Group 4:

* Ortiz Muñoz, Marlon Jahir
* Pachacama Tipan, Alexander Paul
* Ponce Chuqui, Camilo Alejandro
* Quichimbo Simba, Andrea Estefania
* Quilumbaquin Chicaiza, Karen Nicole
* Sarmiento Sanchez, Cristopher Josue
* Sigcha Manosalvas, Josselyn Susana

***MODULARITY***

**What is?**

Modularity exists in a software system when it is comprised of loosely coupled and cohesive components that isolate each significant or changeable design decision in one component and ensure that related ideas are as close as possible.

**The Consequences are:**

Understandability, testability, maintainability, reliability, security, extensibility, and reuse. Helps identify loosely coupled units of work.

**Adherence Criteria:**

• Localization of design decisions

• Low Coupling

• High Cohesion

• Modular or Extended Modular Reasoning

• Congruency

**LOCALIZATION OF DESIGN DECISIONS**

• Localization of Design Decisions involves

• Identifying an individual design decision and then

• Implementing that decision in one place

• Some IDE’s, like IntelliJ, flag duplicate code in many instances

• Poor localizations of design decisions causes poor modularization, and directly contributes to poor maintainability and reuse.

• Compare the following two variations of “Shapes”

**LOW COUPLING:**

-Interdependence between software components.

- When there is high coupling, causes a ripple changes in other components.

- In is low coupling, then a developer can make change or extension with minimal impact on other components.

- Developers should aim for low coupling.

**COUPLING**

Some causes of coupling, in an approximate high-to-low order

• Content coupling: (as it is called pathological coupling): a module is subject to

concepts encapsulated in another module instead of a summary

• Common coupling: modules share un-encapsulated global data

• Control coupling: one module controls the flow of another

• Stamp coupling: Module share a composite data structure but only need part of it

• Data coupling: The module shares elementary and individual data parts, by means of

parameter passing

• Message coupling: State is decentralized and components communication only via messages passing or method calls

**COHESION:**

• Coincidental cohesion: components grouped together arbitrarily   
• Logical cohesion: components grouped together because they are logically categorized to do the same kind of thing  
 • Temporal cohesion: components grouped together because they process close together in time   
• Procedural cohesion: components grouped together because they always follow a certain sequence of execution  
 • Communicational/informational cohesion: components grouped together because they operate on the same data   
• Sequential cohesion: components grouped together because the output of one is the input of another   
• Functional cohesion: components grouped together because they all contribute to a single well defined task

**HIGH COHESION**

• It is the degree to which the elements of a component

they are together

• Cohesion measures the strength of the relationship between pieces of

functionality with a given component.

• In good modularity, developers aim for high cohesion.

• High cohesion leads to:

Reduction of module complexity, greater maintenance capacity and greater reuse

**MODULAR REASONING**

Modular reasoning: a schedule can understand everything about a

method by looking at that method and the data members of its class.

• Extended modular reasoning: To understand a method, A, a developer has

to find the methods that call A and those that A calls.

• Overall reasoning: to understand a method, a developer may have to search

beyond the items listed for overall reasoning.

*“For good modularity, a developer should try design for modular or extended modular reasoning”*.

