"AÑO DEL DIÁLOGO Y LA RECONCILIACIÓN NACIONAL"

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN
ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN


# PROYECTO FINAL: IMPLEMENTACION DE UN ERP

CIENCIA DE LA COMPUTACIÓN II

DOCENTE: ING ALVARO MAMANI ALIAGA

INTEGRANTES:

BARRIOS CORNEJO SELENE
PILCO PANCCA LUZ

AREQUIPA- PERÚ

2018

## INTRODUCCIÓN

En la actualidad, la mayoría de las organizaciones se están enfrentando a una era en la que los cambios tecnológicos e informáticos impactan dentro y fuera de la organización; y cuya eficiente gestión, permite obtener ventajas comparativas con las cuales es posible lograr un posicionamiento favorable dentro de un mercado cada vez más cambiante y competitivo.

Por lo antes mencionado, los responsables de la administración de las instituciones que se resistan a alinear el funcionamiento organizacional con los cambios tecnológicos, se verán en la triste situación de ir perdiendo clientes y en el peor de los casos a desaparecer del mercado. Los empresarios y altos mandos de una organización deben tomar el riesgo de proponer cambios tecnológicos e informáticos en sus organizaciones , ya que los beneficios que se pueden obtener son mayores que los costos que se generan.

Las organizaciones que tengan adecuadamente estructurada su información e integrada en una base de datos con acceso inmediato a ella, tienen garantizada una gran ventaja competitiva, ya que las decisiones se podrán tomar con mayor confiabilidad y en el menor tiempo posible. Para lograr lo antes mencionado, se pueden utilizar lo que se conoce como Sistemas de Gestión Empresarial , que apoyan al análisis de la información como un recurso fundamental de la organización; además de controlar, planificar, organizar, dirigir y automatizar todos los aspectos operativos de la empresa, brinda soporte para la toma de
decisiones, así como ventajas estratégicas.

Es importante señalar que estos sistemas de gestión empresarial buscan acortar la distancia que existe entre la empresa, los clientes y los proveedores a fin de crear una comunicación constante y responder a sus requerimientos y necesidades con rapidez, calidad y eficiencia.

Como parte de los Sistemas de Gestión Empresarial se encuentran los denominados Sistemas de Planificación de Recursos Empresariales (ERP)-tema principal de este proyecto-, los cuales han tenido mucha aceptación para su desarrollo e implementación, considerando que facilitan los procesos de la organización, mantienen un orden e integración en sus datos y el acceso a la información es eficiente.

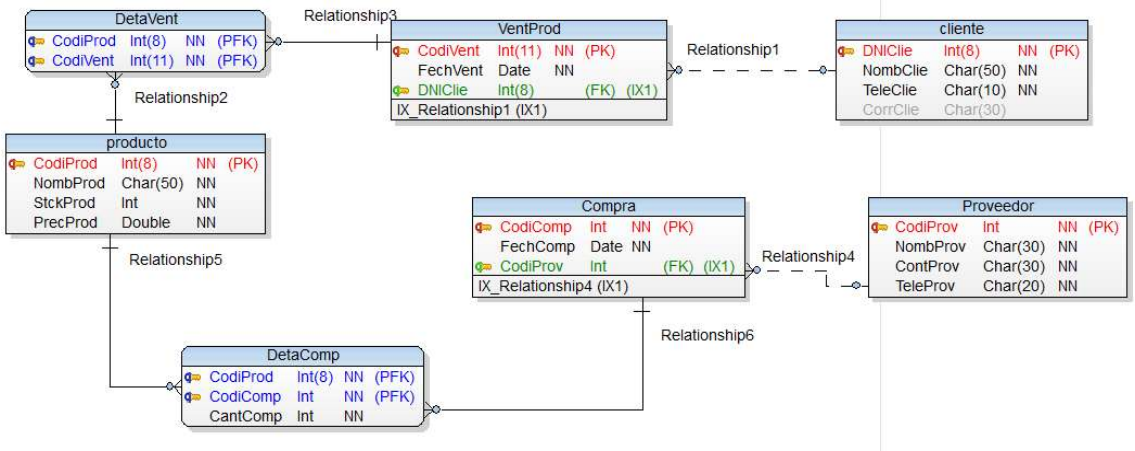
El objetivo del trabajo que se reporta en este documento consistió en: aportar conocimientos acerca del proceso relacionado con la implementación de un ERP en una Institución de Salud. Para fundamentar conceptualmente, se obtuvo información de: artículos de revistas de investigación científica, páginas de Internet de fuentes confiables y de algunos libros relacionados con la temática. Sin embargo, el principal aporte que se hace en este trabajo, se relaciona principalmente con la experiencia adquirida al participar activamente en la fase de implementación del sistema ERP.

