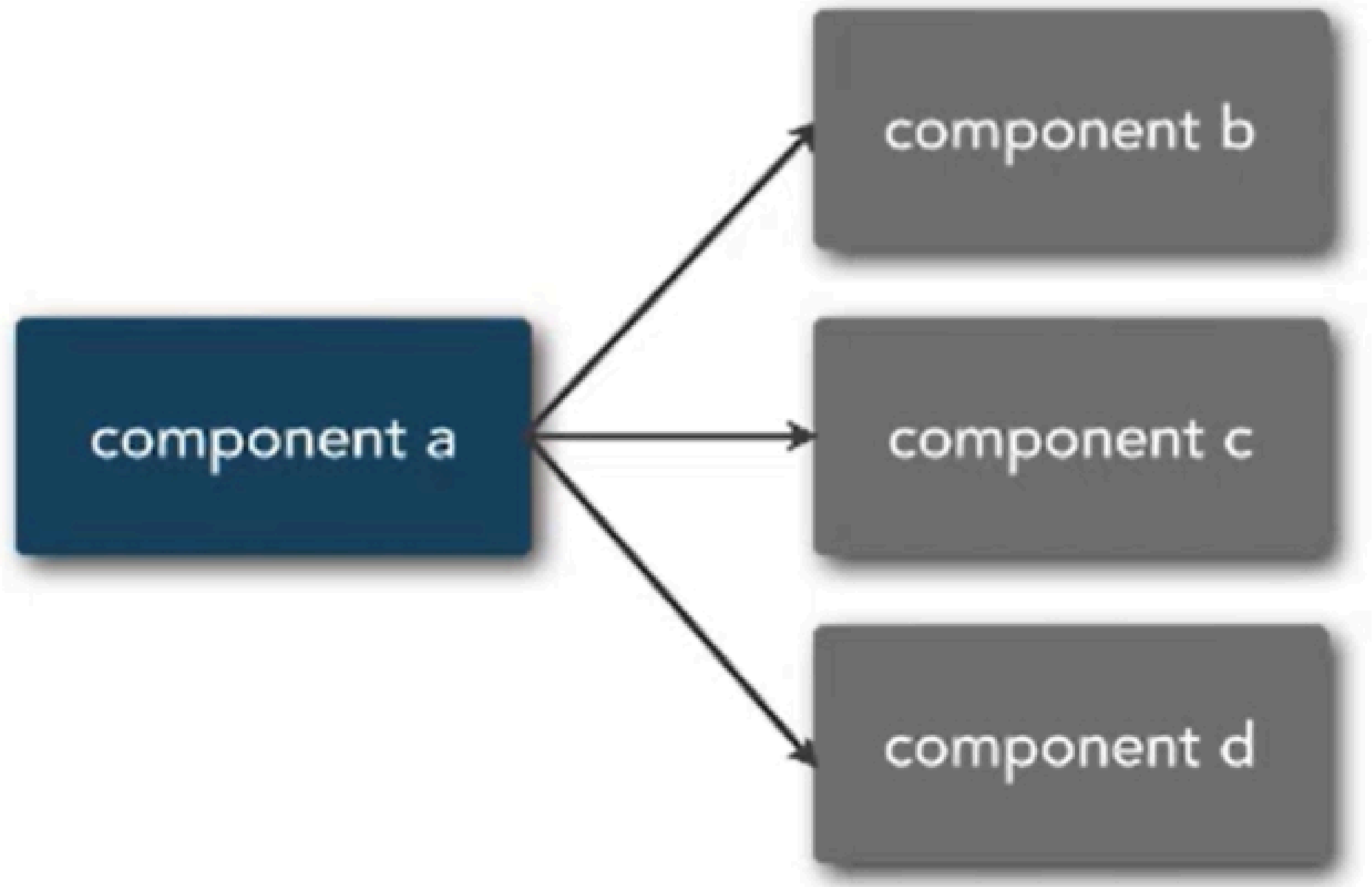
Type 1 (form of “static” level of Coupling)

afferent Coupling
the degree to which other component are dependent on the target component
also called “Fan-in”



Type 2 (form of “static” level of Coupling)

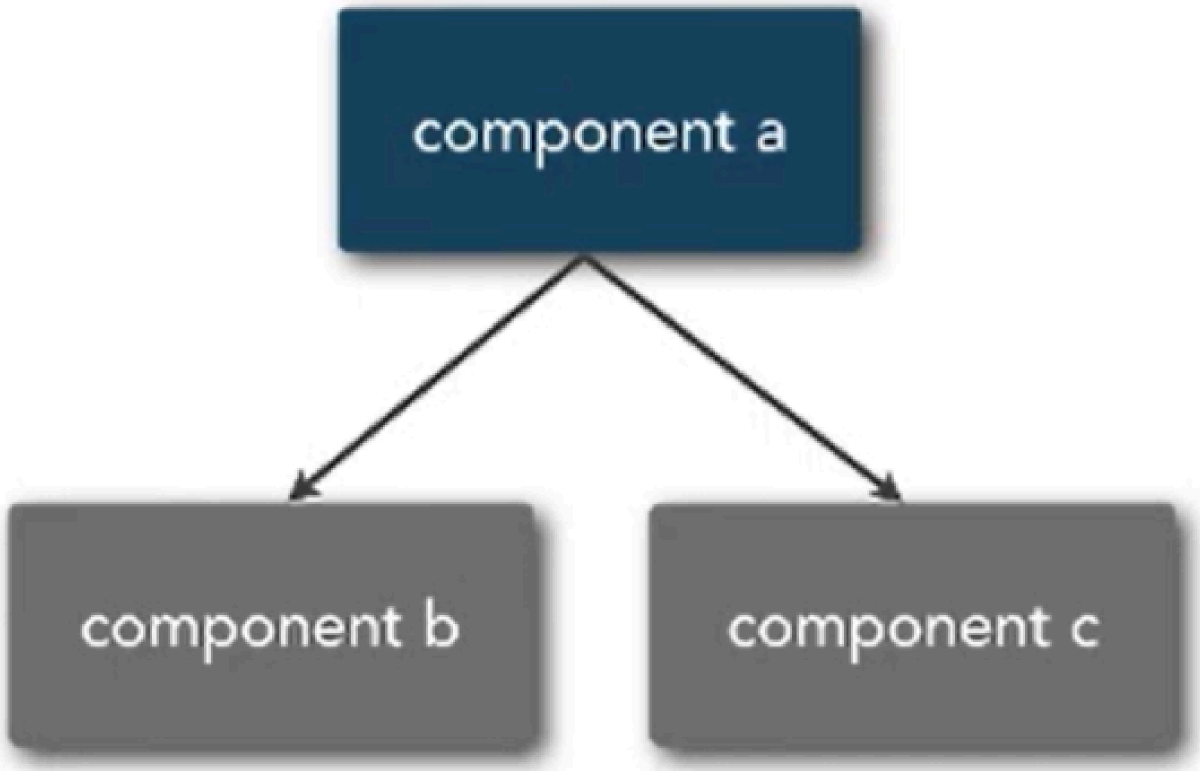
efferent Coupling
the degree to which the target component are dependent on other component
also called “Fan-out”



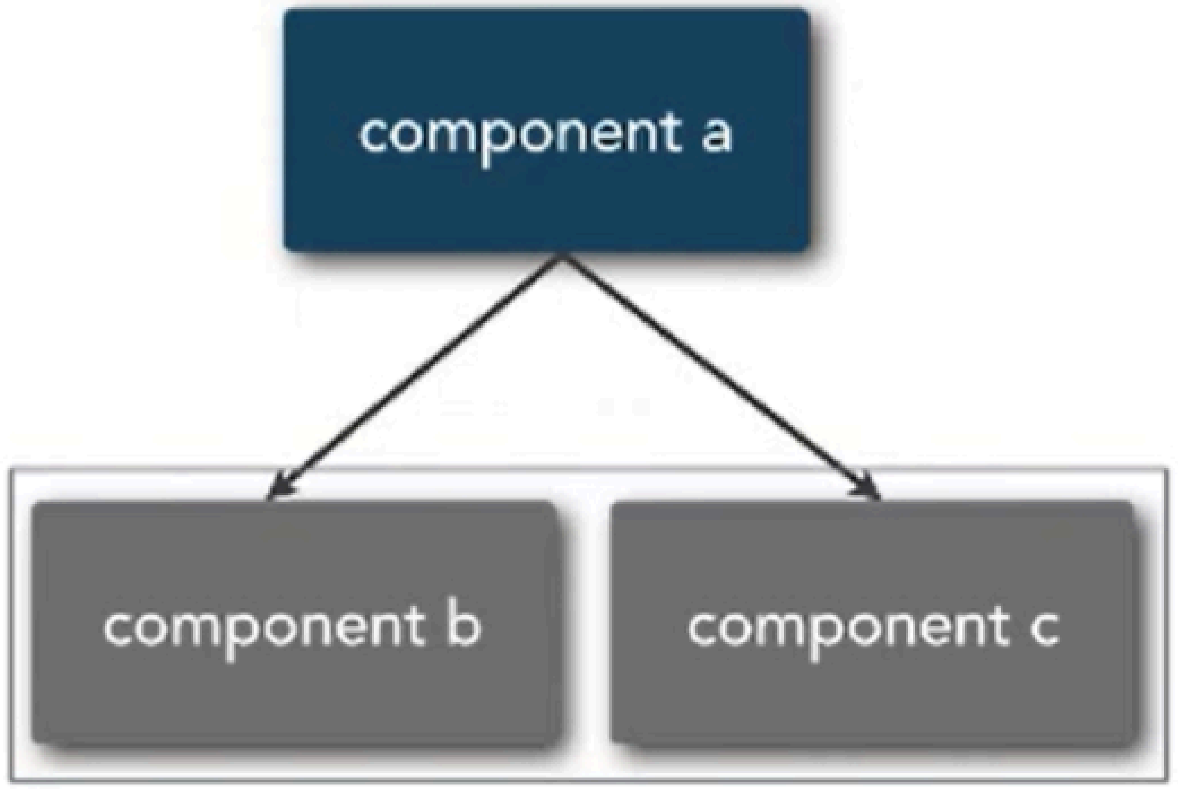
“In temporal coupling Components are coupled based on when they run, not what they do.”
-Gemini

