__init__.py

```
# app/__init__.py
from .config import Config
from . import models
from . import views
from . import controllers
from . import utils

__version__ = '1.0.0'
__author__ = 'Tu Nombre'
```

config.py

```
# app/config.py
import os
from pathlib import Path
from datetime import datetime

class Config:
    # Información de la aplicación
    APP_NAME = 'Sistema de Gestión de CVs'
    VERSION = '1.0.0'
    DEBUG = True

    # Rutas base
    BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    APP_DIR = os.path.join(BASE_DIR, 'app')

    # Directorios de recursos
    RESOURCES_DIR = os.path.join(APP_DIR, 'resources')
    TEMPLATES_DIR = os.path.join(RESOURCES_DIR, 'templates')
    IMAGES_DIR = os.path.join(RESOURCES_DIR, 'images')
    STYLES_DIR = os.path.join(RESOURCES_DIR, 'styles')

    # Directorios de datos
    DATA_DIR = os.path.join(BASE_DIR, 'data')
    OUTPUT_DIR = os.path.join(DATA_DIR, 'output')
    TEMP_DIR = os.path.join(DATA_DIR, 'temp')

    # Rutas de archivos
    DB_PATH = os.path.join(DATA_DIR, 'cv_database.db')
    CV_TEMPLATE_PATH = os.path.join(TEMPLATES_DIR, 'cv_template.docx')
    LOGO_PATH = os.path.join(IMAGES_DIR, 'logo.png')

    # Credenciales de demo
    DEMO_USER = 'admin'
    DEMO_PASSWORD = 'admin123'

    # Configuraciones de la aplicación
    AREAS = [
        'Administración',
        'Oficina Técnica',
        'Construcción',
        'Calidad',
        'SSOMA'
    ]

    TIPOS_OBRA = [
        'Planta de Cemento y Cal',
        'Naves Industriales',
        'Hidroeléctrica',
        'Central Termoeléctrica',
        'Planta Eléctrica',
        'Puentes',
        'Planta Petróleo',
        'Obras Civiles',
        'Planta de Gas',
        'Construcción Estructuras',
        'Montaje de Estructura Pesada',
```

```
            'Chancadora/Fajas (seca)',
            'Molinos/Celda de Flotación',
            'Túneles',
            'Desmontaje',
            'Electricidad & Instrumentación',
            'Tubería Agua/Relaves',
            'Planta Desalinizadora',
            'Truckshop - Almacenes (Auxiliares)'
        ]

        @classmethod
        def create_directories(cls):
            """Crea todos los directorios necesarios para la aplicación"""
            directories = [
                cls.DATA_DIR,
                cls.OUTPUT_DIR,
                cls.TEMP_DIR,
                cls.RESOURCES_DIR,
                cls.TEMPLATES_DIR,
                cls.IMAGES_DIR,
                cls.STYLES_DIR
            ]

            for directory in directories:
                os.makedirs(directory, exist_ok=True)

        @classmethod
        def get_output_path(cls, filename: str) -> str:
            """Genera una ruta única para archivos de salida"""
            timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
            return os.path.join(cls.OUTPUT_DIR, f'{filename}_{timestamp}.docx')

        @classmethod
        def clean_temp_files(cls):
            """Limpia archivos temporales antiguos"""
            if os.path.exists(cls.TEMP_DIR):
                for file in os.listdir(cls.TEMP_DIR):
                    file_path = os.path.join(cls.TEMP_DIR, file)
                    try:
                        if os.path.isfile(file_path):
                            os.unlink(file_path)
                    except Exception as e:
                        print(f'Error al eliminar {file_path}: {e}')
```