## 1.-FINALIDAD DEL PROYECTO:

1. Creacion de un sistema de recursos empresariales utilizando patrones de diseño y recursos aprendidos en clase tales como
2. Crear la interfaz con un aplicativo llamado interfaz QT, el cual realiza formularios del ERP de forma dinamica.
3. Creacion de una base de datos en MySql.
4. Creacion de los esquemas de la base de datos con TOAD.

## 2.-ESQUEMA DE LA BASE DE DATOS:

MODULO DE VENTAS




## 3.-BUSQUEDA DE INFORMACION

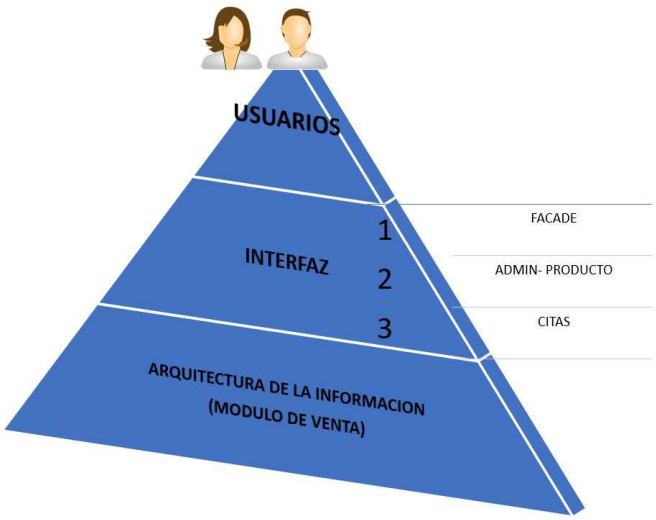
Acotamos el problema desde lo general a lo particular. Algunas de los puntos que se tratarán son:

**Principios de funcionamiento:**

- Implementar modulos de venta y citas.
- Implementar clases que nos permita administrar los clientes , productos( MEDICAMENTOS) y citas.

## 4.-MODELANDO LA ARQUITECTURA DE LA APLICACION

El propósito de un modelo es reducir las ambigüedades que se producen en las descripciones con el lenguaje natural y ayudar al usuario y sus colegas a visualizar el diseño y a analizar diseños alternativos. El modelo debe usarse conjuntamente con otros documentos o explicaciones. Por sí mismo, un modelo no constituye una especificación completa de la arquitectura.




## 5.-CONOCIMIENTOS NECESARIOS Y CUESTIONES A TENER EN CUENTA

### QT CREATOR

Qt Creator es un IDE multi plataforma programado en C++, JavaScript y QML creado por Trolltech el cual es parte de SDK para el desarrollo de aplicaciones con Interfaces Gráficas de Usuario (GUI por sus siglas en inglés) con las bibliotecas Qt.

Qt Creator se centra en proporcionar características que ayudan a los nuevos usuarios de Qt a aprender, también aumentar la productividad.

- Editor de código con soporte para C+, QML y ECMAscript.
- Herramientas para la rápida navegación del código.
- Resaltado de sintaxis y auto-completado de código.
- Control estático de código y estilo a medida que se escribe.

- Soporte para refactoring de código.
- Ayuda sensitiva al contexto.
- Plegado de código (code folding).
- Paréntesis coincidentes y modos de selección.

**Depurado Visual**

El depurador visual (visual debugger) para C++. Qt Creator muestra la información en bruto procedente de GDB de una manera clara y concisa.