Type 3 (form of “non-static” level of Coupling)

temporal Coupling
component are coupled due to non-static timing dependencies

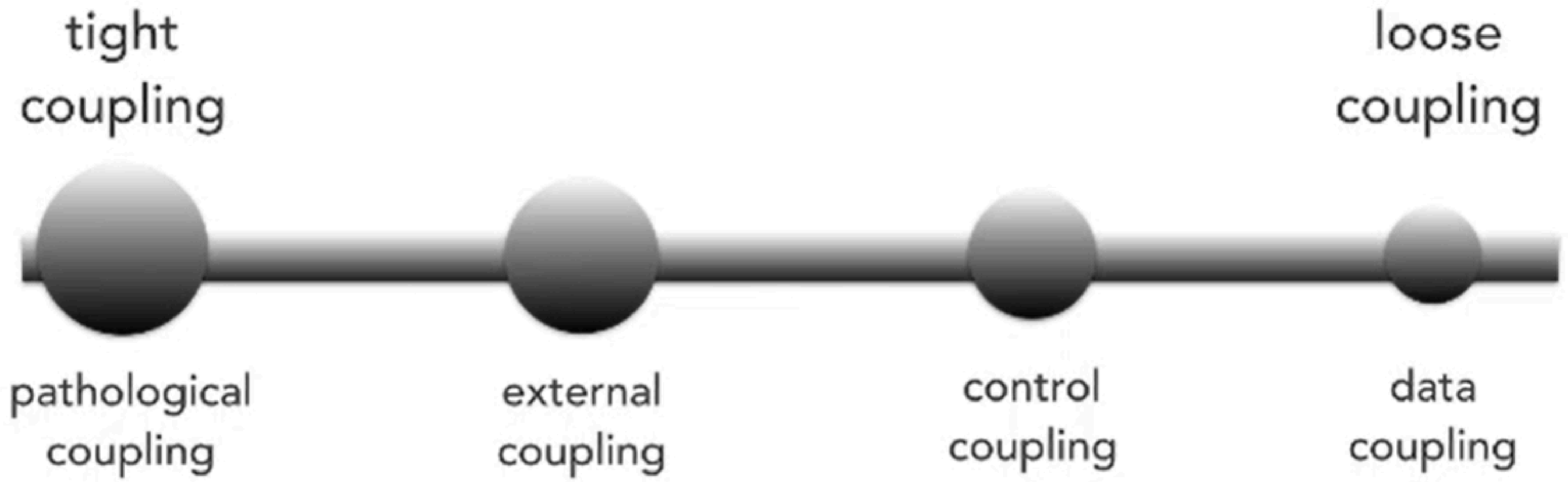


Orchestration
B must called before C



Unit of work that must be completed in a single logical unit of work

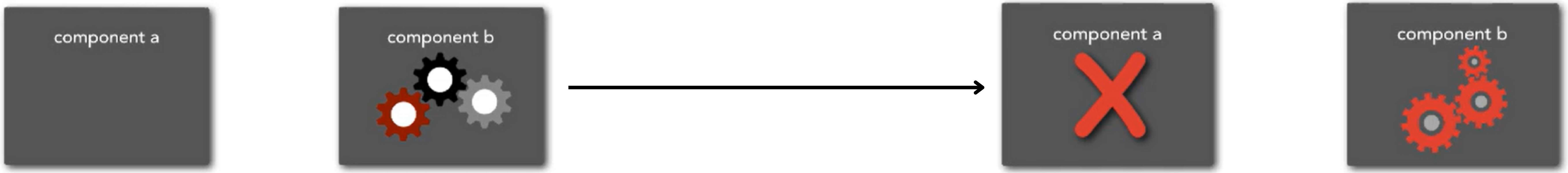
component coupling



These are 4 level of coupling

Pathological coupling

one component relies on the inner workings of another component

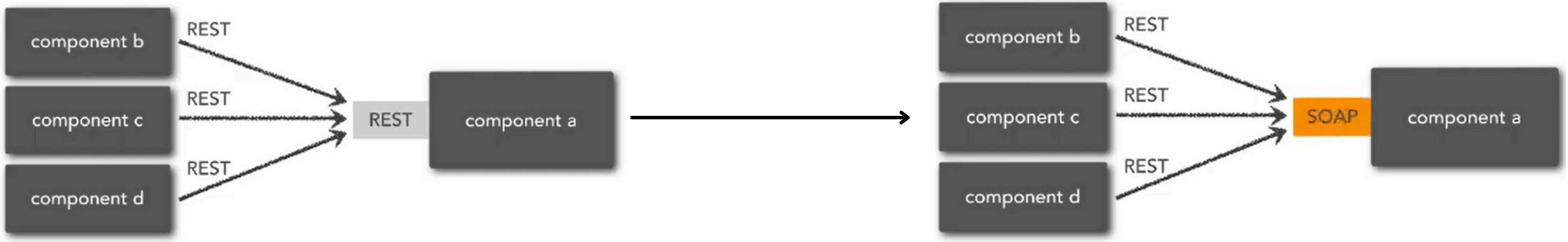


A relies on inner working of B

modify inner workings of B
affect the component A

External coupling

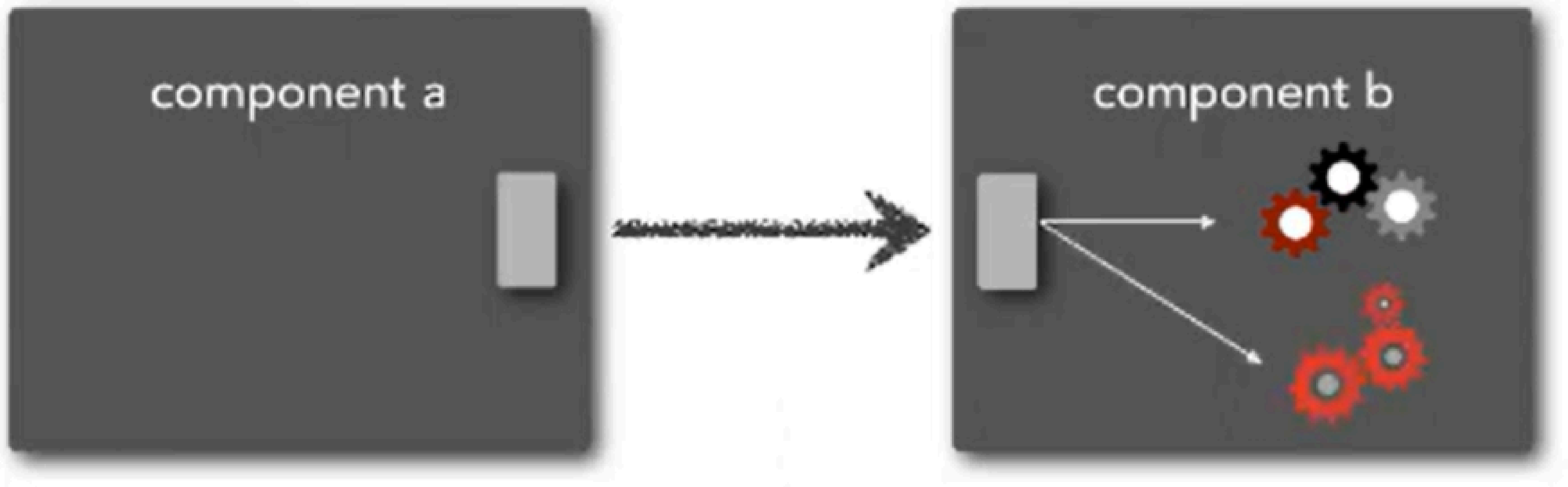
multiple component share an externally imposed protocol or data format



changed end-point to SOAP

Control coupling

one component passes information to another component on what to do

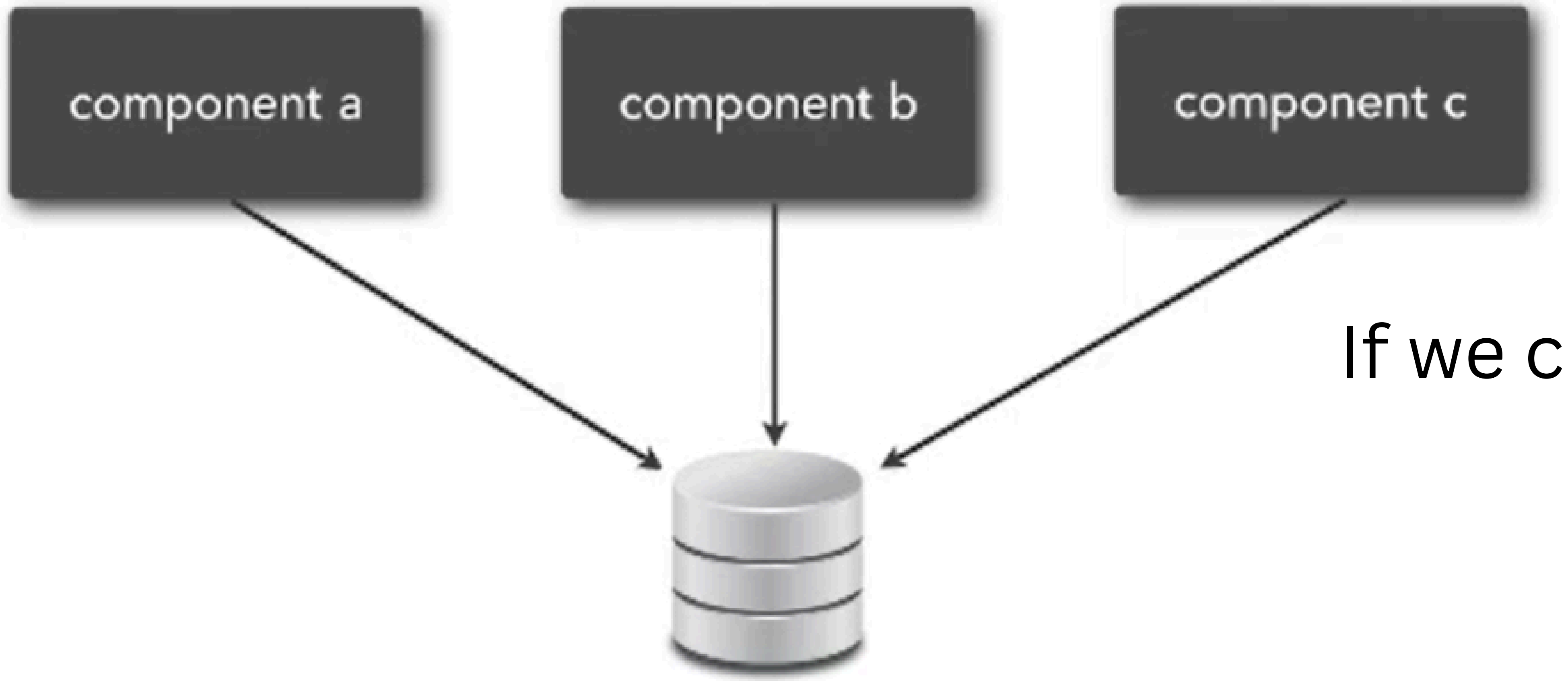


A make decision for B
(B relies on A)

Usually easy to solve
by making a separated operation for further decoupled systems

Data coupling

the degree to which component are bound to a shared data context



If we change the data format, it may affect those components.

