

Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood_cubix  48660186

 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

Dependency Inversion Principle

Dependency Inversion Principle

- The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.
- Consists of two parts:
 - High-level modules should not depend on low-level modules. Both should depend on abstractions.
 - Abstractions should not depend on details. Details should depend on abstractions.

Ex 1: DIP

- So as we can see the Project class is a high level module and it depends on low level modules such as BackEndDeveloper and FrontEndDeveloper. We are actually violating the first part of the dependency inversion principle.
- Also by the inspecting the implement function of the Project.class we realize that the methods writeJava and writeJavascript are methods bound to the corresponding classes. Regarding the project scope those are details since in both cases they are forms of development. Thus the second part of the dependency inversion principle is violated.

Ex 1: DIP

- Let's get started with some code that violates that principle.

As a software team and we need to implement a project. For now the software team consists of:

- A BackEnd Developer
- A Front End Developer

Ex 1: violates DIP principle.

```
public class BackEndDeveloper {
    public void writeJava() {
        System.out.println("Write Java");
    }
}

public class FrontEndDeveloper {
    public void writeJavascript() {
        System.out.println("Write Javascript");
    }
}

public class Project {
    private BackEndDeveloper backEndDeveloper = new BackEndDeveloper();
    private FrontEndDeveloper frontEndDeveloper = new FrontEndDeveloper();
    public void implement() {
        backEndDeveloper.writeJava();
        frontEndDeveloper.writeJavascript();
    }
}

public class DemoApp {
    public static void main(String[] args) {
        Project proj = new Project();
        proj.implement();
    }
}
```

Output:

Write Java

Write Javascript

Ex 1: violates DIP principle.

- So as we can see, the Project class is a high-level module, and it depends on low-level modules such as BackEndDeveloper and FrontEndDeveloper. We are actually violating the first part of the dependency inversion principle.
- Also, by inspecting the implement function of Project.class, we realize that the methods writeJava and writeJavascript are methods bound to the corresponding classes. Regarding the project scope, those are details since, in both cases, they are forms of development. Thus, the second part of the dependency inversion principle is violated.

Ex 1: applying DIP principle

```
public interface Developer {  
    void develop();  
}
```

```
public class BackEndDeveloper implements Developer {  
    @Override  
    public void develop() {  
        writeJava();  
    }  
    private void writeJava() {  
        System.out.println("Write Java");  
    }  
}
```

```
public class FrontEndDeveloper implements Developer {  
    @Override  
    public void develop() {  
        writeJavascript();  
    }  
    public void writeJavascript() {  
        System.out.println("Write Java");  
    }  
}
```


Ex 1: applying DIP principle

```
import java.util.List;
public class Project {
    private List<Developer> developers;
    public Project(List<Developer> developers) {
        this.developers = developers;
    }
    public void implement() {
        developers.forEach(d->d.develop());
    }
}

import java.util.ArrayList;
public class DemoApp {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        FrontEndDeveloper FD = new FrontEndDeveloper();
        BackEndDeveloper BD = new BackEndDeveloper();
        List<Developer> list=new ArrayList<Developer>()
        list.add(FD);
        list.add(BD);
        Project man = new Project(list);
        man.implement();
    }
}
```

Output:

Write Java

Write Javascript

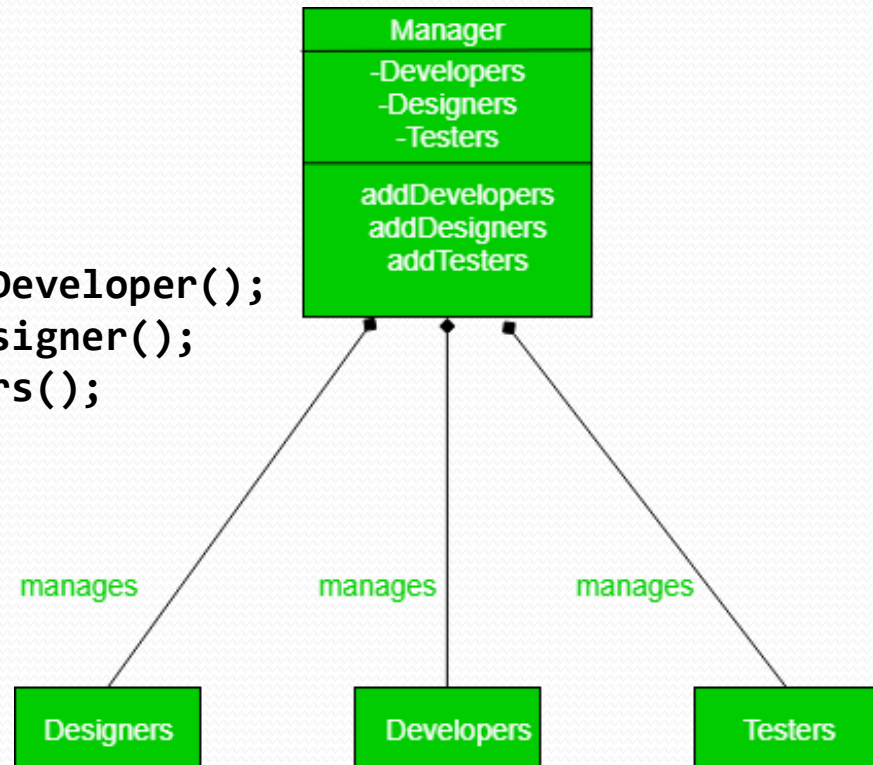
Ex2– DIP violation

```
public class Developer {
    public void writeCode() {
        System.out.println("developer added");
    }
}

public class Designer {
    public void writeHTMLCSS() {
        System.out.println("designer added");
    }
}

public class Testers {
    public void writeTestCase() {
        System.out.println("tester added");
    }
}

public class Manager {
    private Developer developers = new Developer();
    private Designer designers = new Designer();
    private Testers testers = new Testers();
    public void addDeveloper() {
        developers.writeCode();
    }
    public void addDesigners() {
        designers.writeHTMLCSS();
    }
    public void addTesters() {
        testers.writeTestCase();
    }
}
```



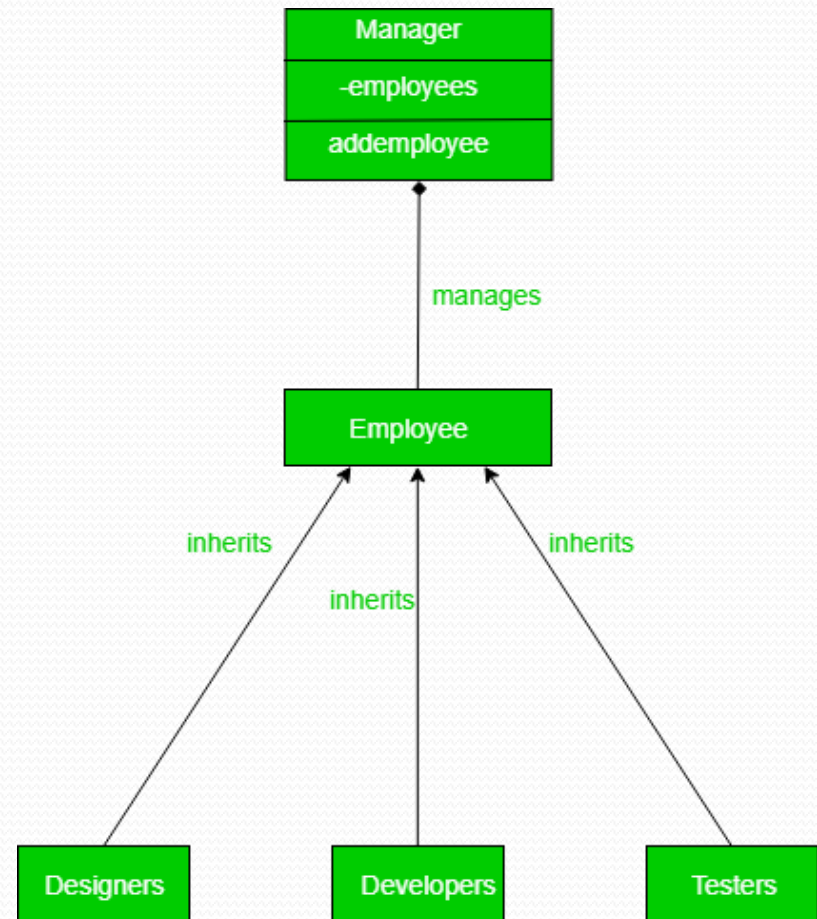
Ex2– DIP violation

```
public class DemoApp {  
    public static void main(String[] args) {  
        Manager man = new Manager();  
        man.addDeveloper();  
        man.addDesigners();  
        man.addTesters();  
    }  
}
```

Output:

```
developer added  
designer added  
tester added
```

- Class exercise : Convert the code so it confirms to DIP.
- Hint : see the new UML diagram, and see example 1 code above.



