Practica 2:

**Design principles y Design Patterns**

● SOLID, es un acrónimo de 5 importantes principios de la programación orientada a objetos. Explicar y brindar ejemplos.

**S: Single responsibility principle o Principio de responsabilidad única**

Establece que una clase, componente o microservicio debe ser responsable de una sola cosa, debe tener una única responsabilidad. Si por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.

Existe una frase: “Has una cosa y hazlo bien”. Esta frase aplica a este principio.

Martin Fowler define como responsabilidad a la “razón del cambio” y establece que una clase o componente debería tener una y solo una razón para cambiar.

class Coche {

String marca;

Coche(String marca){ this.marca = marca; }

String getMarcaCoche(){ return marca; }

void guardarCocheDB(Coche coche){ ... }

}

¿Por qué este código viola el principio de responsabilidad única?

La clase Coche permite tanto el acceso a las propiedades de la clase como a realizar operaciones sobre la BBDD, por lo que la clase ya tiene más de una responsabilidad.

Supongamos que debemos realizar cambios en los métodos que realizan las operaciones a la BBDD. En este caso, además de estos cambios, probablemente tendríamos que tocar los nombres o tipos de las propiedades, métodos, etc, cosa que no parece muy eficiente porque solo estamos modificando cosas que tienen que ver con la BBDD.

Para evitar esto, debemos separar las responsabilidades de la clase, por lo que podemos crear otra clase que se encargue de las operaciones a la BBDD:

class Coche {

String marca;

Coche(String marca){ this.marca = marca; }

String getMarcaCoche(){ return marca; }

}

class CocheDB{

void guardarCocheDB(Coche coche){ ... }

void eliminarCocheDB(Coche coche){ ... }

}

**O: Open/closed principle o Principio de abierto/cerrado**

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertos para su extensión, pero cerrados para su modificación. Nosotros nos podemos asegurar que nuestro código cumple con este principio usando herencia o implementando interfaces, permitiendo a las clases sustituirse polimórficamente entre ellas.

class Coche {

String marca;

Coche(String marca){ this.marca = marca; }

String getMarcaCoche(){ return marca; }

}

Si quisiéramos iterar a través de una lista de coches e imprimir sus marcas por pantalla:

public static void main(String[] args) {

Coche[] arrayCoches = {

new Coche("Renault"),

new Coche("Audi")

};

imprimirPrecioMedioCoche(arrayCoches);

}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

if(coche.marca.equals("Renault")) System.out.println(18000);

if(coche.marca.equals("Audi")) System.out.println(25000);

}

}

Esto no cumpliría el principio abierto/cerrado, ya que si decidimos añadir un nuevo coche de otra marca:

Coche[] arrayCoches = {

new Coche("Renault"),

new Coche("Audi"),

new Coche("Mercedes")

};

También tendríamos que modificar el método que hemos creado anteriormente:

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

if(coche.marca.equals("Renault")) System.out.println(18000);

if(coche.marca.equals("Audi")) System.out.println(25000);

if(coche.marca.equals("Mercedes")) System.out.println(27000);

}

}

Como podemos ver, para cada nuevo coche habría que añadir nueva lógica al método precioMedioCoche().

Para que cumpla con este principio podríamos hacer lo siguiente:

abstract class Coche {

// ...

abstract int precioMedioCoche();

}

class Renault extends Coche {

@Override

int precioMedioCoche() { return 18000; }

}

class Audi extends Coche {

@Override

int precioMedioCoche() { return 25000; }

}

class Mercedes extends Coche {

@Override

int precioMedioCoche() { return 27000; }

}

public static void main(String[] args) {

Coche[] arrayCoches = {

new Renault(),

new Audi(),

new Mercedes()

};

imprimirPrecioMedioCoche(arrayCoches);

}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

System.out.println(coche.precioMedioCoche());

}

}

Otro ejemplo:

class Post

{

void CreatePost(Database db, string postMessage)

{

if (postMessage.StartsWith("#"))

{

db.AddAsTag(postMessage);

}

else

{

db.Add(postMessage);

}

}

}

La clase anterior no cumple con este principio. La siguiente modificación cumple con el requerimiento:

class Post

{

void CreatePost(Database db, string postMessage)

{

db.Add(postMessage);

}

}

class TagPost : Post

{

override void CreatePost(Database db, string postMessage)

{

db.AddAsTag(postMessage);

}

}

**L: Liskov substitution principle o Principio de sustitución de Liskov**

Este principio establece que si S es subtipo de T, entonces los objetos de tipo T podrían ser reemplazados con objetos de tipo S.