A, B and C don’t know each other but bound a shared data context

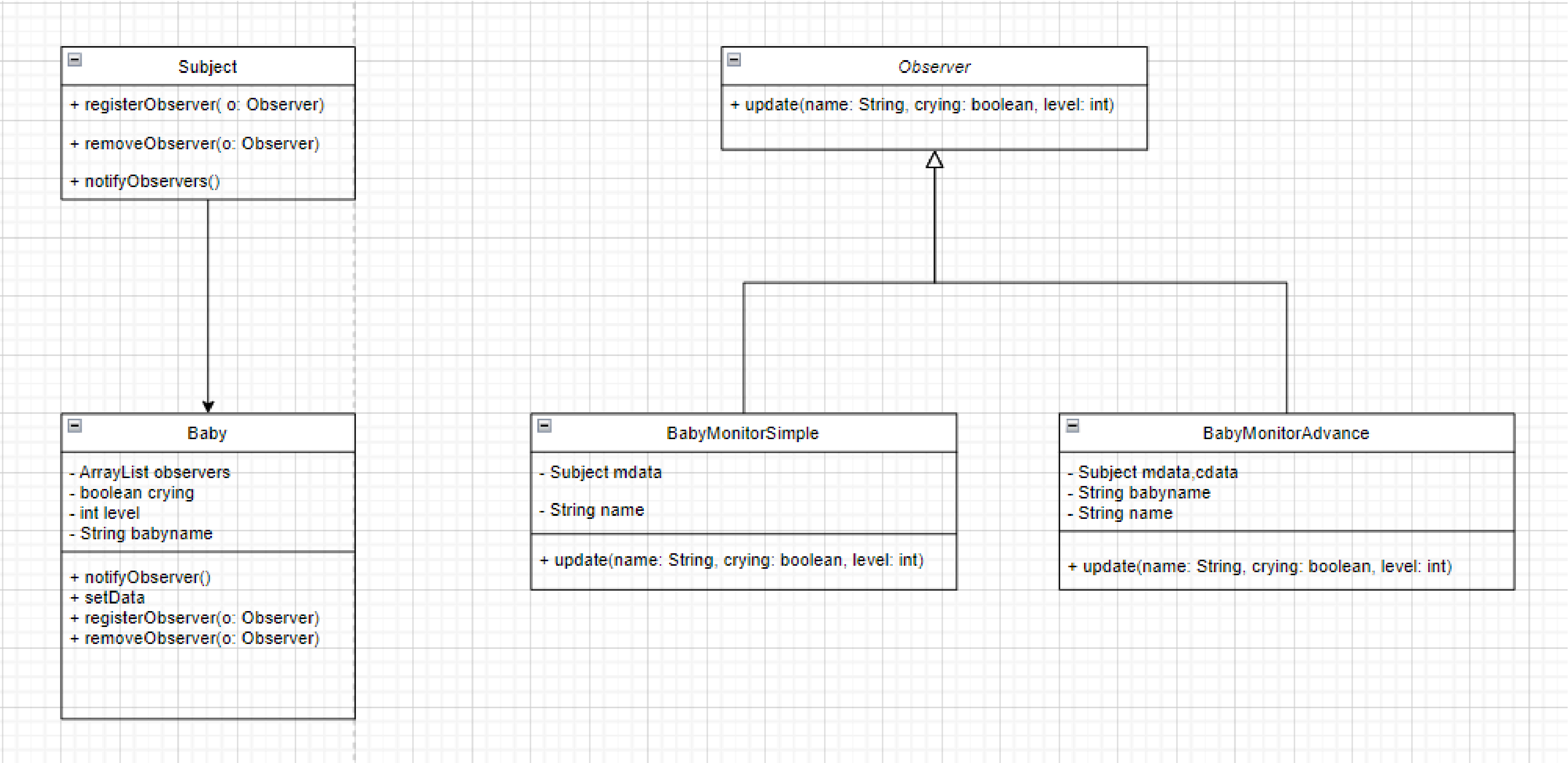
Task 2

Implement a simple Baby Monitoring System (In-Class Example) using Java Observable with:

1. Push strategy
2. Pull strategy

Add a short description

Push Strategy


Subject
+ registerObserver(o: Observer)
+ removeObserver(o: Observer)
+ notifyObservers()
Observer
+ update(name: String, crying: boolean, level: int)
Baby
- ArrayList observers
- boolean crying
- int level
- String babyname
+ notifyObserver()
+ setData
+ registerObserver(o: Observer)
+ removeObserver(o: Observer)
BabyMonitorSimple
- Subject mdata
- String name
+ update(name: String, crying: boolean, level: int)
BabyMonitorAdvance
- Subject mdata,cdata
- String babyname
- String name
+ update(name: String, crying: boolean, level: int)

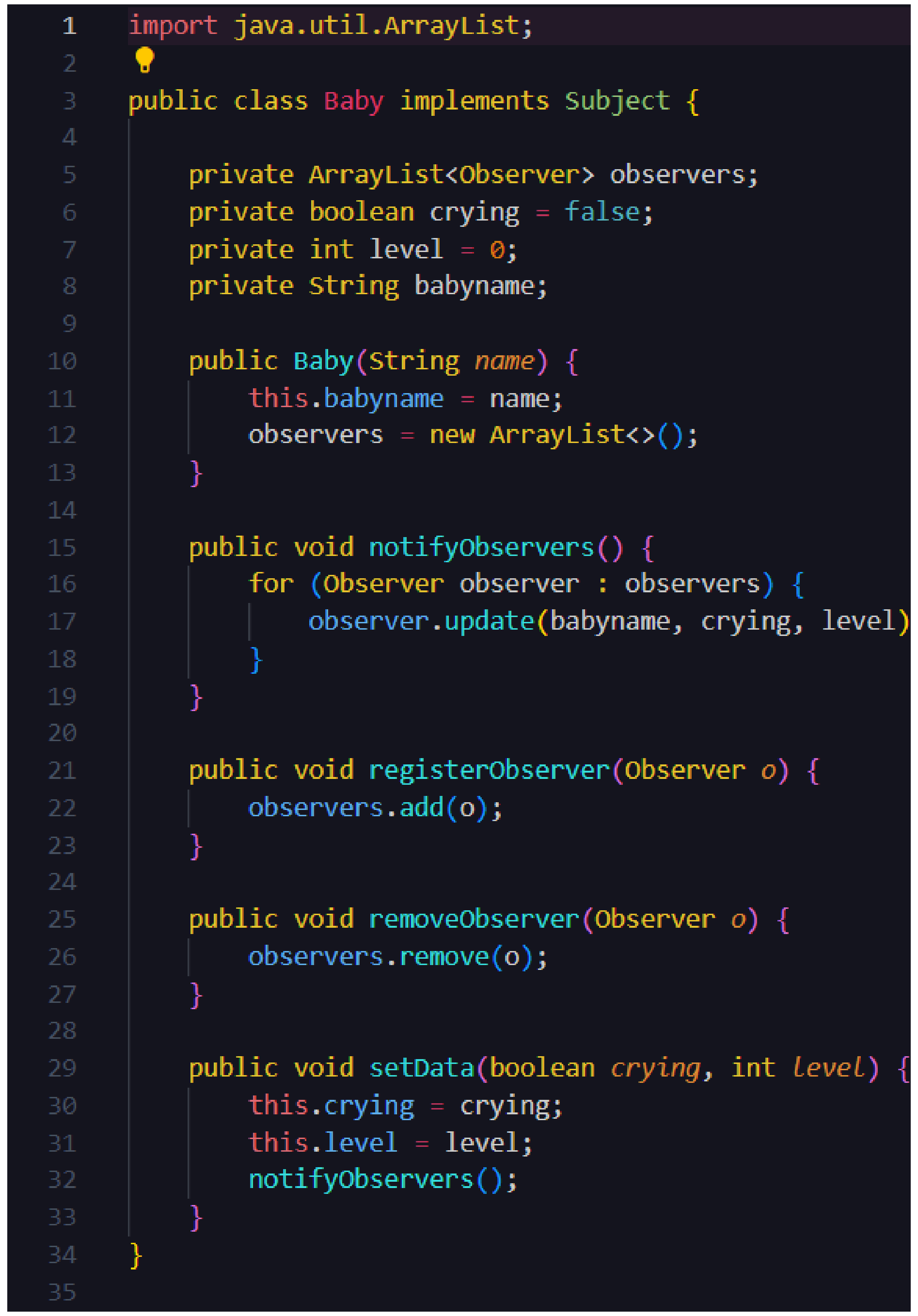
1. Subject (Baby) Implementation:

- The Baby class maintains a list of registered observers (observers using ArrayList).
- The setData method in the Baby class updates the crying state (crying) and level (level).
- Crucially, in the push strategy, the setData method calls the notifyObservers method. This triggers notifications to all registered monitors.
- The notifyObservers method iterates through the list of observers and calls their update method, passing the baby name (babyname), crying state (crying), and level (level) as arguments.

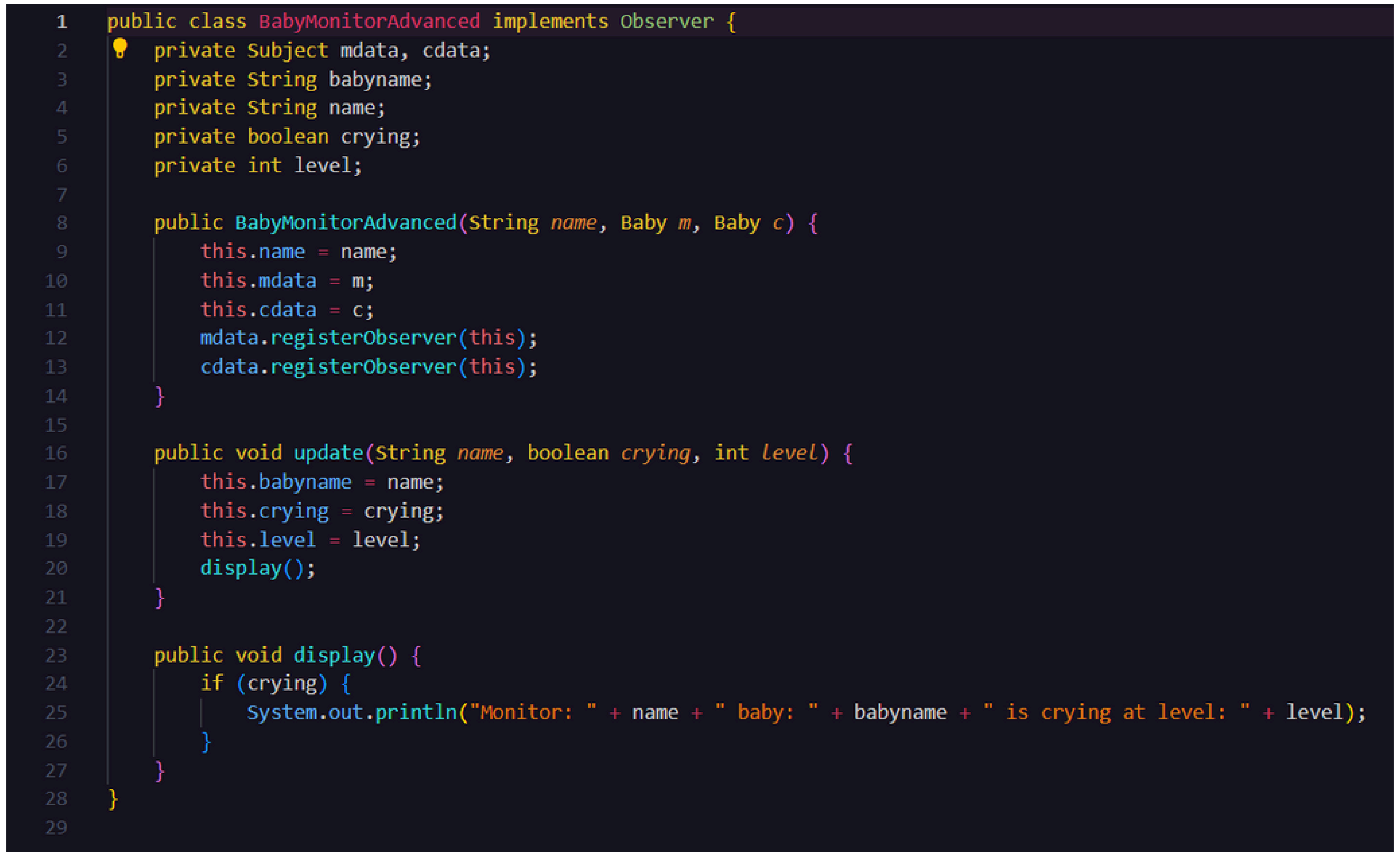
2. Observer (Monitor) Implementation:

- Both BabyMonitorSimple and BabyMonitorAdvanced classes implement the Observer interface, which defines the update method.
- The update method in these classes receives the pushed data (baby name, crying state, and level) as arguments from the Baby object.

