# UD 04 - DESARROLLO DE COMPONENTES

## 1. Técnicas y estándares. Modelo-Vista-Controlador

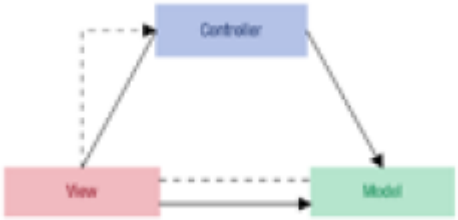
### Definición de MVC

El patrón Modelo-Vista-Controlador (MVC) divide la aplicación en tres componentes para conseguir que estén desacoplados:

- **Modelo**: Datos de la aplicación (Tablas)
- **Vista**: Interfaz de usuario (XML)
- **Controlador**: Forma en la que la interfaz reacciona a la entrada del usuario (Código)
- El controlador tiene acceso completo a la vista y al modelo
- El modelo tiene acceso limitado a la vista: El modelo envía notificación de que sus datos han sido modificados para que la vista actualice el contenido. El modelo no necesita conocer la vista, pero esta sí que necesita acceder al modelo.
- La vista tiene acceso limitado al controlador: Las dependencias que la vista tiene sobre el controlador deben ser mínimas para que este pueda ser sustituido en cualquier momento.

En el MVC de Oddo:

- Modelo: Tablas de la base de datos
- Vista: Archivos XML que definen la interfaz de usuario del módulo (vistas formulario, vistas tipo árbol...)
- Controlador: Código que define el comportamiento de la aplicación. Objetos creados en Python




### Especificaciones técnicas y funcionales

**Técnicas**

- La arquitectura cliente-servidor se permite al cliente hacer llamadas a procedimientos remotos para comunicación (ejecutar código en otra máquina). Esto se hace mediante los protocolos XML-RPC (llamada, argumentos y resultados enviados por HTTP y codificadas con XML) o Net-RCP (más reciente, basado en funciones en Python)
- Odoo funciona sobre un framework OpenObject que permite desarrollo rápido de aplicaciones (RAD) conteniendo:
  - ORM: Mapeo objeto-relacional a objetos Python
  - Arquitectura MVC
  - Diseñador de informes
  - Herramientas de Bussiness Intelligence y cubos multidimensionales
  - Cliente de escritori oy web

**Funcionales**

Odoo se compone de un núcleo y varios módulos. Estos componentes además de un nombre comercial tienen un nombre técnico.

- **base**: módulo básico con objetos como empresas (res.parner), direcciones de empresas (res.partnet.address), usuarios (res.user), monedas (res.current)...
- **account**: Gestión contable y financiera
- **product**: Productos y tarifas
- **purchase**: Gestión de compras
- **sale**: Gestión de ventas
- **mrp**: Fabricación y planificación de recursos
- **crm**: Gestión de las relaciones entre clientes y proveedores

En el directorio `addons` del servidor:

- Por cada módulo se corresponde una carpeta con el nombre del módulo en ese directorio
- Hay directorios con subcarpetas como report, wizard, test que contienen archivos del tipo de objeto utilizado (informes, asistentes y accesos directos, datos de prueba...)

Los archivos `_____.py` contienen valores dentro de un diccionario Python ({}):
name: nombre del módulo
version: versión del módulo
description: descripción de módulo
author: autor del módulo
website: sitio web del módulo
license: licencia del módulo (por defecto GPL )
depends: lista de módulos de los que depende el módulo (el módulo base es el más usado porque se definen datos para vistas, informes...)
init_xml: archivos XML que se cargan con la instalación del módulo
installable: determina si el módulo es instalable o no

# 2. Lenguaje de los sistemas ERP-CRM. Python en Odoo

Los ERP tienen lenguajes de programación modernos, muchos orientados a objetos. Pueden tener lenguajes propietarios como SAP ABAP o Apex.
En el caso de Odoo se utiliza Python (Guido van Rossum, años 80 en Países Bajos; existe la Python Software Foundation (PSF) para poseer propiedad intelectual sobre la licencia y promover el uso del lenguaje; Licencia de código abierto, compatible con GPL)

**Características de Python**

- **Sintaxis** sencilla
- **Lenguaje interpretado**: Se ejecutan usando un intérprete. El código fuente se traduce a archivo bytecode con extensión `.pyc` o `.pyo` que se ejecutará en sucesivas ocasiones
- **Tipado dinámico**: El tipo de los datos se determina en tiempo de ejecución
- **Fuertemente tipado**: No se puede usar la variable como si fuera de otro tipo, salvo conversión
- **Multiplataforma**
- **Orientado a objetos**

Tutorial en: https://tutorialpython.com/

## 2.1. Empezando con Python

Los archivos pueden ejecutarse con `python programa.py`

El primer programa:

```
print('Hola mundo')
```

Los comentario en el lenguaje se escriben con # para una línea o con """ si es de más de una línea

En este lenguaje no hay delimitadores de sentencia como los puntos y coma, ni delimitadores de bloques como las llaves.