Declara que una subclase debe ser sustituible por su superclase, y si al hacer esto, el programa falla, estaremos violando este principio.

Cumpliendo con este principio se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.

public static void imprimirNumAsientos(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

if(coche instanceof Renault)

System.out.println(numAsientosRenault(coche));

if(coche instanceof Audi)

System.out.println(numAsientosAudi(coche));

if(coche instanceof Mercedes)

System.out.println(numAsientosMercedes(coche));

}

}

imprimirNumAsientos(arrayCoches);

Esto viola tanto el principio de substitución de Liskov como el de abierto/cerrado. El programa debe conocer cada tipo de Coche y llamar a su método numAsientos() asociado.

Así, si añadimos un nuevo coche, el método debe modificarse para aceptarlo.

// ...

Coche[] arrayCoches = {

new Renault(),

new Audi(),

new Mercedes(),

new Ford()

};

public static void imprimirNumAsientos(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

if(coche instanceof Renault)

System.out.println(numAsientosRenault(coche));

if(coche instanceof Audi)

System.out.println(numAsientosAudi(coche));

if(coche instanceof Mercedes)

System.out.println(numAsientosMercedes(coche));

if(coche instanceof Ford)

System.out.println(numAsientosFord(coche));

}

}

imprimirNumAsientos(arrayCoches);

Para que este método cumpla con el principio, seguiremos estos principios:

public static void imprimirNumAsientos(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

System.out.println(coche.numAsientos());

}

}

imprimirNumAsientos(arrayCoches);

Ahora al método no le importa el tipo de la clase, simplemente llama al método numAsientos() de la superclase. Solo sabe que el parámetro es de tipo coche, ya sea Coche o alguna de las subclases.

abstract class Coche {

// ...

abstract int numAsientos();

}

class Renault extends Coche {

// ...

@Override

int numAsientos() {

return 5;

}

}

Otro ejemplo:

class Post

{

void CreatePost(Database db, string postMessage)

{

db.Add(postMessage);

}

}

class TagPost : Post

{

override void CreatePost(Database db, string postMessage)

{

db.AddAsTag(postMessage);

}

}

class MentionPost : Post

{

void CreateMentionPost(Database db, string postMessage)

{

string user = postMessage.parseUser();

db.NotifyUser(user);

db.OverrideExistingMention(user, postMessage);

base.CreatePost(db, postMessage);

}

}

class PostHandler

{

private database = new Database();

void HandleNewPosts() {

List<string> newPosts = database.getUnhandledPostsMessages();

foreach (string postMessage in newPosts)

{

Post post;

if (postMessage.StartsWith("#"))

{

post = new TagPost();

}

else if (postMessage.StartsWith("@"))

{

post = new MentionPost();

}

else {

post = new Post();

}

post.CreatePost(database, postMessage);

}

}

}

El ejemplo anterior no cumple con el principio, debemos realizar el siguiente cambio.

class MentionPost : Post

{

override void CreatePost(Database db, string postMessage)

{

string user = postMessage.parseUser();

db.NotifyUser(user);

db.OverrideExistingMention(user, postMessage);

base.CreatePost(db, postMessage);

}

}

**I: Interface segregation principle o Principio de segregación de la interfaz**

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este cliente no usa, pero que otros clientes sí usan, este cliente estará siendo afectado por los cambios que fuercen otros clientes en dicha interfaz.

interface IAve {

void volar();

void comer();

}

class Loro implements IAve{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//..

}

}

class Tucan implements IAve{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//..