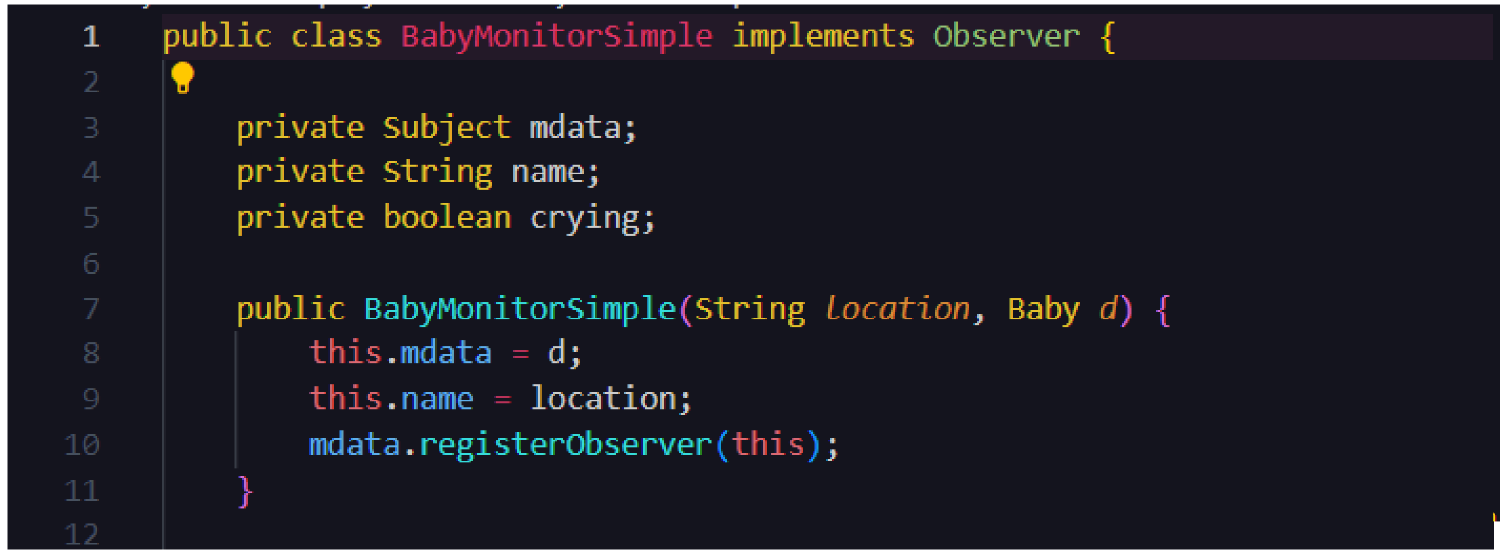
Push Strategy



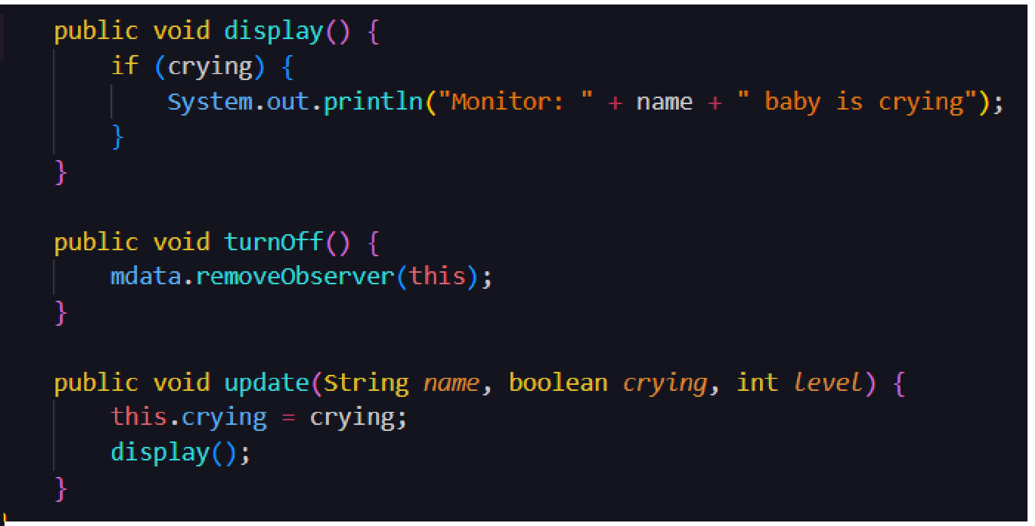
```java
import java.util.ArrayList;

public class Baby implements Subject {

    private ArrayList<Observer> observers;
    private boolean crying = false;
    private int level = 0;
    private String babyname;

    public Baby(String name) {
        this.babyname = name;
        observers = new ArrayList<>();
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(babyname, crying, level)
        }
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    public void setData(boolean crying, int level) {
        this.crying = crying;
        this.level = level;
        notifyObservers();
    }
}
```



```java
public class BabyMonitorAdvanced implements Observer {
    private Subject mdata, cdata;
    private String babyname;
    private String name;
    private boolean crying;
    private int level;

    public BabyMonitorAdvanced(String name, Baby m, Baby c) {
        this.name = name;
        this.mdata = m;
        this.cdata = c;
        mdata.registerObserver(this);
        cdata.registerObserver(this);
    }

    public void update(String name, boolean crying, int level) {
        this.babyname = name;
        this.crying = crying;
        this.level = level;
        display();
    }

    public void display() {
        if (crying) {
            System.out.println("Monitor: " + name + " baby: " + babyname + " is crying at level: " + level);
        }
    }
}
```

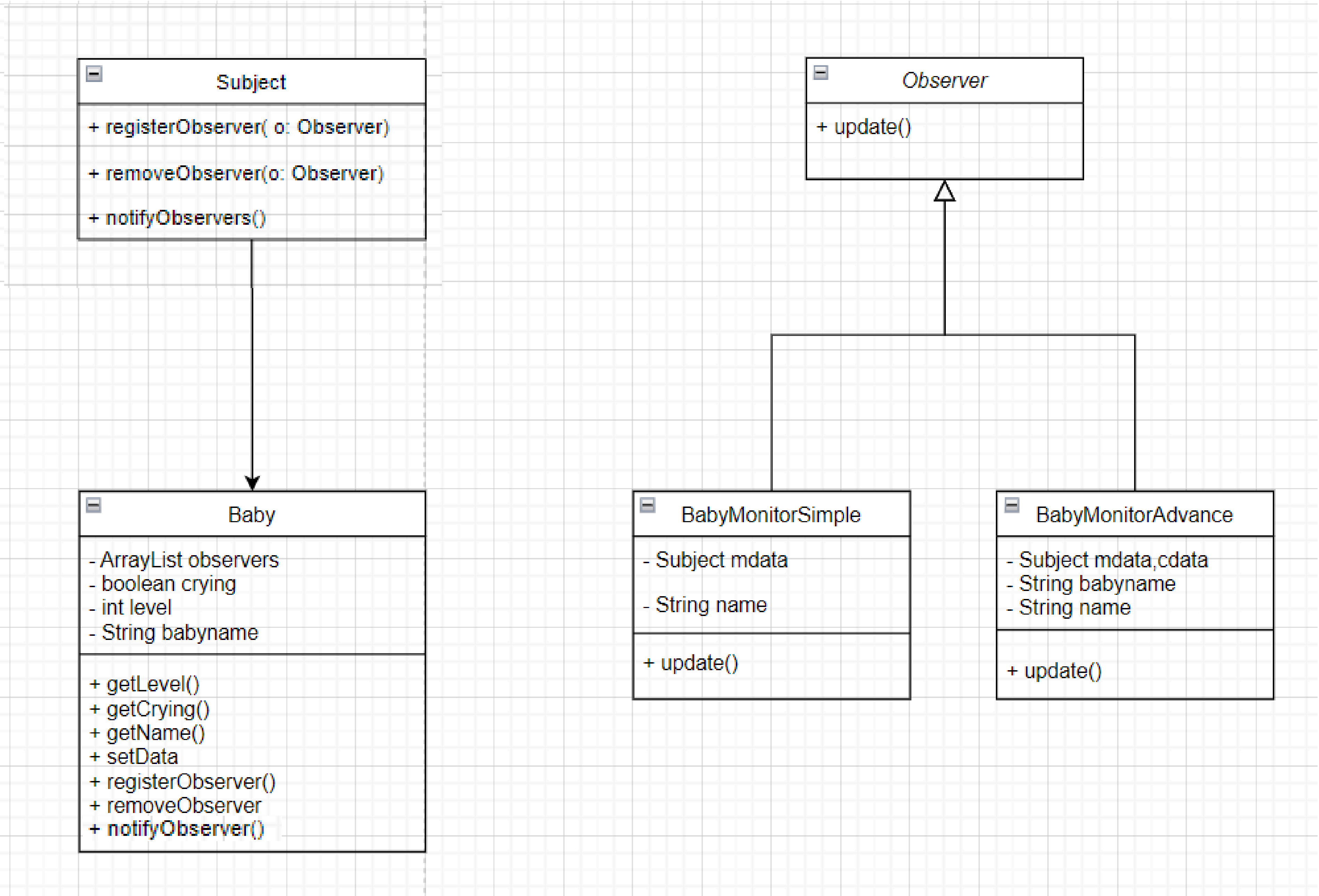


```java
public class BabyMonitorSimple implements Observer {

    private Subject mdata;
    private String name;
    private boolean crying;

    public BabyMonitorSimple(String location, Baby d) {
        this.mdata = d;
        this.name = location;
        mdata.registerObserver(this);
    }

```



```java
public void display() {
    if (crying) {
        System.out.println("Monitor: " + name + " baby is crying");
    }
}

public void turnOff() {
    mdata.removeObserver(this);
}

public void update(String name, boolean crying, int level) {
    this.crying = crying;
    display();
}
```

Pull Strategy



Modify Baby class:

- Add a method getCrying() that returns the current crying state (boolean).
- Add a method getLevel() that returns the current crying level (int).
- Modify setData(crying, level) to update internal state and not call notifyObservers.