## 2.2. Variables

No se declaran, se utilizan directamente.

- Números enteros (Se representan con el tipo int que usa 32 o 64 bits dependiendo del tipo de procesador o el tipo long que permite almacenar números de cualquier precisión, limitados por la memoria disponible en la máquina ya que a bajo nivel se implementa con C por debajo)
- Números reales (float)
- Números complejos
- Cadenas
- Booleanos
- Operadores aritméticos (...división entera //, exponencial **), booleanos (and, or, not), relacionales (== != < > <= >=)

```
# Define dos variables numero1 y numero2
numero1 = int(2)
numero2 = float(2.5)

"""
Y ahora me apetece jugar con printf
"""
# Concatenando strings
print("Querido Alcalde de Alicante " + "que así se llama")
# Imprimiendo numerito
print(numero1)
# Imprimiendo numeritos concatenados al string
print("Este es el numero1: (Parsealo a str o no compila)" + str(numero1))
print("Este es el numero2: (Parsealo a str o no compila) " + str(numero2))
# Con text blocks
print("""
    Oye pues es un poquito curioso esto de Python    Una pena que solo lo vaya a usar un
día""")
# Con text blocks formateados
print("""
Ojalá saber quien es {0} para saludarlo
""".format("M. Rajoy"))
# Con payloads
ciudad = "Málaga"
print(f"Vaya ciudad más bonita es {ciudad}, aunque muy llena de turistas está")
```

## 2.3. Colecciones

### A. Listas

Son mutables. Se definen con corchetes.

`lista[i]`: Devuelve el elemento
`lista.pop(i)`: Devuelve el elemento de i y lo borra
`lista.append(elemento)`: Añade elemento al final
`lista.insert(i, elemento)`: Inserta en posición i
`lista.extends(lista2)`: Fusiona lista con lista 2
`lista.remove(elemento)`: Elimina la primera vez que aparece el elemento
Pueden ser desempaquetadas (asigno la lista a múltiples variables en una sola instrucción)

```
# Declaramos una lista
myList=["Manzana", "Pera"]
fruta = myList[0]
## La puedo desempaquetar
a,b = myList
print(str(a))
print(str(b))
print(f"Mi fruta preferida es la {fruta}")
print("Pero si quiero la elimino de la lista {0} y chao".format(str(myList)))
myList.pop(0)
print("¿ves? ya no está:"+str(myList))
myList2=["Patata", "Lechuga", "Pimiento"]
```

```
## Elimino
del(myList2[1])
print("Pongamos un {0} y a juntarlo con la otra".format(myList2))
myList.extend(myList2)
print("¿ves? {0}".format(str(myList)))

#¿Y si fuese una tupla? (Inmutable)
myTupla=("Manzana", "Pera")
print(str(myTupla))
```

### B. Tuplas

Las tuplas son inmutables. Se definen con paréntesis.

```
myTupla=("Manzana", "Pera")
```

### C. Diccionarios

Diccionarios antes no tenían orden.
Ahora guardan el orden de inserción de los elementos.

`diccionario.get('key'):` Toma del diccionario esa clave
`diccionario.pop('key')` : Toma y elimina del diccionario esa clave
`diccionario.update({'key':'valor'})`: Actualiza valor de esa clave
`"key" in diccionario`: devuelve true o false según esté o no en el diccionario como key
`"definicion" in diccionario`: devuelve true o false según esté o no en el diccionario como valor
`del diccionario[`key`]` Elimina del diccionario
`diccionario["key"] = "value"`: Añade al diccionario

```
# Inicio un diccionario
diccionario = {'diasDescuento':'20','diasIntegro':'30','diasDevolucion':'40'}
print(diccionario)
# Cojo una de las claves
print(diccionario.get('diasDescuento'))
# Compruebo si una de las claves está
print('diasDescuento' in diccionario)
# O si uno de los valores está
print('40' in diccionario.values())
# Elimino un valor
del diccionario['diasDevolucion']
print(diccionario)
# Modifico un valor
diccionario.update({'diasDescuento':'25'})
# Añado un valor
diccionario["diasRebajas"] = "12"
print(diccionario)
```