- Interrupción de la ejecución del programa.
- Ejecución línea por línea o instrucción a instrucción.
- Puntos de interrupción (breakpoints).
- Examinar el contenido de llamadas a la pila (stack), los observadores y de la *variables locales y globales.

**MARIA DB**

MariaDB es un sistema de gestión de bases de datos derivado de MySQL con licencia GPL (General Public License).

**Software de terceras partes**

Hay bastantes paquetes propietarios y libres de terceras partes diseñados para MySQL que también están disponibles para integrarse con MariaDB. Algunos ejemplos son:

- DBEdit — una aplicación de administración libre para MariaDB y otras bases de datos.
- dbForge Studio for MySQL — aplicación propietaria de gestión de bases de datos MySQL compatible con MariaDB.
- Navicat — una serie de aplicaciones propietarias de gestión de bases de datos para Windows, Mac OS X y Linux.
- SQLyog — aplicación propietaria de gestión de bases de datos MySQL compatible con MariaDB para Windows y Linux.
- HeidiSQL — un cliente de fuente abierta y libre para MySQL, 100% compatible con MariaDB, incluido con el paquete MSI para Windows de MariaDB desde la versión 5.2.7.56 **(USADA)**
- phpMyAdmin — una aplicación web de administración libre para MySQL compatible con MariaDB.

**TOAD**

TOAD es una aplicación informática de desarrollo SQL y administración de base de datos, considerada una herramienta útil para los DBAs (administradores de base de datos).

**6.-PLANIFICACION**

| PROGRAMAS | VERSION |
|---|---|
| Qt Creator | 5.1 |
| MariaDB | 10.5 |
| TOAD | 6.4 |

**7.PROCESO DE ELABORACION**

1. Aprender el manejo de Qt Creator.
2. Diseñar e implementar nuestras respetivas clases, psra mejor entendimmiento realizar un diagrama de clases.
3. Diseñar la base de datos y generar un su respectivo diagrama con TOAD para el mejor manejo y comprension.
4. Realizar las respetivas conexiones con nuestra base de datos.

**8.SOFTWARE**

| CODIGO | EXPLICACION |
|---|---|
| `#ifndef ADMINBASE_H`<br>`#define ADMINBASE_H`<br><br>`#include <string>`<br>`using namespace std;`<br><br>`class AdminBase`<br>`{`<br>`public:`<br>`  static void Agregar(string);`<br>`  static void BuscarPorClave(int);`<br>`  static void BuscarTodos();`<br>`  static void Modificar(string);`<br>`  static void Eliminar(int);`<br>`};`<br><br>`#endif // ADMINBASE_H` | Esta clase no dice todas las funciones miembro que tiene todo nuestro ERP. |

```
class AdminCitas : public AdminBase
{
    public:
        static void Agregar(QString scita){
            AdminData data = AdminData::getInstance();
            data.execQuery(scita);
        }

        static QList<Cita*> BuscarTodos(){
            AdminData data = AdminData::getInstance();
            QSqlQuery query=data.execQuery("SELECT * FROM citas ;");
            QList<Cita*> lista;
            while (query.next()) {
                lista.append(new Cita(query.value(0).toInt(),
                                      query.value(1).toString(),
                                      query.value(2).toInt()
                                      )
                             );
            }
            return lista;
        }

};
```

La clase AdminCitas hereda de AdminBase y se encarga de administrar todas las citas.
La siguiente linea de codigo contiene el patron de diseño Singleton:

```
AdminData data = AdminData::getInstance();
```

QSttring scita es una sentencia para la base de datos , en Qt Creator se reduce a un QString.