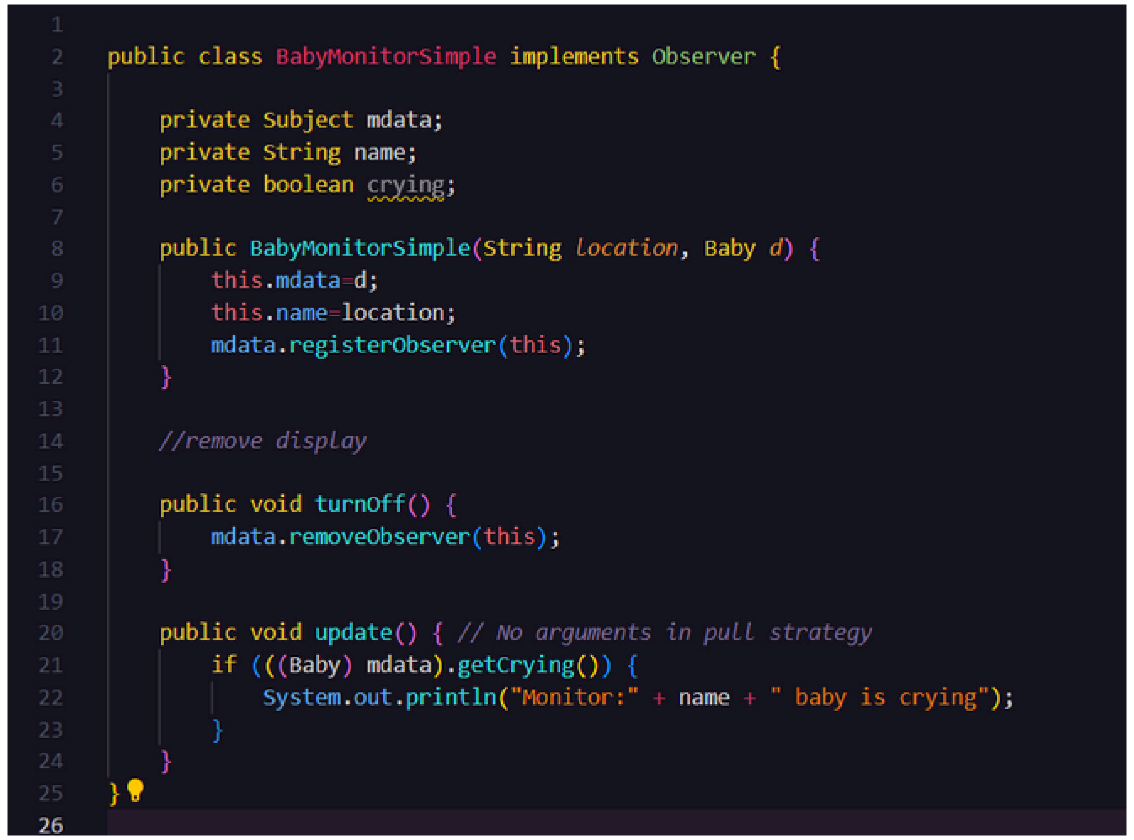
Modify Observer Interfaces (Subject and Observer):

- No changes needed for Subject.
- Modify Observer's update method to not take arguments.

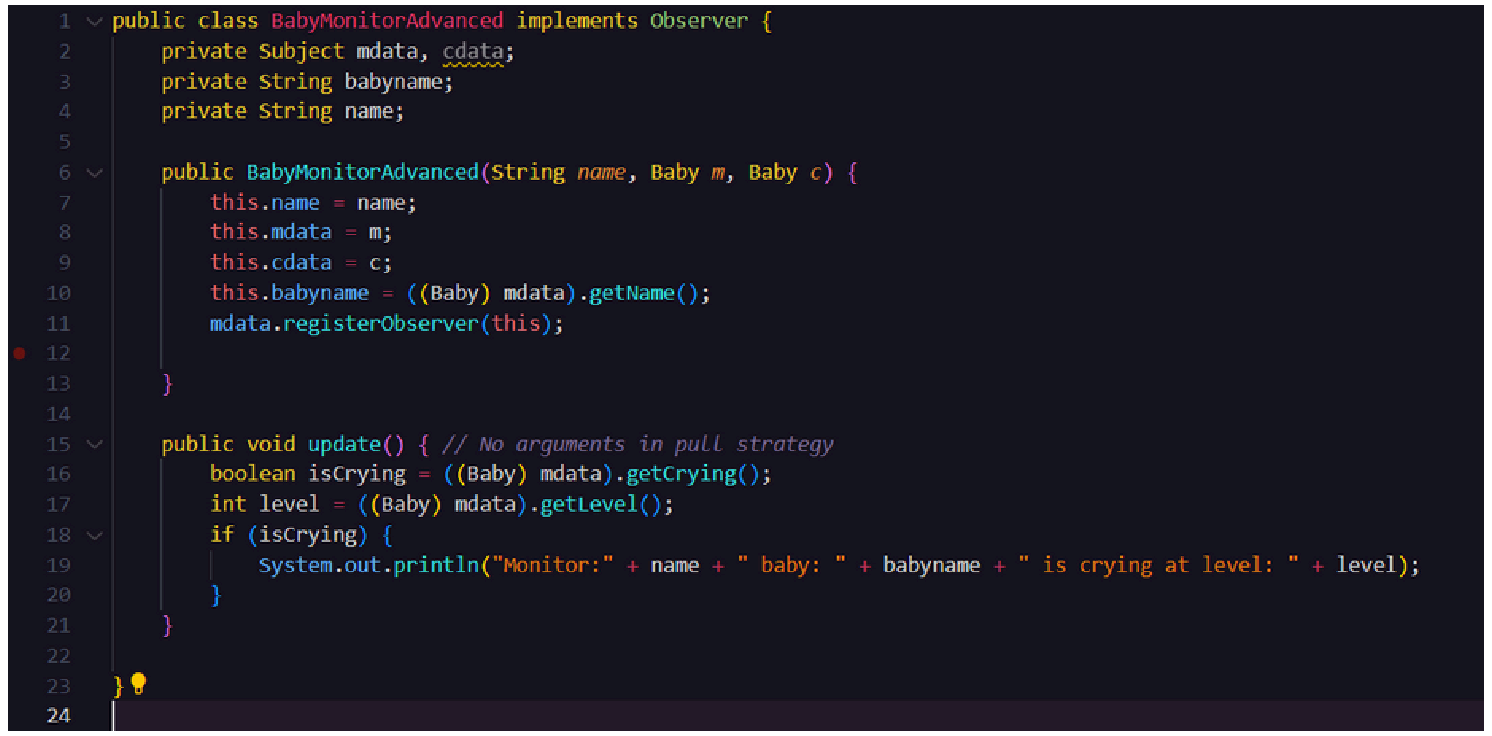
Modify Observers (BabyMonitorSimple and BabyMonitorAdvanced):

- Instead of relying on the update arguments, the observers will call the new methods (getCrying and getLevel) on the Subject (the Baby) to retrieve the latest data.

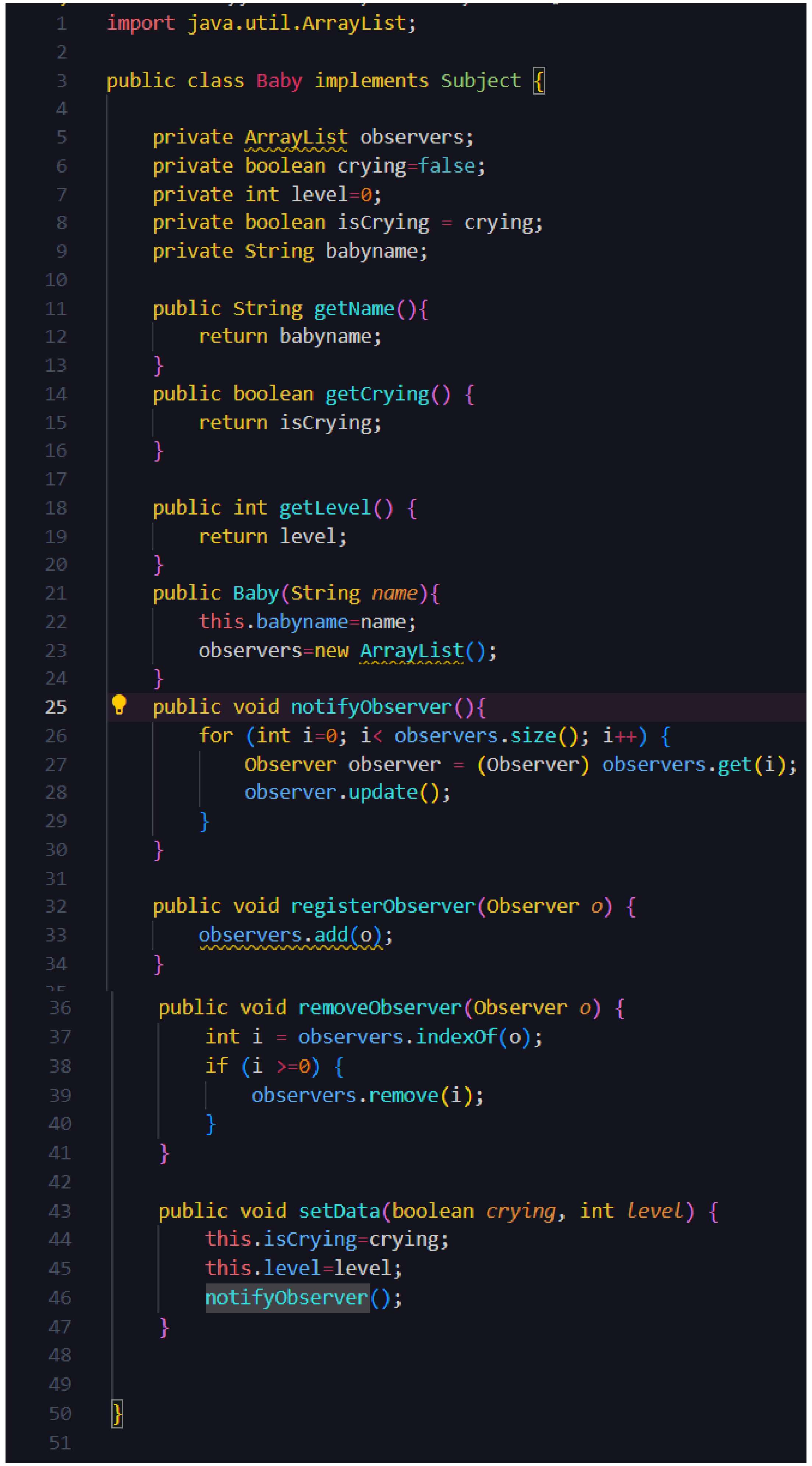
Pull Strategy



```java

public class BabyMonitorSimple implements Observer {

    private Subject mdata;
    private String name;
    private boolean crying;

    public BabyMonitorSimple(String location, Baby d) {
        this.mdata=d;
        this.name=location;
        mdata.registerObserver(this);
    }

    //remove display

    public void turnOff() {
        mdata.removeObserver(this);
    }

    public void update() { // No arguments in pull strategy
        if (((Baby) mdata).getCrying()) {
            System.out.println("Monitor:" + name + " baby is crying");
        }
    }
}