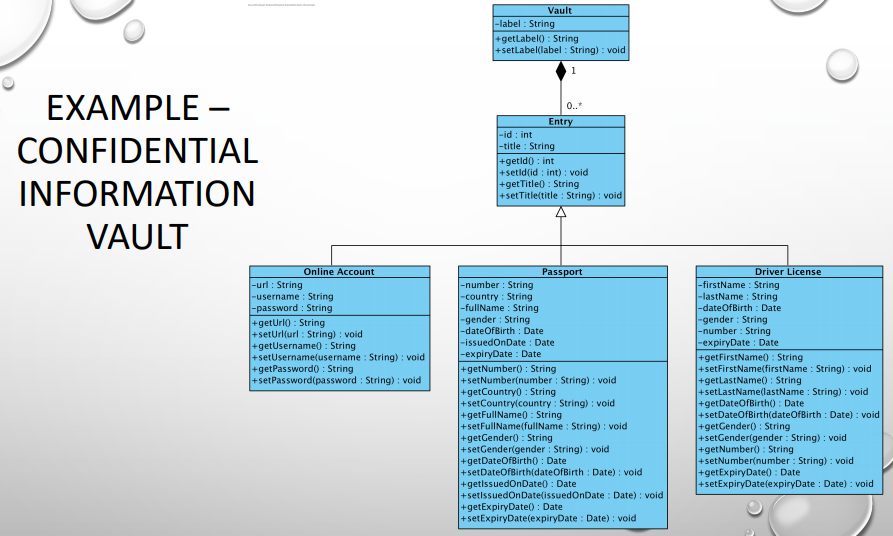
**CONGRUENCY:**

**\***In OO software development**.**

\* Its defined properties match the common properties of all its object.

\* If a class has one or more properties that  
are not properties for some subset, **it is an overstatement.**

\***There is an understatement**, if the objects of the class all have some  
common property not defined for the class.



**COMMERCE**

• Localization of design decisions and high cohesion lead to many grain components, which are good for capacity, extensibility and reusability, but it can make readability difficult.

• Modularity will not guarantee desirable features, for example, understandability, testability, maintainability, reliability, security,extensibility and reuse.

• Lack of modularity will compromise desirable features.

• Adherence or violation of the modularity principle typically affects

multiple components.

**CODE EXAMPLE**

This is a hello world compiled and run as modularity.

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | holamundo  ├── Makefile  └── src  └── holamundo  ├── com  │ └── ejemplo  │ └── HolaMundo.java  └── module-info.java  Examples for other modules. |

**// nombre del módulo. open permite la reflexión en todo el módulo**

**open module com.ejemplo**

**{**

**// exporta un paquete para que otros módulos accedan a sus paquetes públicos**

**exports com.apple;**

**// indica una dependencia con el módulo com.orange**

**requires com.orange;**

**// indica una dependencia con com.banana. el 'static' hace que la dependencia**

**// sea obligatoria durante compilación pero opcional durante ejecución**

**requires static com.banana;**

**// indica una dependencia al módulo com.berry y sus dependencias**

**requires transitive com.berry;**

**// permite reflexión en el módulo com.pear**

**opens com.pear;**

**// permite reflexión en el paquete com.lemon pero solo desde el módulo com.mango**

**opens com.lemon to com.mango;**

**// expone el tipo MyImplementation que implementa el servicio MyService**

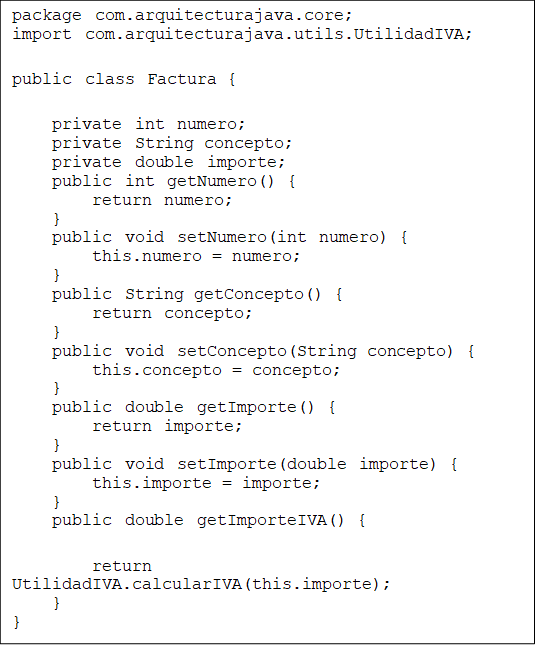
**provides com.service.MyService with com.consumer.MyImplementation**