### D. Conjuntos

No admiten duplicados

Se crean con llaves pero como si fuesen una lista

```
mySet = {'rojo', 'amarillo', 'verde'}
# Lo puedo usar para borrar de duplicados de una lista
lista = ['Fernando', 'Fernando', 'Felipe']
print(lista)
# ['Fernando', 'Fernando', 'Felipe']
lista = list(set(lista))
print(lista)
# ['Fernando', 'Felipe']
```

## 2.4. Sentencias del lenguaje

### A. Condicionales

Se hacen con `if`, `elif` y `else`
Y los operadores lógicos son `and`, `or`...

```
x = 10
if x > 0 and x % 2 == 0:
    print("x es un número positivo par")
elif x > 0 and x % 2 != 0:
    print("x es un número positivo impar")
elif x == 0:
    print("x es igual a cero")
else:
    print("x es un número negativo")
```

---

## Match

```
status = 418
match status:
    case 400:
        print("Bad request")
    case 404:
        print("Not found")
    case 418:
        print("I'm a teapot")
    case _:
        print("Something's wrong with the internet")
```

### B. Bucles

El `while` no tiene misterio (salvo que no hay `i++`)

El `for` es `for i in range(n)` . Aunque si se necesita se puede especificar el paso.

El foreach es `for elemento in mi_lista` o similar

```
contador = 0
print("contador while")
while contador < 5:
    print(contador)
    contador += 1

print("contador for in range")
for i in range(5):
    print(i)

print("contador for in range inverso")
for i in range(10, 0, -1):
    print(i)

print("contador foreach")
mi_lista = [1, 2, 3, 4, 5]
for elemento in mi_lista:
    print(elemento)

print("contador foreach diccionario")
# Bucle for con diccionario
mi_diccionario = {"a": 1, "b": 2, "c": 3}
for clave in mi_diccionario:
    print(clave, mi_diccionario[clave])
```

Sentencias `break`, `continue`, `else` hacen lo esperado..
Y, si el bucle for, termina sin encontrar un `break` podría usarse una clausula `else` para ejecutar algo

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

La sentencia `pass` no hace nada. Se puede usar cuando una sentencia es requerida por la sintaxis pero el programa no requiere ninguna acción.

```
while true
        pass

clas MyEmptyClass
        pass

def initlog(*args)
        pass #todo
```

## 2.5. Llamadas a funciones

Se inicia como
`def nombreFuncion:`

La identación es la que permite diferenciar qué está dentro y qué está fuera de la función.

Puede o no tener un `return` Cuando la función no tiene `return`, devuelve el valor "None"

```
#Declaramos funcion
def sumaNumeritos(a, b):
    print(a+b)

def restaNumeritos(a,b):
        return a - b

def cuadradoNumerito(a):
    return a**2
# Comprobamos que devuelve "None"
print(sumaNumeritos(1,2))
# Comprobamos que devuelve -1
print(restaNumeritos(1,2))
# Comprobamos cómo es la operación de la potencia
print(cuadradoNumerito(3))
```

## 2.6. Clases y objetos

Se definen con la palabra `class` seguida del nombre de la clase, dos puntos (:) y el cuerpo de la clase.

```
class Individuo:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def dimeEdad(self):
        if self.edad >= 18:
            print("Mayor de edad")
        else:
            print("Menor de edad")

# Crear una instancia de Individuo
individuo = Individuo("Jesús", 23)
individuo.dimeEdad()  # Salida: "Mayor de edad"
print(individuo.edad) # Salida: Jesús
```

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        if self.nombre is not None:
            print(f"Hola, mi nombre es {self.nombre}.")
        else:
            print("Hola, soy una persona sin nombre.")

# Crear una instancia de Persona sin proporcionar nombre
persona = Persona()
persona.saludar()  # Salida: "Hola, soy una persona sin nombre."

# Establecer el nombre de la persona después de la creación
persona.nombre = "Juan"
persona.saludar()  # Salida: "Hola, mi nombre es Juan."
```

## 2.7. Módulos y paquetes

Un módulo es un archivo que contiene **declaraciones y definiciones de Python**. (O sea, un archivo de Python `.py`) (Archivos)
Un paquete es una colección de módulos relacionados. (Directorios)
Esto ayuda a:

- Hacer el código más mantenible
- Reutilizar código

Se usa con `import nombre_modulo`
O si quisieramos importar algo concreto `from nombre_modulo import Persona`

funciones.py

```
def comenzamosModulos():
    print("Comenzamos los modulos")
```

miCodigo.py