```



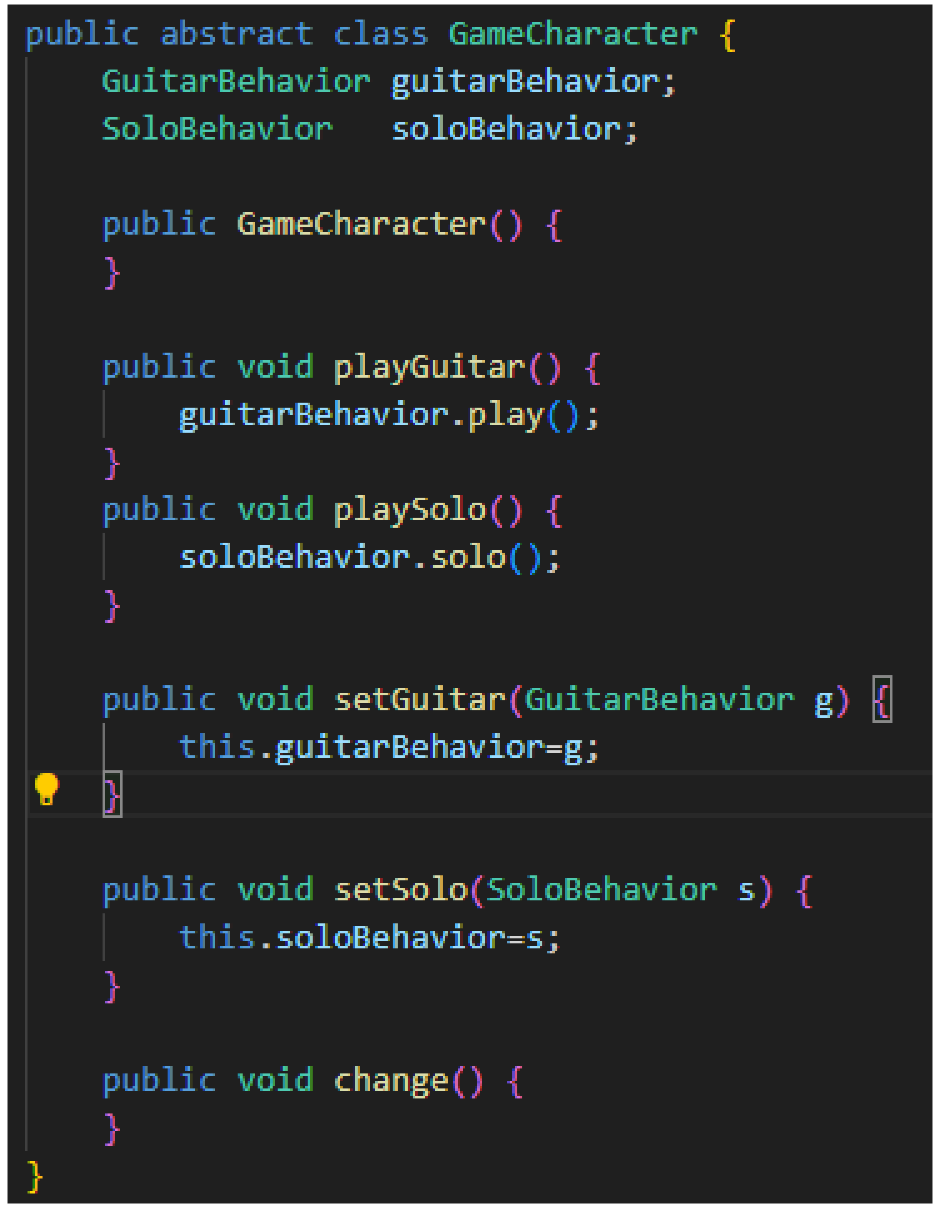
```java
public class BabyMonitorAdvanced implements Observer {
    private Subject mdata, cdata;
    private String babyname;
    private String name;

    public BabyMonitorAdvanced(String name, Baby m, Baby c) {
        this.name = name;
        this.mdata = m;
        this.cdata = c;
        this.babyname = ((Baby) mdata).getName();
        mdata.registerObserver(this);

    }

    public void update() { // No arguments in pull strategy
        boolean isCrying = ((Baby) mdata).getCrying();
        int level = ((Baby) mdata).getLevel();
        if (isCrying) {
            System.out.println("Monitor:" + name + " baby: " + babyname + " is crying at level: " + level);
        }
    }

}

```



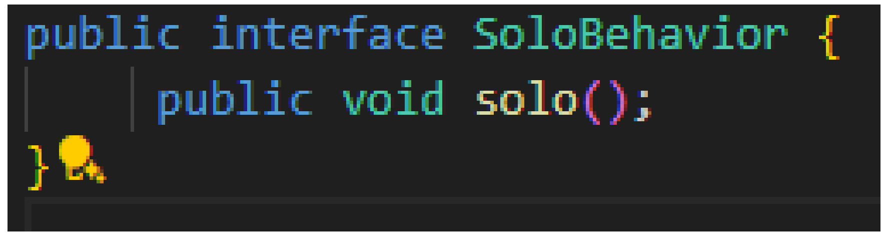
```java
import java.util.ArrayList;

public class Baby implements Subject {

    private ArrayList observers;
    private boolean crying=false;
    private int level=0;
    private boolean isCrying = crying;
    private String babyname;

    public String getName(){
        return babyname;
    }
    public boolean getCrying() {
        return isCrying;
    }

    public int getLevel() {
        return level;
    }
    public Baby(String name){
        this.babyname=name;
        observers=new ArrayList();
    }
    public void notifyObserver(){
        for (int i=0; i< observers.size(); i++) {
            Observer observer = (Observer) observers.get(i);
            observer.update();
        }
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >=0) {
            observers.remove(i);
        }
    }

    public void setData(boolean crying, int level) {
        this.isCrying=crying;
        this.level=level;
        notifyObserver();
    }


}

```

Task 3

Unchanged

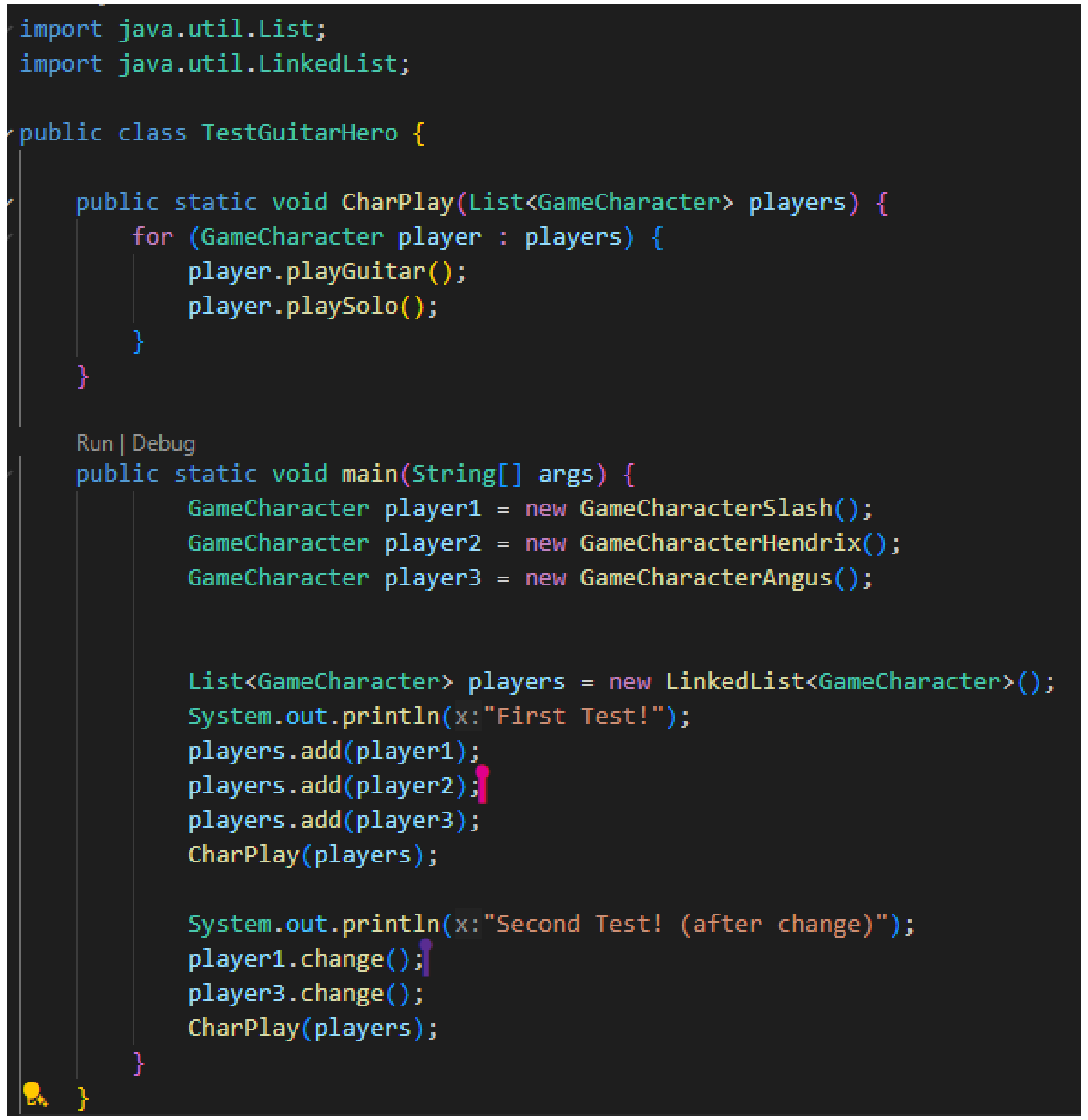


```
public abstract class GameCharacter {
    GuitarBehavior guitarBehavior;
    SoloBehavior   soloBehavior;

    public GameCharacter() {
    }

    public void playGuitar() {
        guitarBehavior.play();
    }
    public void playSolo() {
        soloBehavior.solo();
    }

    public void setGuitar(GuitarBehavior g) {
        this.guitarBehavior=g;
    }

    public void setSolo(SoloBehavior s) {
        this.soloBehavior=s;
    }

    public void change() {
    }
}
```