}

}

Hasta aquí todo bien. Pero ahora imaginemos que queremos añadir a los pingüinos. Estos son aves, pero además tienen la habilidad de nadar. Podríamos hacer esto:

interface IAve {

void volar();

void comer();

void nadar();

}

class Loro implements IAve{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//...

}

@Override

public void nadar() {

//...

}

}

class Pinguino implements IAve{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//...

}

@Override

public void nadar() {

//...

}

}

El problema es que el loro no nada, y el pingüino no vuela. Esto viola el principio de segregación de interfaz, ya que estas clases (los clientes) no tienen por qué depender de métodos que no usan.

interface IAve {

void comer();

}

interface IAveVoladora {

void volar();

}

interface IAveNadadora {

void nadar();

}

class Loro implements IAve, IAveVoladora{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//...

}

}

class Pinguino implements IAve, IAveNadadora{

@Override

public void nadar() {

//...

}

@Override

public void comer() {

//...

}

}

Puesto más simple: No agregue funcionalidad a una interface existente, en vez de esto cree una nueva interface.

Otro ejemplo,

interface IPost

{

void CreatePost();

}

interface IPostNew

{

void CreatePost();

void ReadPost();

}

Correcto:

interface IPostCreate

{

void CreatePost();

}

interface IPostRead

{

void ReadPost();

}

**D: Dependency inversion principle o Principio de inversión de dependencia**

Este principio establece un desacoplamiento de los componentes de software mediante:

1. Los componentes de alto nivel no deberían depender de componentes de bajo nivel. Ambos deberían depender de abstracciones.
2. Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones.

class DatabaseService{

//...

Dato getDatos(){ //... }

}

class AccesoADatos {

private DatabaseService databaseService;

public AccesoADatos(DatabaseService databaseService){

this.databaseService = databaseService;

}

Dato getDatos(){

databaseService.getDatos();

//...

}

}

Imaginemos que en el futuro queremos cambiar el servicio de BBDD por un servicio que conecta con una API. ¿Qué cambios tendríamos que hacer? El inconveniente ocurre debido a que nuestro componente de alto nivel (AccesoADatos) depende de un componente de más bajo nivel (DatabaseService), violando así el principio de inversión de dependencias. El componente de alto nivel debería depender de abstracciones.

interface Conexion {

Dato getDatos();

void setDatos();

}

class AccesoADatos {

private Conexion conexion;

public AccesoADatos(Conexion conexion){

this.conexion = conexion;

}

Dato getDatos(){

conexion.getDatos();

}

}

class DatabaseService implements Conexion{

//...

@Override

public Dato getDatos() { //... }

@Override

public void setDatos() { //... }

}

● Sobre el patrón de diseño Singleton, explique: ○ ¿Porque se puede considerar que no cumple con el principio de Single Responsibility? ○ ¿Porque se considera que dificulta realizar tests de unidad?

Singleton pattern is violating this principle. Instead of clear separation of concerns, the class has two distinct responsibilities - the first is making sure that only one instance can be created. The second is core functionality of the class - accessing DB, managing some unique resource or whatever is it supposed to do.

The class having too many responsibilities and too many reasons to change is much harder to maintain as changes in one responsibility affect other responsibilities as well.

One of the main disadvantages of singletons is that they make unit testing very hard. They introduce global state to the application. The problem is that you cannot completely isolate classes dependent on singletons. When you are trying to test such a class, you inevitably test the Singleton as well. When unit testing, you want the class to be as loosely coupled with other classes as possible and all the dependencies of the class should be ideally provided externally (either by constructor or setters), so they can be easily mocked. Unfortunately, that is not possible with singletons as they introduce tight coupling and the class retrieves the instance on its own.

● Dependency Injection design pattern ○ Explique el patrón ○ ¿Cuál es la diferencia entre Dependency Inversion y Dependency Injection? ○ Brinde ejemplos de código donde se aplique.

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 4 types of classes.

The service you want to use.

The client that uses the service.

An interface that’s used by the client and implemented by the service.

The injector which creates a service instance and injects it into the client.