```
import funciones

funciones.comenzamosModulos()
```

---

Para que Python trate a un directorio como paquete debe crearse un archivo `__init__.py` en dicha carpeta. Para hacer que un módulo pertenezca a un paquete hay que copiar el archivo `nombre_modulo.py` en el directorio del paquete.
Los paquete se importan también con la sentencia `import`

Muchos pragramadores elaboran módulos y los comparten.

Para instalar el módulo simplemente hay que copiar el archivo .py en la carpeta en la que se tenga el código fuente. Sin embargo, puede instalarse de forma más automatizada.

## 2.8 Manejo de excepciones

Las excepciones en Python se manejan con `try-except`

```
try:
    def dividir(a, b):
        return a / b
    dividir(1,0)
```

```
except:
    print ("Ha ocurrido un error")

try:
    numerito = 2
    print("Este es el numero1: (Parsealo a str o no funciona)" + numerito)
except:
    print ("Ha ocurrido un error")
```

## 2.9. Librerías de funciones (APIs)

Python proporciona una API estándar cuyo código fuente está abierto y libre.
Destacan:

- `os`: Funciones del sistema operativo como creación de archivos y directorios, leer o escribir un fichero, manipular rutas
- `sys`: Acceder a información sobre el intérprete de Python como el prompt del sistema, datos de licencia y cualquier parámetro o variable que tenga que ver con el intérprete
- `datetime`: Módulos para la manipulación de fechas y horas
- `math`: Funciones matemáticas
- `locale`: configuración regional del sistema (moneda, formato de fecha y hora, números, codificación)
- `MySQLdb`: interfaz para acceder a MySQL desde Python

## 2.10. Inserción, modificación y eliminación de datos en objetos (Módulos Odoo)

Primero debe situarse la carpeta `addons` de Oddo en el sistema operativo. (En nuestra instalación de Odoo el archivo `addons.conf` la opción `addons_path`, es la ruta donde por defecto se instalan los módulos)

Es interesante usar una ruta diferente a los módulos propios que se desarrollen para no perderlos (editando el archivo `odoo.conf` y mediento una nueva ruta separada por "," de la otra.

Para crear el módulo se debe en la carpeta del módulo

- **Crear estructura del módulo** colgando de la carpeta `nombre_modulo` y teniendo cuidado con los permisos. Puede usarse el comando
  `odoo-bin scaffold`
  o bien
  `python odoo-bin scaffold`

  También puede descargarse la estructura de un módulo de la documentación oficial de Odoo
- **Configurar los datos del módulo** (`__manifest.py__`) y meterlo en el servidor. Activar modo desarrollador y actualizar las aplicaciones.
- **Creación del modelo, meterlo en el servidor y reiniciar el servicio de Odoo** (es necesario cada vez que se modifique un archivo Python)
- **Creación de las vistas** ( archivo.xml), **editar archivo** `views.xml` e introducirlo en el servidor. No es necesario reiniciar el servicio de Odoo pero sí actualizar el módulo o desinstalar y volver a instalar.
- Comprobar el módulo. Meter registros, consultar, modificar, eliminar, realizar búsquedas...

En la estructura del módulo de Odoo se encuentran los archivos:

- `__init__.py`: Donde se inicia el módulo. Se indica qué modelo quiere vincularse al módulo. El modelo es la base de datos. Es necesario para que la carpeta se trate como paquete de Python. Contiene los import de cada archivo del módulo.
- `__terp.py__`: Contiene un diccionario Python con la descripción del módulo, autor, version, módulos de los que depende
- `nombre_modulo.py`: Clase definiendo campos que tendrá. Al crearla se crea el modelo (tabla en BBDD) y el controlador porque se define el comportamiento que tendrá.

- `nombre_modulo_view.xml`: Vista del módulo o del objeto que creará el módulo. Deben tenerse conocimientos de XML o coger vista de otro objeto y a partir de ella crear la del nuevo objeto.

# 3. Entornos de desarrollo y herramientas de desarrollo en sistemas ERP-CRM

Para programar en Python se podría usar como IDE sencillo: Idle. (Otros IDEs Visual Studio Code, Sublime Text) Allí puede abrirse el código con Python, revisarse la sintaxis (Run/Check) o ejecutar el programa (Run/Run module) entre otras opciones.
Puede accederse a extensiones o plugins de Odoo desde diferentes entornos.

## Depuración de un programa

Python incorpora un depurador dentro de su biblioteca de módulos llamado `pdb`
Tiene puntos de ruptura, ejecución paso a paso de código, inspeccionar valores de variables y otras opciones de depuración.

Aunque es mejor hacerlo desde entornos gráficos como desde el IDE IDLE en su módulo de depuración.

# 4. Formularios e informes en sistemas ERP-CRM

- **Formularios**: Interfaz de un módulo. Una vez creado el objeto del módulo `nombre_modulo.py` deben crearse los menús, acciones, vistas (formulario u otro tipo) para interactuar con el objeto. Estos elementos se describen en `nombre_modulo_view.xml`
- **Informes**: Pueden ser estadísticos (listados de datos y gráficos por pantalla) o documentos cuya finalidad es ser impresos.

### Arquitectura de formularios e informes. Elementos

Se construyen de forma dinámica por la descripción XML de la pantalla del cliente.

La estructura `nombre_modulo_view.xml` es similar a esta:

```
<?xml version="1.0" encoding="utf-8"?>
<odoo>
        <data>
                <record model="object_model_name" id="object_xml_id">
                        <field name="field1">value1</field>
                        <field name="field2">value2</field>
                </record>
                <record model="object_model_name2" id="object_xml_id2">
                        <field name="field1" ref="module.object_xml_id"/>
                        <field name="field2" eval="ref('module.object_xml_id')"/>
                </record>
        </data>
</odoo>
```

(Ahora Odoo, antes OpenERP)

Los registros se definen con la etiqueta `record` que indica el inicio y el fin de la descripción del registro. Cada tipo de registro hace referencia a un objeto diferente.
Con el atributo model puede definirse si es una acción o una vista.

Dentro del record van los field que definen las características del registro.

Un ejemplo de creación de vista de tipo formulario es:

```
<record model="ir.ui.view" id="view_agenda_form">
                <field name="name">agenda</field>
                <field name="model">agenda</field>
                <field name="type">form</field>
                <field name="priority" eval="5"/>
```

```
                <field name="arch" type="xml">
                <form string="Agenda">
                        <field name="nombre" select="1"/>
                        <field name="telefono" select="1" />
                </form>
        </field>
</record>
```

El `field` `name="name"` hace referencia al nombre de la vista. El `field` `name="model"` al objeto del modelo, es decir, la tabla en la base de datos del cual muestra la información.
La vista a su vez se forma por dos campos "nombre" y "telefono"

Puede definirse un elemento del menú para acceder al objeto agenda

```
<menuitem name="Agenda" id="menu_agenda_agenda_form"/>
<menuitem name="Agenda" id="menu_agenda_form" parent="agenda.menu_agenda_agenda_form"
action="action_agenda_form"/>
```

Y la acción asociada a ese elemento de menú

```
<record model="ir.actions.act_window" id="action_agenda_form">
        <field name="res_model">agenda</field>
        <field name="domain">[]</field>
</record>
```

En este ejemplo, definimos dos vistas: una vista de formulario para el modelo `product.product` y una vista de árbol para el mismo modelo. En la vista de formulario, definimos un formulario con tres campos: `name`, `description` y `list_price`, agrupados dentro de un grupo. En la vista de árbol, definimos un árbol con tres columnas: `name`, `default_code` y `type`.

Ten en cuenta que estas definiciones de vista XML se proporcionan dentro de archivos `.xml` en los módulos de Odoo. Luego, estos archivos XML son cargados por el sistema de Odoo para construir la interfaz de usuario de la aplicación.

```
<odoo>
    <data>
        <!-- Ejemplo de una vista formulario -->
        <record id="view_form_producto" model="ir.ui.view">
            <field name="name">Formulario de Producto</field>
            <field name="model">product.product</field>
            <field name="arch" type="xml">
                <form string="Producto">
                    <sheet>
                        <group>
                            <field name="name"/>
                            <field name="description"/>
                            <field name="list_price"/>
                        </group>
                    </sheet>
                </form>
            </field>
        </record>

        <!-- Ejemplo de una vista árbol -->
        <record id="view_tree_producto" model="ir.ui.view">
            <field name="name">Árbol de Producto</field>
            <field name="model">product.product</field>
            <field name="arch" type="xml">
                <tree string="Productos">
                    <field name="name"/>
                    <field name="default_code"/>
                    <field name="type"/>
                </tree>
            </field>
        </record>
    </data>
</odoo>
```

## Herramientas para creación de formularios e informes

Se usará básicamente la creación de archivos XML con el nombre de `nombre_modulo_view.xml`
Otra forma para los informes es utilizando Jasper Reports (librería generación informes). Consiste en generar un XML con la descripción del informe a crear, que es tratado para generar un documento en formato de salida (PDF, HTML, XLS...)