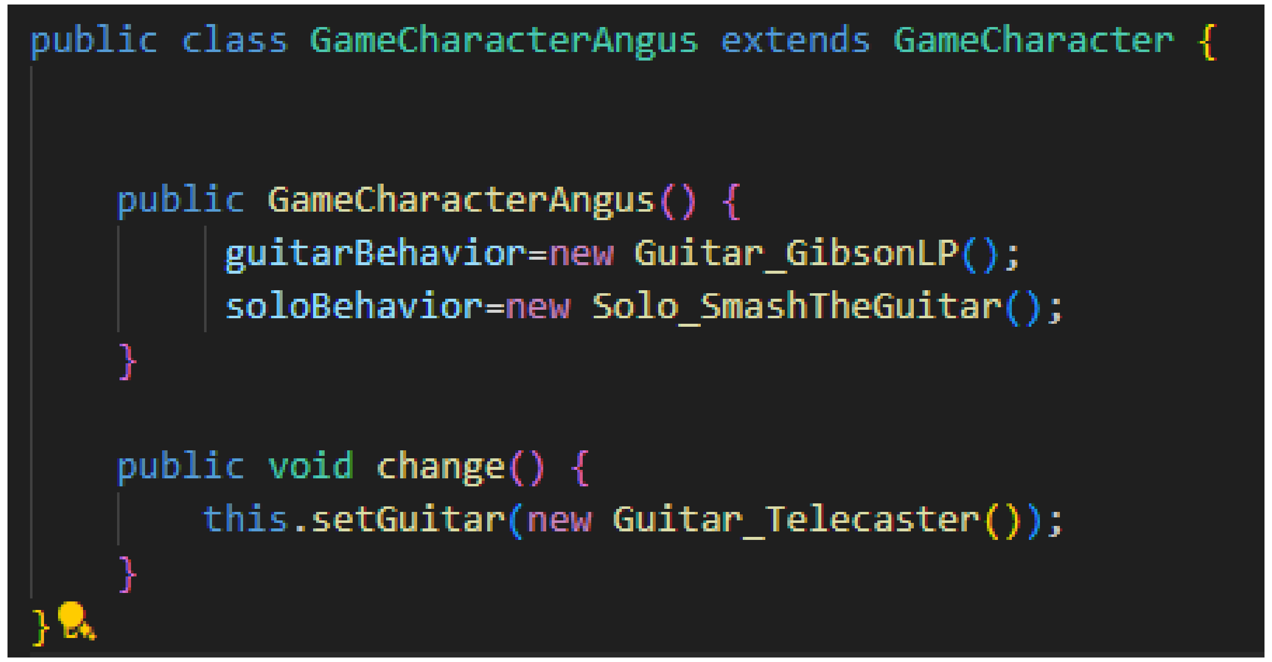
```
public interface SoloBehavior {
    public void solo();
}
```

Moddified

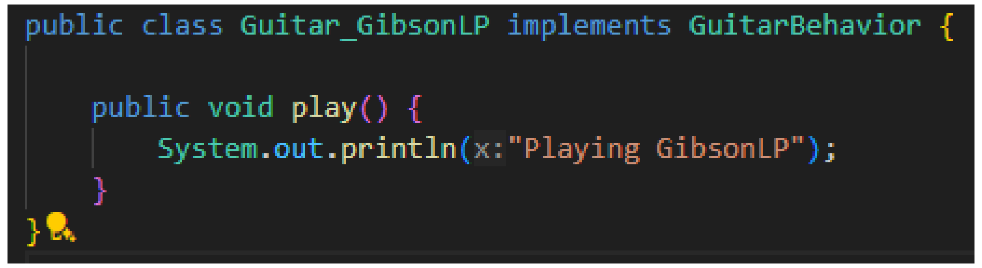


```
import java.util.List;
import java.util.LinkedList;

public class TestGuitarHero {

    public static void CharPlay(List<GameCharacter> players) {
        for (GameCharacter player : players) {
            player.playGuitar();
            player.playSolo();
        }
    }

    Run | Debug
    public static void main(String[] args) {
            GameCharacter player1 = new GameCharacterSlash();
            GameCharacter player2 = new GameCharacterHendrix();
            GameCharacter player3 = new GameCharacterAngus();


            List<GameCharacter> players = new LinkedList<GameCharacter>();
            System.out.println(x:"First Test!");
            players.add(player1);
            players.add(player2);
            players.add(player3);
            CharPlay(players);

            System.out.println(x:"Second Test! (after change)");
            player1.change();
            player3.change();
            CharPlay(players);
        }
    }
```

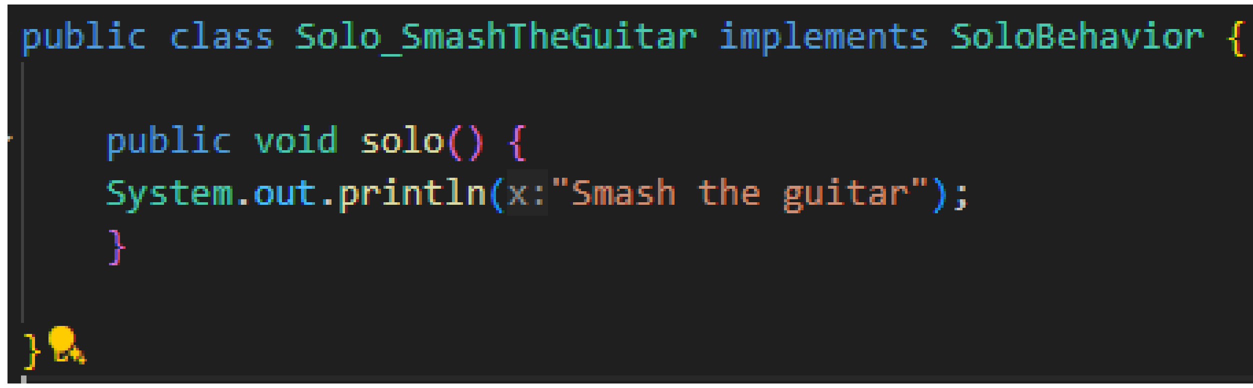
New



```
public class GameCharacterAngus extends GameCharacter {


    public GameCharacterAngus() {
        guitarBehavior=new Guitar_GibsonLP();
        soloBehavior=new Solo_SmashTheGuitar();
    }

    public void change() {
        this.setGuitar(new Guitar_Telecaster());
    }
}
```

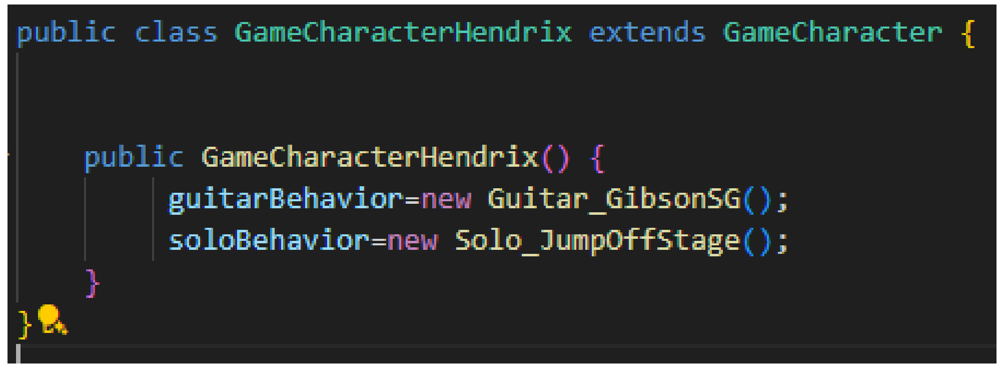


```
public class Guitar_GibsonLP implements GuitarBehavior {

    public void play() {
        System.out.println(x:"Playing GibsonLP");
    }
}
```

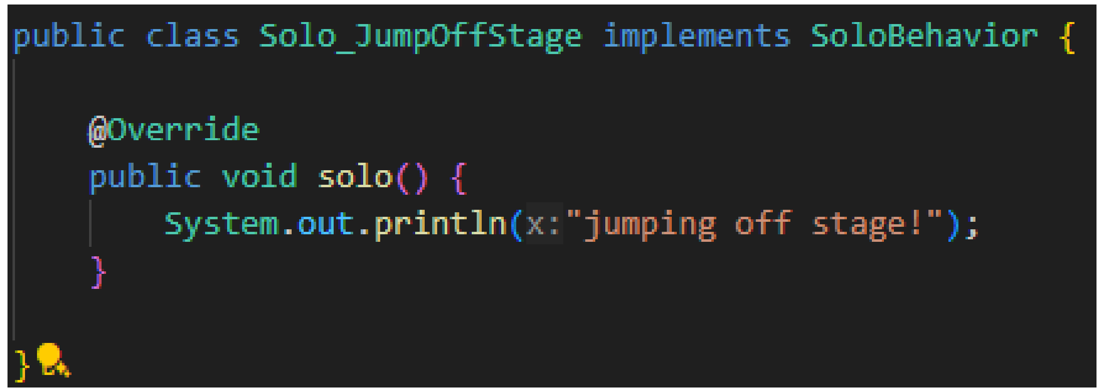


```
public class Solo_SmashTheGuitar implements SoloBehavior {

    public void solo() {
    System.out.println(x:"Smash the guitar");
    }

}
```

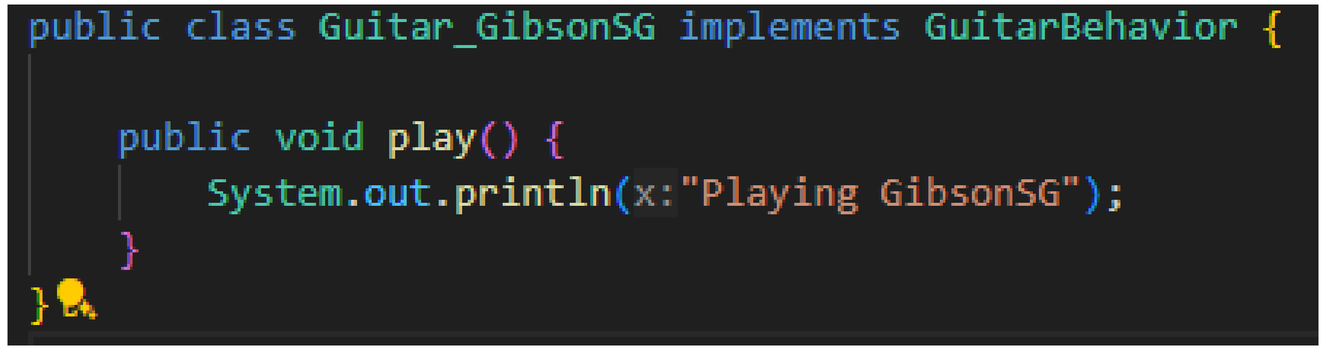
Unchanged



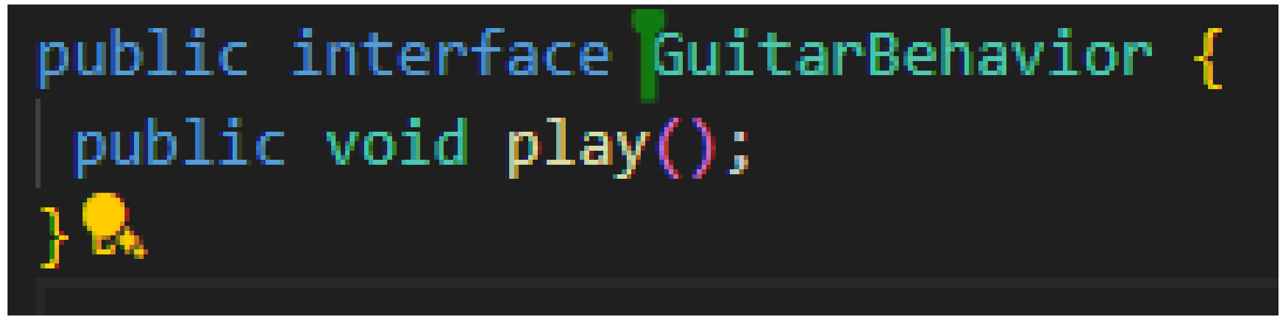
```java
public class GameCharacterHendrix extends GameCharacter {


    public GameCharacterHendrix() {
        guitarBehavior=new Guitar_GibsonSG();
        soloBehavior=new Solo_JumpOffStage();
    }
}
```