*Constructor Injection*

we provide the dependency through the constructor, this is called a constructor injection.

public class CustomerBusinessLogic

{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)

{

\_dataAccess = custDataAccess;

}

public CustomerBusinessLogic()

{

\_dataAccess = new CustomerDataAccess();

}

public string ProcessCustomerData(int id)

{

return \_dataAccess.GetCustomerName(id);

}

}

public interface ICustomerDataAccess

{

string GetCustomerData(int id);

}

public class CustomerDataAccess: ICustomerDataAccess

{

public CustomerDataAccess()

{

}

public string GetCustomerName(int id)

{

//get the customer name from the db in real application

return "Dummy Customer Name";

}

}

public class CustomerService

{

CustomerBusinessLogic \_customerBL;

public CustomerService()

{

\_customerBL = new CustomerBusinessLogic(new CustomerDataAccess());

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

*Property Injection*

The dependency is provided through a public property:

public class CustomerBusinessLogic

{

public CustomerBusinessLogic()

{

}

public string GetCustomerName(int id)

{

return DataAccess.GetCustomerName(id);

}

public ICustomerDataAccess DataAccess { get; set; }

}

public class CustomerService

{

CustomerBusinessLogic \_customerBL;

public CustomerService()

{

\_customerBL = new CustomerBusinessLogic();

\_customerBL.DataAccess = new CustomerDataAccess();

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

*Method Injection*

In the method injection, dependencies are provided through methods. This method can be a class method or an interface method.

interface IDataAccessDependency

{

void SetDependency(ICustomerDataAccess customerDataAccess);

}

public class CustomerBusinessLogic : IDataAccessDependency

{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic()

{

}

public string GetCustomerName(int id)

{

return \_dataAccess.GetCustomerName(id);

}

public void SetDependency(ICustomerDataAccess customerDataAccess)

{

\_dataAccess = customerDataAccess;

}

}

public class CustomerService

{

CustomerBusinessLogic \_customerBL;

public CustomerService()

{

\_customerBL = new CustomerBusinessLogic();

((IDataAccessDependency)\_customerBL).SetDependency(new CustomerDataAccess());

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

● Explicar brevemente la diferencia entre MVC, MVP, MVVC, MVW.

**Web Applications**

● Explique los siguientes tipos de aplicaciones web

○ Dynamic Web Applications

A server-side dynamic web page is a web page whose construction is controlled by an application server processing server-side scripts. In server-side scripting, parameters determine how the assembly of every new web page proceeds, including the setting up of more client-side processing. Such web pages are often created with the help of server-side languages such as ASP, ColdFusion, Go, JavaScript, Perl, PHP, Ruby, Python, WebDNA and other languages

○ Single Page Applications

A single-page application (SPA) is a web application or web site that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server. This approach avoids interruption of the user experience between successive pages, making the application behave more like a desktop application. In a SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load,[1] or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions.

There are various techniques available that enable the browser to retain a single page even when the application requires server communication.

Web browser JavaScript frameworks, such as AngularJS, Ember.js, ExtJS, Knockout.js, Meteor.js, React and Vue.js have adopted SPA principles.

○ Progressive Web Applications

By Google (<https://developers.google.com/web/progressive-web-apps/?hl=en>):