Aqui observamos dos funciones miembro de la Clase:

1. Agregar -> agrega una cita
2. BuscarTodos ->busca una cita

```
}
static void Modificar(QString cliente){
    AdminData data = AdminData::getInstance();
    data.execQuery(cliente);
}
static void Eliminar(int clave){
    AdminData data = AdminData::getInstance();
    data.execQuery("DELETE FROM  cliente WHERE DNIClie = "+QString::number(clave)+";");
}
```

3.Modificar -> modifica los datos de una cita

4.Eliminar -> elimina una cita.

```
#include <QDebug>
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QString>
#include <QMessageBox>
#include <QSqlError>

using namespace std;

class AdminData
{
    private:
        QSqlDatabase data;
        AdminData();
    public:
        // singleton
        static AdminData& getInstance();
        QSqlQuery execQuery(QString);


};
```

En la parte superior observamos:
#include <QDebug>
Lista de todos los miembros, incluidos los miembros heredados.
#include <QSqlDatabase>
Sirve para la conexion con la base de datos.
#include <QSqlQuery>
Consulta a la base de datos.
#include <QString>
Proporciona una cadena de caracteres Unicode.
#include <QMessageBox>
Muestra los mensajes de error.
#include <QsqlError>
Muestra error si hay mal conexion con la base de datos.
CONEXION BASE DE DATOS!!!!
La clase AdminData se encuentra implementada con el patron de diseño Singleton,el cual se encarga de instanciar un objeto , podemos observar que nuestro constructor se encuentra en private.

```cpp
#include <QtDebug>
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QString>
#include <QMessageBox>
#include <QSqlError>

AdminData::AdminData()
{
    data = QSqlDatabase::addDatabase("QMYSQL");
    data.setHostName("localhost");
    data.setPort(3306);
    data.setDatabaseName("erp");
    data.setUserName("root");
    data.setPassword("root");
    data.open();
    qDebug() << "open Conection";
}

QSqlQuery AdminData::execQuery(QString statement){
    QSqlQuery query;
    query.exec(statement);

    return query;
}

AdminData& AdminData::getInstance(){
    static AdminData instance;
    return instance;
}
```

CONEXION BASE DE DATOS!!!!

Esta parte del codigo es sumamente importante porque es para la conexion de la base de datos , esa es la funcion de nuestra clase AdminData...en el constructor declaramos el puerto, nombre de la base de datos, el usuario y contrseña.

```cpp
class AdminProducto: public AdminBase
{
    public:
        static void Agregar(QString producto){
            //producto  =  sentencia insert
            AdminData data = AdminData::getInstance();
            data.execQuery(producto);
        }
        static QList<Producto*> BuscarPorClave(int clave){
           AdminData data = AdminData::getInstance();
           QSqlQuery query=data.execQuery("SELECT * FROM pro
           QList<Producto*> lista;
           while (query.next()) {
               lista.append(new Producto(query.value(0).toIn
                                         query.valu
                                         query.valu
                                         query.valu
                                         )
                                );
           }
           return lista;
        }
        static QList<Producto*> BuscarTodos(){
            AdminData data = AdminData::getInstance();
            QSqlQuery query=data.execQuery("SELECT * FROM pr
            QList<Producto*> lista;
```

La clase AdminData hereda de AdminBase y posee las funciones miembro :

1. Agregar
2. Buscar
3. Modificar
4. Eliminar

Podemos observar que el patron de diseño Singleton sigue presente en la misma linea de codigo mencionada anteriormente.

```
class Cliente
{
    private:
        int dni;
        QString Nombre;
        QString Telefono;
        QString Correo;

    public:
        Cliente(){

        }
        Cliente(int dni, QString Nombre, QString Tele
            this->dni = dni;
            this->Nombre =Nombre;
            this->Telefono=Telefono;
            this->Correo=Correo;
        }
        virtual ~Cliente(){

        }

        int Getdni() {
            return dni;
        }
        void Setdni(int val) {
```

Observamos aqui la clase Cliente que basicamente tiene los dato miembro de la clase , su respectivo constructor y destructor.Estamos utilizando funciones set y get para el cliente.

1. Funciones Set: Acceden a los datos miembros de la clase.
2. Funciones Get: obtienes los valores de los datos miembtro de la clase.