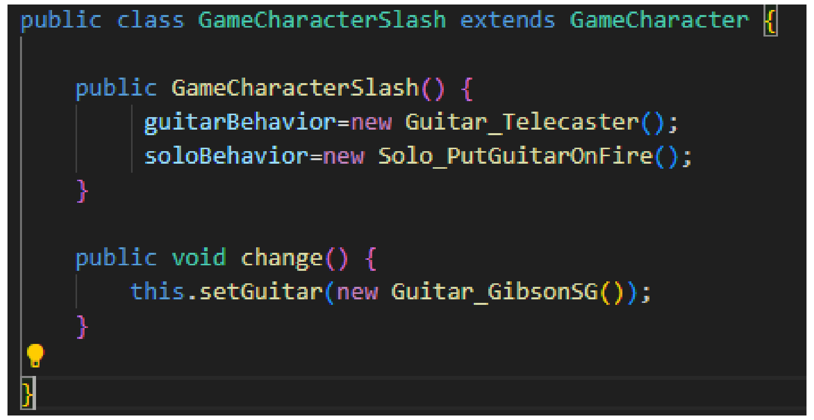
```java
public class Solo_JumpOffStage implements SoloBehavior {

    @Override
    public void solo() {
        System.out.println(x:"jumping off stage!");
    }

}
```



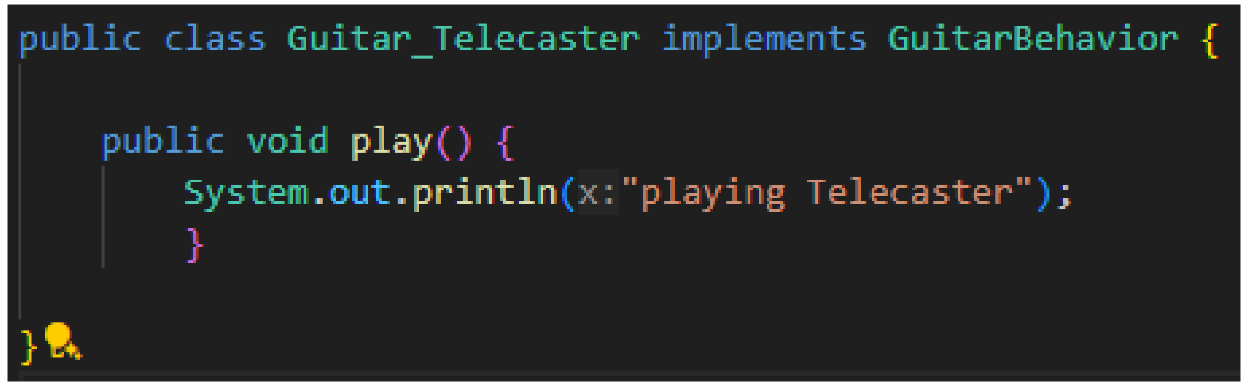
```java
public class Guitar_GibsonSG implements GuitarBehavior {

    public void play() {
        System.out.println(x:"Playing GibsonSG");
    }
}
```



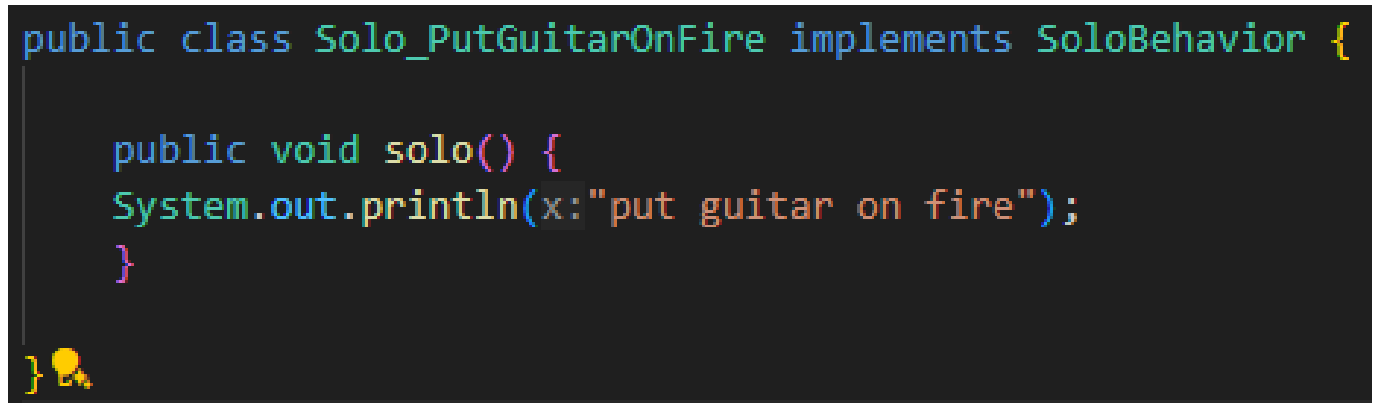
```java
public interface GuitarBehavior {
 public void play();
}
```



```java
public class GameCharacterSlash extends GameCharacter {

    public GameCharacterSlash() {
        guitarBehavior=new Guitar_Telecaster();
        soloBehavior=new Solo_PutGuitarOnFire();
    }

    public void change() {
        this.setGuitar(new Guitar_GibsonSG());
    }

}
```

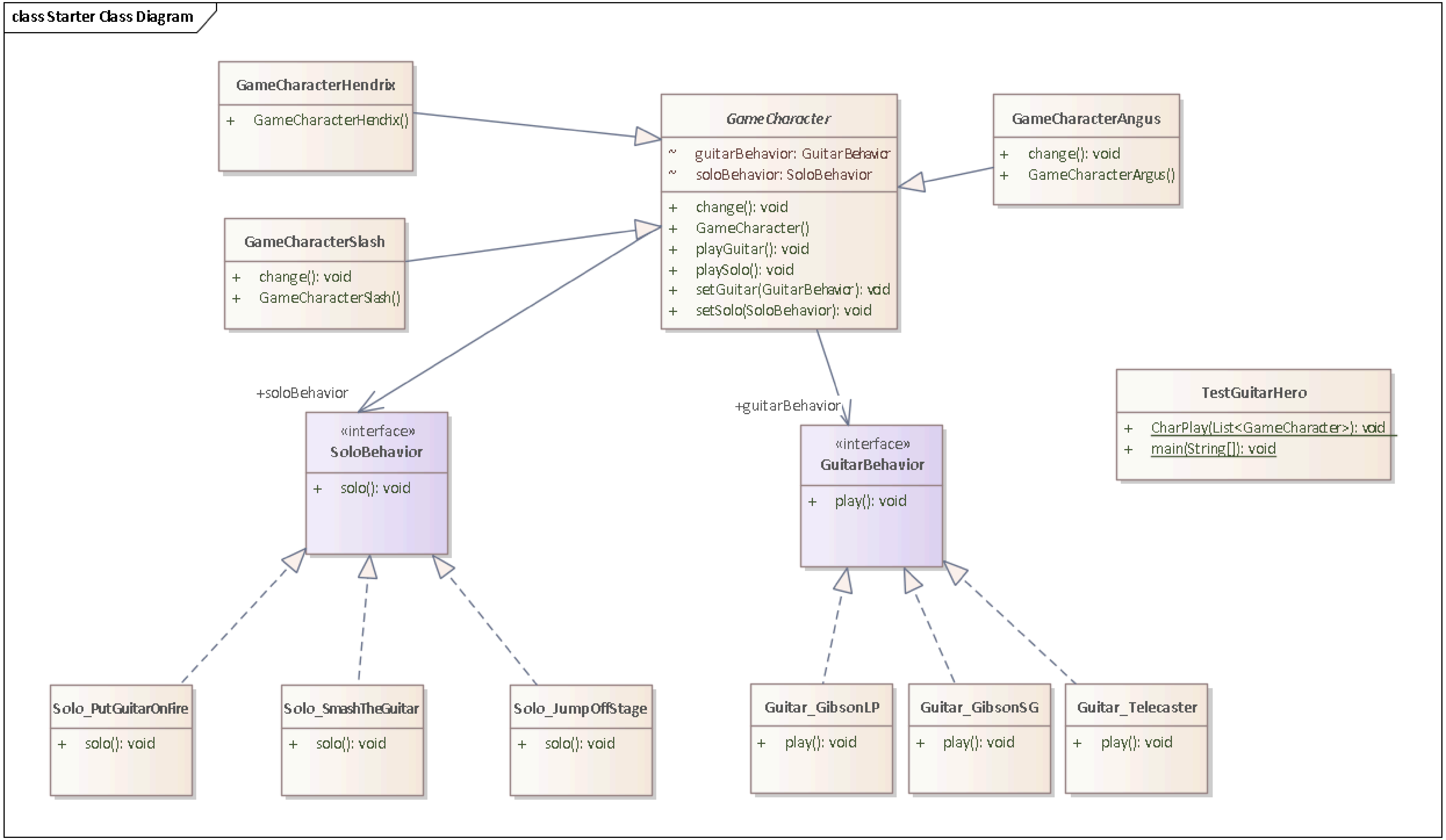


```java
public class Guitar_Telecaster implements GuitarBehavior {

    public void play() {
        System.out.println(x:"playing Telecaster");
        }

}
```



```java
public class Solo_PutGuitarOnFire implements SoloBehavior {

    public void solo() {
    System.out.println(x:"put guitar on fire");
    }

}
```

Class Diagram



class Starter Class Diagram

GameCharacterHendrix
+ GameCharacterHendrix()

GameCharacter
~ guitarBehavior: GuitarBehavior
~ soloBehavior: SoloBehavior
+ change(): void
+ GameCharacter()
+ playGuitar(): void
+ playSolo(): void
+ setGuitar(GuitarBehavior): void
+ setSolo(SoloBehavior): void

GameCharacterAngus
+ change(): void
+ GameCharacterAngus()

GameCharacterSlash
+ change(): void
+ GameCharacterSlash()

+soloBehavior

«interface»
SoloBehavior
+ solo(): void

+guitarBehavior

«interface»
GuitarBehavior
+ play(): void

TestGuitarHero
+ CharPlay(List<GameCharacter>): void
+ main(String[]): void

Solo_PutGuitarOnFire
+ solo(): void

Solo_SmashTheGuitar
+ solo(): void

Solo_JumpOffStage
+ solo(): void

Guitar_GibsonLP
+ play(): void

Guitar_GibsonSG
+ play(): void

Guitar_Telecaster
+ play(): void

Present By

65011277 Chanasorn Howattanakulphong
65011320 Kanokjan Singhsuwan
65011381 Napatr Sapprasert
65011400 Natthawut Lin
65011462 Phupa Denphatcharangkul
65011558 Suvijuk Samitimata
65011572 Teerapat Senanuch