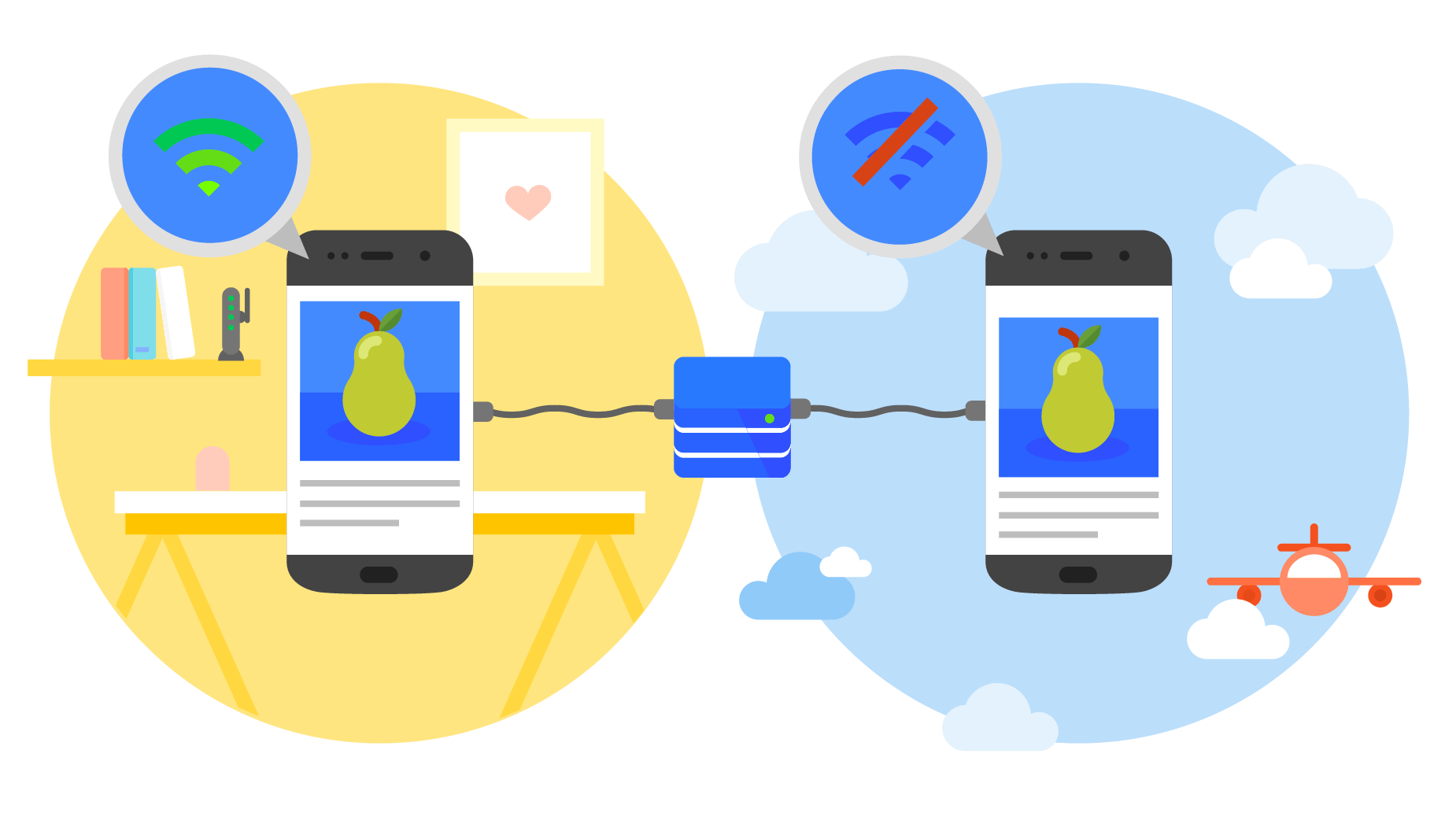
Progressive Web Apps are user experiences that have the reach of the web, and are:

Reliable - Load instantly and never show the downasaur, even in uncertain network conditions.

Fast - Respond quickly to user interactions with silky smooth animations and no janky scrolling.

Engaging - Feel like a natural app on the device, with an immersive user experience.

This new level of quality allows Progressive Web Apps to earn a place on the user's home screen.



**Java Build Tools**

● Describir:

○ Apache Ant

Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications (<https://ant.apache.org/>)

○ Apache Maven (<https://maven.apache.org/>)

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

○ Gradle (<https://gradle.org/>)

Gradle is an open-source build automation tool focused on flexibility and performance. Gradle build scripts are written using a Groovy or Kotlin DSL. Read about Gradle features to learn what is possible with Gradle.

● Apache Maven

○ ¿Qué es el archivo POM de maven?

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. Examples for this is the build directory, which is target; the source directory, which is src/main/java; the test source directory, which is src/test/java; and so on. When executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal.

Some of the configuration that can be specified in the POM are the project dependencies, the plugins or goals that can be executed, the build profiles, and so on. Other information such as the project version, description, developers, mailing lists and such can also be specified.