Los destructores virtuales son útiles cuando puede eliminar una instancia de una clase derivada a través de un puntero a la clase base:

```
class FacadeAdmin{
    public:
    static void CrearCliente(Cliente * cliente){
        AdminCliente::Agregar(cliente->insertQuery());
    }
    static QList<Cliente*> BuscarCliente(int clave){
        QList<Cliente*> lista=AdminCliente::BuscarPorClave
        return lista;
    }
    static QList<Cliente*> BuscarTodosCliente(){
        QList<Cliente*> lista=AdminCliente::BuscarTodos();
        return lista;
    }
    static void ModificarCliente(Cliente * cliente){
        AdminCliente::Modificar(cliente->updateQuery());
    }
    static void EliminarCliente(int clave){
        AdminCliente::Eliminar(clave);
    }

    static void CrearProducto(Producto * producto){
        AdminProducto::Agregar(producto->insertQuery());
    }
```

Bueno el nombre de esta clase es FacadeAdmin porque estamos utilizando el patron de diseño Facade ,el cual sse encarga de brindar una fachada a la clase. Nuestro Admin solo podra tener los privilegios de acceder a :

1. Crear
2. Buscar
3. Buscar
4. Modificar
5. Eliminar

ya sea para clientes ,productos y venta de productos.

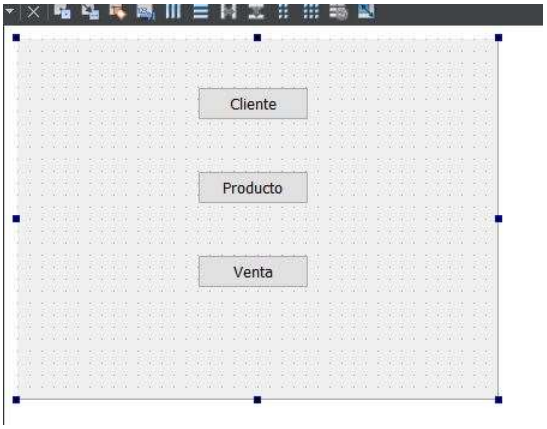
```
class FacadeCitas{
    public:
    static QList<Cliente*> BuscarCliente(int cla
        QList<Cliente*> lista=AdminCliente::Busc
        return lista;
    }
    static void AgregarCitas(QString qcita){
        AdminCitas::Agregar(qcita);
    }

    static QList<Cita*> ObtenerLista(){
        QList<Cita*> data=AdminCitas::BuscarTodo
        return data;
    }

};
```

Volvemos a encontrar un Facade , esta ves lo usaremos para citas...pero quizas se preguntaran porque..entonces en esta parte del codigo lo usamos porque nos permite simplificar el interface de comunicación entre dos objetos , en este caso : Clientes y Citas.

| CODIGO PARA LOS FORMULARIOS | EXPLICACION |
|---|---|

```
#ifndef FORMADMIN_H
#define FORMADMIN_H

#include <QWidget>

namespace Ui {
class FormAdmin;
}

class FormAdmin : public QWidget
{
    Q_OBJECT

public:
    explicit FormAdmin(QWidget *parent =
    ~FormAdmin();

private slots:
    void on_BCliente_clicked();

    void on_BProducto_clicked();

    void on_BVenta_clicked();

private:
    Ui::FormAdmin *ui;
};

#endif // FORMADMIN_H
```

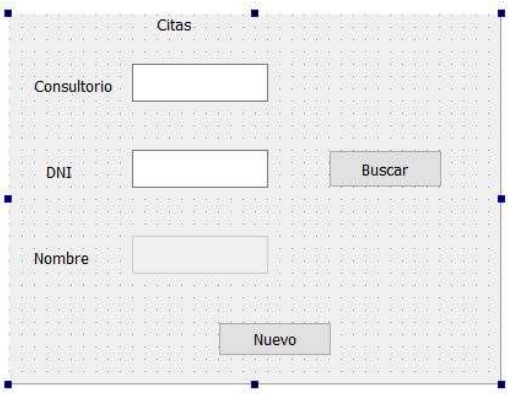
En esta imagen observamos el codigo para uno de nuestros formularios.Este formulario se ve de esta manera :