Ex 3 – DIP violation

- // Dependency Inversion Principle - Bad example

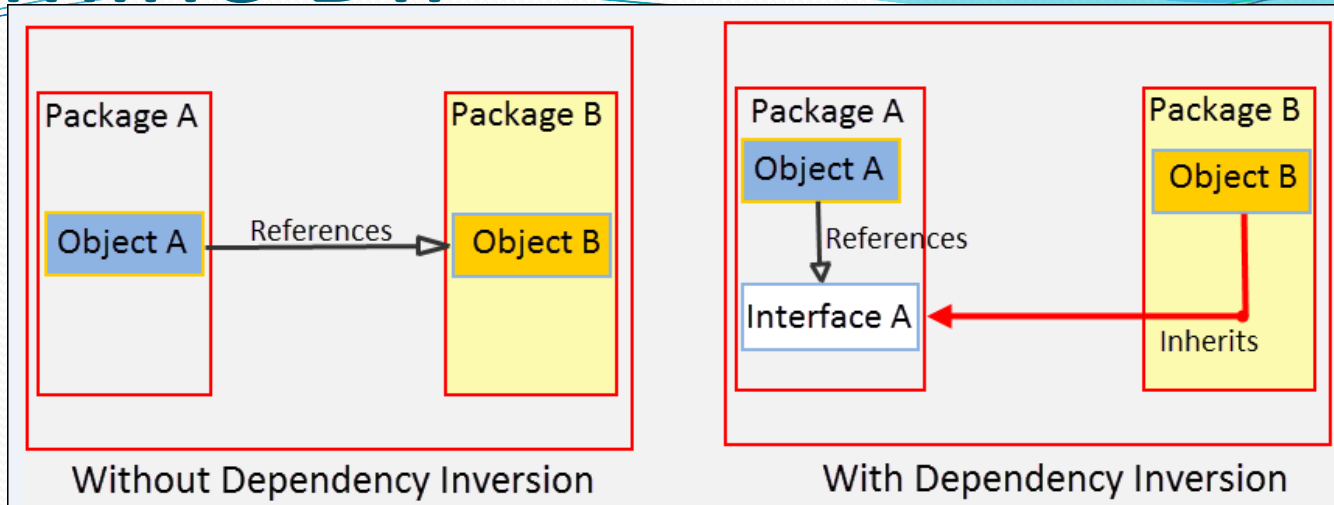
```
class Worker {  
    public void work() {  
        // ....working  
    }  
}  
  
class Manager {  
    Worker worker;  
    public void setWorker(Worker w) {  
        worker = w;  
    }  
    public void manage() {  
        worker.work();  
        superworker.work();  
    }  
}  
  
class SuperWorker {  
    public void work() {  
        //.... working much more  
    }  
}
```

Ex 3 – solution - DIP

- // Dependency Inversion Principle - Good example

```
interface IWorker {  
    public void work();  
}  
class Worker implements IWorker{  
    public void work() {  
        // ....working  
    }  
}  
class SuperWorker implements IWorker{  
    public void work() {  
        //.... working much more  
    }  
}  
class Manager {  
    IWorker worker;  
    public void setWorker(IWorker w) {  
        worker = w;  
    }  
    public void manage() {  
        worker.work();  
    }  
}
```