○ Explique brevemente el Build Lifecycle de maven y cada una de sus etapas.

(<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>)

A Build Lifecycle is a well-defined sequence of phases, which define the order in which the goals are to be executed. Here phase represents a stage in life cycle. As an example, a typical Maven Build Lifecycle consists of the following sequence of phases.

|  |  |  |
| --- | --- | --- |
| **Phase** | **Handles** | **Description** |
| prepare-resources | resource copying | Resource copying can be customized in this phase. |
| validate | Validating the information | Validates if the project is correct and if all necessary information is available. |
| compile | compilation | Source code compilation is done in this phase. |
| Test | Testing | Tests the compiled source code suitable for testing framework. |
| package | packaging | This phase creates the JAR/WAR package as mentioned in the packaging in POM.xml. |
| install | installation | This phase installs the package in local/remote maven repository. |
| Deploy | Deploying | Copies the final package to the remote repository. |

● Crear un proyecto Java del tipo web application utilizando Maven

**Versionado de código**

● ¿Qué son las herramientas para el versionado de código?

Una herramienta de versionado me permite tener una trazabilidad de los cambios realizados en los componentes de software y por medio de esta trazabilidad tener conocimiento de cuales fueron los cambios realizados, en que momento, deshacer cambios, comparar cambios en el componente. Estas herramientas brindan soporte en la etapa de Diseño de la configuración del software.

● ¿Cuál es la diferencia entre herramientas de versionado centralizado y descentralizado?

Centralizados:

Existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos. Ejemplos son CVS, Subversion

Desentralizados: Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: Git

● ¿Cuáles son las diferencias en las herramientas de versionado de código Subversion (SVN) y GIT?

Que tienen distintas arquitecturas de almacenamiento:

Centralizado: SVN

Desentralizado: GIT

● Explique que es un workflow o branching model

Es el flujo de actividades a realizar a la hora de aplicar cambios a los componentes del software. El más común es el Gitflow Workflow.



● Explique en que consiste las técnica de code review / pull request

Code Review: It represents the process of reviewing someone’s code before it gets merged into the existing code base. Whenever a developer wants to perform any sort of change, he or she must create a Pull Request (PR).

Pull Request represents all of the changes made against the current state of the code base. Other developers in the team will then comment on the work done and accept/refuse the change.

● Sobre el proyecto anteriormente creado:

○ Inicializar un repositorio Git

a. Go to repositories tab

b. Select New and write a Repository name. Example: Unnoba-Repository

c. Create a folder. Example: Git-Proyects

d. Go to the previously created folder and execute git clone <https://github.com/ndigrazia/Unnoba-Repository.git>

f. Go to D:\git\Git-Proyects\Unnoba-Repository>

○ Determinar tipos de archivos o directorios que convienen no ser versionados (según el tipo de aplicación y lenguaje) e incluirlo .gitignore.

○ realizar un commit inicial con la estructura inicial del proyecto

g.

echo "# Unnoba-Repository" >> README.md

git add .

git commit -m "first commit"

git push -u origin master

○ crear la rama develop y sobre ella crear la clase User (con atributos email y password y sus respectivos getters y setters) en package ar.edu.unnoba.poo.2019.model.

h.

git branch develop

git push -u origin develop

git checkout develop

Switch Workspace to D:\git\Git-Proyects\Unnoba-Repository

Create Proyect Maven on Eclipse

<groupId>unnoba</groupId>

<artifactId>unnoba-datastore</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

Create User class

NOTE:

list remote branches: git branch -r

You can check them out as local branches with: git checkout -b LocalName origin/remotebranchname

○ realizar un nuevo commit para incluir esta modificaciones.

In develop:

Git add .

git commit -m "User commit"

git push -u origin develop

○ cambiar al branch master

git checkout master

○ realizar un merge con la rama develop

git merge develop

Git add .

git commit -m "User Mege commit"

git push -u origin master

○ crear un tag especificando la version 0.1

git tag -a v0.1 -m "my version 0.1"

git push --follow-tags