Es importante aclarar que Qt genera un codigo unico cuando nosotros creamos un boton o un cuadro de texto , simplemente es hermoso.
Los private slots son basicamente donde estamos haciendo que nuestros botones Cleinte,Producto y Venta funcionen..OJO BCliente, BProducto y BVenta son los nombres de nuestros botones para reconocerlos mas rapidamente.

```
#ifndef FORMCITASNUEVO_H
#define FORMCITASNUEVO_H

#include <QWidget>

namespace Ui {
class FormCitasNuevo;
}

class FormCitasNuevo : public QWidget
{
    Q_OBJECT

public:
    explicit FormCitasNuevo(QWidget *parent =
    ~FormCitasNuevo();

private slots:
    void on_BNuevo_clicked();

    void on_BBuscar_clicked();

private:
    Ui::FormCitasNuevo *ui;
};

#endif // FORMCITASNUEVO_H
```

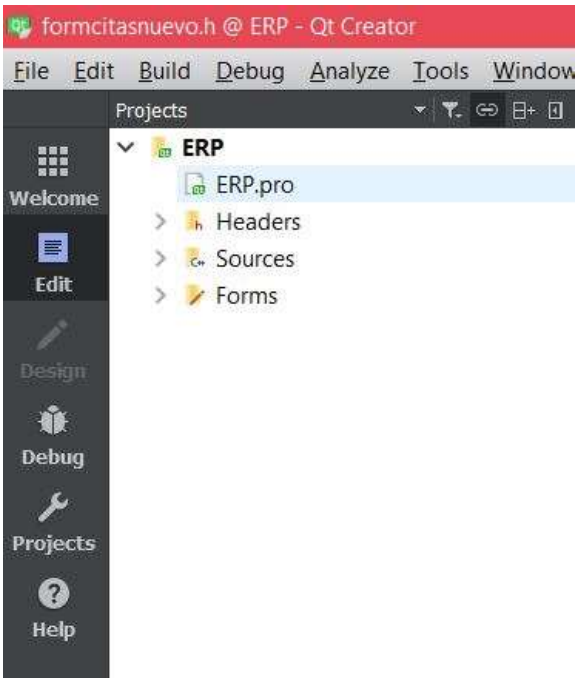
Estamos observando el codigo del formulario para citas… sin la programacion se ve asi :




```
#ifndef FORMLOGIN_H
#define FORMLOGIN_H

#include <QWidget>

namespace Ui {
class FormLogin;
}

class FormLogin : public QWidget
{
    Q_OBJECT

public:
    explicit FormLogin(QWidget *parent = 0);
    ~FormLogin();

private slots:
    void on_loggin_b_clicked();

private:
    Ui::FormLogin *ui;
};

#endif // FORMLOGIN_H
```

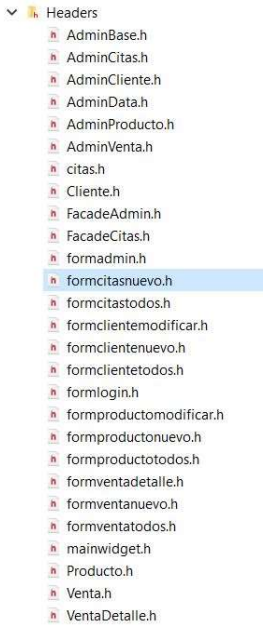

```
void FormLogin::on_loggin_b_clicked()
{
                                        QString
username=ui->username_txt->toPlainText();
    QString clave=ui->clave_txt->text();
    if(username=="admin" && clave=="1234")
    {
       FormAdmin * open=new FormAdmin();
       open->show();
       this->close();
    }
          else   if(username=="Selene"   &&
clave=="2018")
    {
       FormAdmin * open=new FormAdmin();
       open->show();
       this->close();
    }
    if(username=="Luz" && clave=="qtrct")
    {
       FormAdmin * open=new FormAdmin();
       open->show();
       this->close();
    }

}
```

En la siguiente imagen se muestra como esta ordenado nuestro proyecto:




ERP.pro se genera por default al generar un nuevo preoyecto en Qt Creator,junto un archivo.cpp y un archivo.ui.

En los headers tenemos:




Todos estos archivos contienen nuestras clases .

La carpeta sources contiene los archivos .cpp:


Y finalmente esta es la carpeta que contiene todos nuestros formularios (INTERFAZ), archivos .ui :