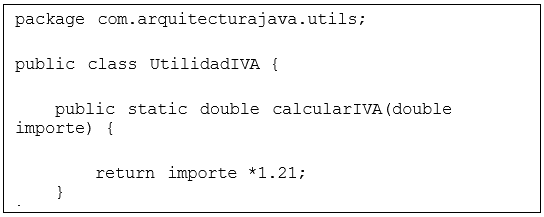
**// usa el servicio com.service.MyService**

**uses com.service.MyService**

**}**

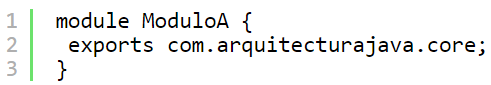
Build a Utility Project. A utility project defines a library or JAR. In this project we are going to include three files (Invoice, Utilities and module-info).





There are two related java classes located in different packages (core and utils).

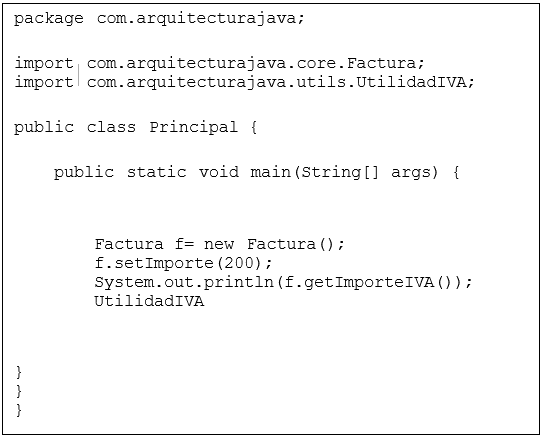
The information contained in the file that is responsible for managing modules is:



We look at the structure of the module.

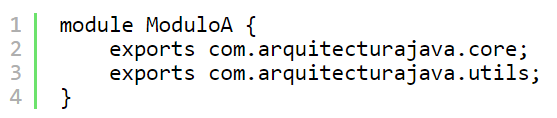


It is a module that has no dependencies but as a peculiarity it does not export all packages. Only the core package is exported, which is the one that contains the Invoice class. It is time to use our library in another Java project that has a main file.

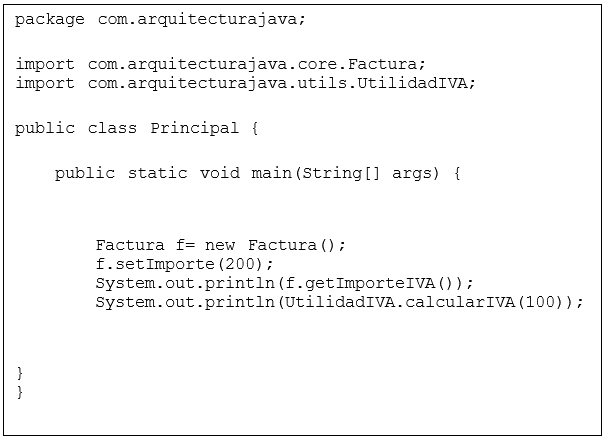


In order to use the other package we had to add the module dependency to our main project.

We reconsider if the encapsulation we have made of our packages is the most correct. In this case it is evident that not. VAT calculation is an operation that can be used by other libraries or by the main program. Therefore we can modify our module and publish it.



In this way we will be able to use the VAT calculation code in a main program.



**Questionnaire**

* The order of the coupling causes is correct

1.- Content docking

2.- Common coupling

3.- Control coupling

4.- Seal coupling

5.- Data coupling

6.- Message coupling

True (x ) False ( )

* Select the sentence that does not an adherence criteria of modulation

1. Localization of design decisions
2. Increased maintainability
3. High Cohesion
4. Modular or Extended Modular Reasoning

* Design decisions need to be “hidden” from the users of the component in which they are placed – this is actually ……

a. Constructor

b. Encapsulation

c. Replace constructor

d. None

* Components grouped together because they are logically categorized to do the same kind of thing

a. Temporal cohesion

b. Procedural cohesion

c. Logical cohesion

d. Functional cohesion

* What principle of modularity usually affects multiple components?

A. comprehension

B. adhesion

C. rape

D. legibility