The principles of OO design :

SRP - Single Responsibility Principle

-> Principle: A class should have only one reason to change.

-> Responsibility = ”a reason to change”

-> If a class has more than one responsibility, then the responsibilities become coupled.

-> The cohesion is low if the module does several things

-> Cohesion should be high

OCP - Open-Closed Principle

--> Open for Extension (new functions are added)

--> Closed for Modification (existing code is

unchanged)

-> What this really means is that you should (re)design so that change leads to extending, not modifying existing code

LSP - Liskov Substitution Principle

-> The key of OCP: Abstraction and Polymorphism

--> Implemented by inheritance

--> How do we measure the quality of inheritance?

-> "If for each object ob1 of type S there is an object ob2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when ob1 is substituted for ob2 then S is a subtype of T." ~ B. Liskov, 1988

-> Subtypes must be substitutable for their base types.

ISP - Interface-Segregation Principle

-> ISP deals with the disadvantages of “fat” interfaces.

--> Clients should not be forced to depend on methods that they do not use.

-> Avoid classes with too many responsibilities (classes whose interfaces are not cohesive).

-> Break interfaces up into cohesive groups of methods, each serving a certain kind of clients.

Example

// Bad example

interface IWorker {

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

// Good example
interface IWorker extends Feedable, Workable { }
interface IWorkable { public void work();}
interface IFeedable { public void eat();}
class Worker implements IWorkable, IFeedable{
public void work() { // ....working }
public void eat() { //.... eating in launch break }
}
class Robot implements IWorkable{
public void work() { // ....working }
}
class SuperWorker implements IWorkable, IFeedable{
public void work() { //.... working much more }
```

public void eat() { //.... eating in launch break}

}

class Manager {

Workable worker;

public void setWorker(Workable w) { worker=w; }

public void manage() { worker.work();}

}

DIP - Dependency-Inversion Principle

-> 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.

-> Structured Analysis and Design tend to create software structures in which high- level modules depend on low-level modules, and in which policy depends on detail.

-> It is the high-level modules that contain the important policy decisions and business models of an application.

DIP – Violating DIP

-> If the business depends on concrete services in the service layer and the services depends on concrete utilities in the utility layer, the business depends

transitively on the utilities.

-> This is very unfortunate because changes in low level modules have effect on high-level modules.

-> High-level modules will be difficult to reuse in other contexts

DIP – Conforming to DIP

-> You should invert the dependencies by using interfaces declared in the upper layer (the client "owns" the interface).

-> Now, the business no longer depends on a concrete service and can be reused with different implementations of the service.

-> NOTE: The book uses a different naming convention for the interfaces.

Example

//Bad example

class Worker {

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

//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 m_worker;
public void setWorker(IWorker w) { m_worker=w;}
public void manage() { m_worker.work();}
}
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 m_worker;

public void setWorker(IWorker w) { m_worker=w;}

public void manage() { m_worker.work();}

}
```

Package Design:

Cohesion:

The **Reuse-Release Equivalence Principle**

-> The granule of reuse is the granule of release.

The **Common-Reuse Principle**

-> The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

The **Common-Closure Principle**

-> The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package and no other packages.

Coupling:

The **Acyclic-Dependencies Principle**

-> Allow no cycles in the package-dependency graph

The **Stable-Dependencies Principle**

-> Depend in the direction of stability.

The **Stable-Abstractions Principle**

-> A package should be as abstract as it is stable.

-> Abstractness = The number of classes in the package / The number of abstract classes in the package

4 + 1 Model:

Logical View -> Development View

\|/ Scenario \|/

->Process View -> Physical View