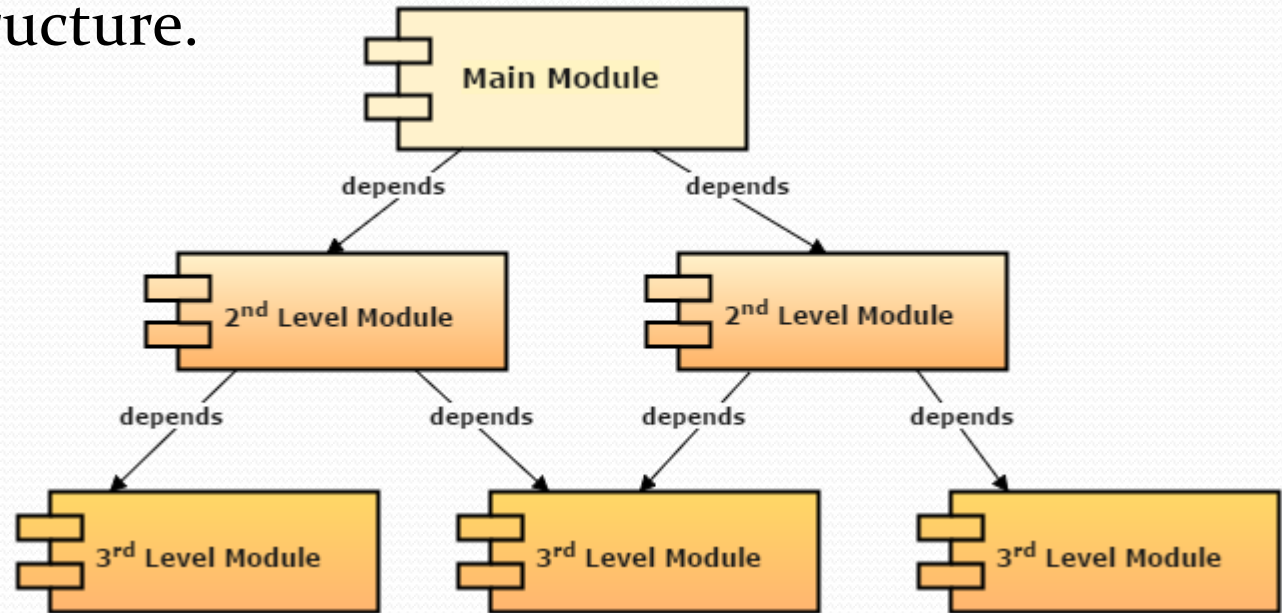
Examine DIP



- Without DIP, Object A in Package A refers Object B in Package B. With DIP, an Interface A is introduced as an abstraction in Package A. Object A now refers Interface A and Object B inherits from Interface A.
- Both Object A and Object B now depends on Interface A, the abstraction.
- It inverted the dependency that existed from Object A to Object B into Object B being dependent on the abstraction (Interface A).

Examine DIP - traditional systems

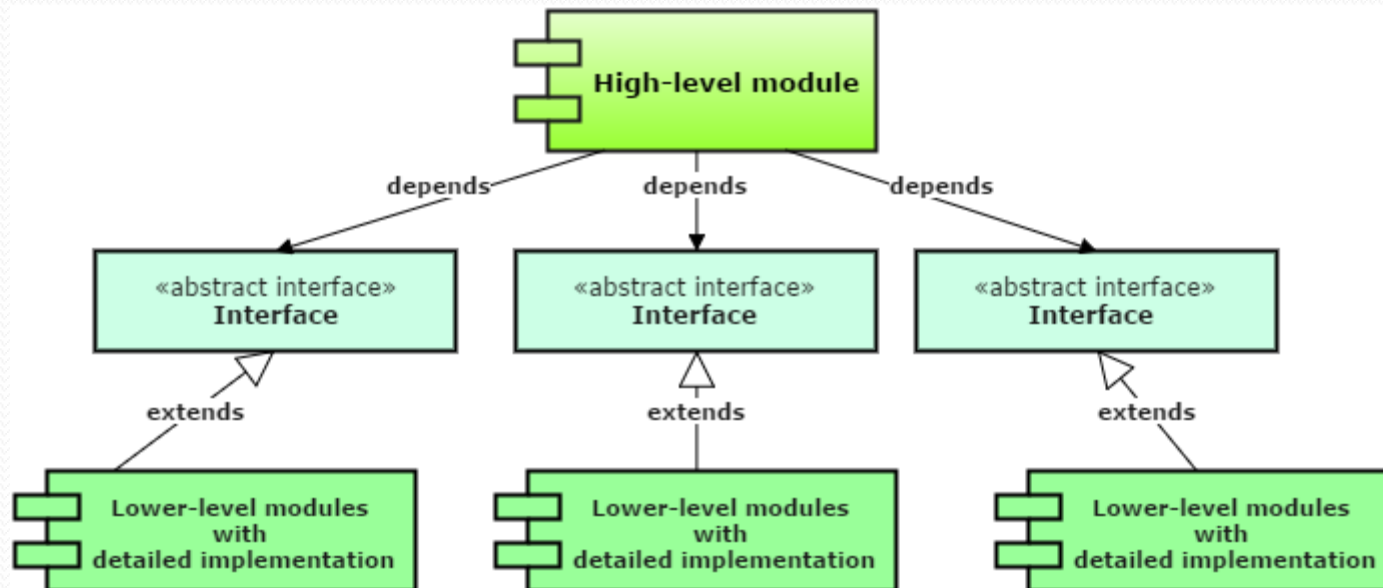
- The traditional procedural systems have their dependencies organized. In procedural systems, higher level modules depend on lower level modules to fulfil their responsibilities. The diagram below shows the procedural dependency structure.



- The traditional systems have a top-down dependency structure with the Main or Root module depending on 2nd Level Modules which in turn depend on 3rd Level Modules.

Examine DIP - dependency structure for OO systems

- **The Dependency Inversion Principle, however, advocates that the dependency structure should rather be inverted when designing object-oriented systems.** Have a look at the diagram below showing dependency structure for object-oriented systems



Ex 4 - DIP

- See the example document provided you to as week07c_DIP_LightBulbExample.pdf
- Happy reading!