constants.py

```python
class TipoDocumentoIdentidad:
    DNI = 'DNI'  # Documento Nacional de Identidad
    CE = 'CE'   # Carnet de Extranjería
    PAS = 'PAS' # Pasaporte
    RUC = 'RUC' # Registro Único de Contribuyentes
    CPP = 'CPP' # Carnet de Permiso Temporal de Permanencia

    @classmethod
    def choices(cls):
        return [
            (cls.DNI, 'DNI'),
            (cls.CE, 'Carnet de Extranjería'),
            (cls.PAS, 'Pasaporte'),
            (cls.RUC, 'RUC'),
            (cls.CPP, 'Carnet PTP')
        ]

class TipoObra:
    # Industria
    PLANTA_CEMENTO = 'Planta de Cemento y Cal'
    NAVES_INDUSTRIALES = 'Naves Industriales'
    HIDROELECTRICA = 'Hidroeléctrica'
    CENTRAL_TERMO = 'Central Termoeléctrica'
    PLANTA_ELECTRICA = 'Planta Eléctrica'
    PUENTES = 'Puentes'
    PLANTA_PETROLEO = 'Planta Petróleo'
    OBRAS_CIVILES = 'Obras Civiles'

    # Minería
    PLANTA_GAS = 'Planta de Gas'
    CONST_ESTRUCTURAS = 'Construcción Estructuras'
    MONTAJE_ESTRUCTURAS = 'Montaje de Estructura Pesada'
    CHANCADORA_FAJAS = 'Chancadora/Fajas (seca)'
    MOLINOS_FLOTACION = 'Molinos/Celda de Flotación'
    TUNELES = 'Túneles'
    DESMONTAJE = 'Desmontaje'

    # Servicios
    ELECTRICIDAD_INSTR = 'Electricidad & Instrumentación'
    TUBERIA_AGUA = 'Tubería Agua/Relaves'
    PLANTA_DESALINIZADORA = 'Planta Desalinizadora'
    TRUCKSHOP = 'Truckshop - Almacenes (Auxiliares)'

    @classmethod
    def get_all_tipos(cls):
        return [
            (cls.PLANTA_CEMENTO, 'Industria'),
            (cls.NAVES_INDUSTRIALES, 'Industria'),
            (cls.HIDROELECTRICA, 'Industria'),
            (cls.CENTRAL_TERMO, 'Industria'),
            (cls.PLANTA_ELECTRICA, 'Industria'),
            (cls.PUENTES, 'Industria'),
            (cls.PLANTA_PETROLEO, 'Industria'),
            (cls.OBRAS_CIVILES, 'Industria'),
            (cls.PLANTA_GAS, 'Minería'),
            (cls.CONST_ESTRUCTURAS, 'Minería'),
```

```
            (cls.MONTAJE_ESTRUCTURAS, 'Minería'),
            (cls.CHANCADORA_FAJAS, 'Minería'),
            (cls.MOLINOS_FLOTACION, 'Minería'),
            (cls.TUNELES, 'Minería'),
            (cls.DESMONTAJE, 'Minería'),
            (cls.ELECTRICIDAD_INSTR, 'Servicios'),
            (cls.TUBERIA_AGUA, 'Servicios'),
            (cls.PLANTA_DESALINIZADORA, 'Servicios'),
            (cls.TRUCKSHOP, 'Servicios')
        ]
```

exceptions.py

```python
# app/exceptions.py
class CVSystemException(Exception):
    """Excepción base para el sistema"""
    pass

class DatabaseError(CVSystemException):
    """Errores relacionados con la base de datos"""
    pass

class ValidationError(CVSystemException):
    """Errores de validación de datos"""
    pass

class TemplateError(CVSystemException):
    """Errores relacionados con las plantillas"""
    pass
```

logger.py

```
# app/logger.py
import logging
import os
from datetime import datetime
from .config import Config

def setup_logger():
    # Crear directorio de logs si no existe
    log_dir = os.path.join(Config.DATA_DIR, 'logs')
    os.makedirs(log_dir, exist_ok=True)

    # Configurar logger
    log_file = os.path.join(
        log_dir,
        f'cv_system_{datetime.now().strftime("%Y%m%d")}.log'
    )

    logging.basicConfig(
        level=logging.DEBUG if Config.DEBUG else logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler(log_file),
            logging.StreamHandler()
        ]
    )

    return logging.getLogger('cv_system')

logger = setup_logger()
```

main.py

```python
# app/main.py
import sys
import os
import logging
from PyQt5.QtWidgets import QApplication, QMessageBox
from PyQt5.QtCore import Qt
from app.controllers.login_controller import LoginController
from app.controllers.main_controller import MainController
from app.config import Config
from app.utils.setup_check import verificar_plantilla_cv


def setup_logging():
    logging.basicConfig(
        level=logging.DEBUG,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.StreamHandler(),
            logging.FileHandler('app.log')
        ]
    )

class Application:
    def __init__(self):
        setup_logging()
        self.logger = logging.getLogger(__name__)
        self.logger.debug("Iniciando aplicación...")

        # Crear las carpetas necesarias
        Config.create_directories()

        # Inicializar la aplicación
        self.app = QApplication(sys.argv)
        self.app.setStyle('Fusion')

        # Cargar estilos
        style_path = os.path.join(Config.STYLES_DIR, 'style.qss')
        if os.path.exists(style_path):
            with open(style_path, 'r') as f:
                self.app.setStyleSheet(f.read())

        # Mantener referencias a los controladores
        self.login_controller = None
        self.main_controller = None

    def start(self):
        try:
            # Iniciar con ventana de login
            self.login_controller = LoginController()

            # Auto-login en modo debug
            if Config.DEBUG:
                self.login_controller.view.usuario_input.setText(Config.DEMO_USER)
                self.login_controller.view.password_input.setText(Config.DEMO_PASSWORD)

            self.login_controller.view.show()
```

```
57              self.login_controller.view.loginSuccess.connect(self.on_login_success)
58
59              return self.app.exec_()
60
61          except Exception as e:
62              self.logger.error(f"Error en la aplicación: {str(e)}", exc_info=True)
63              QMessageBox.critical(None, "Error Fatal", f"Error en la aplicación: {str(e)}")
64              return 1
65
66      def on_login_success(self):
67          try:
68              self.logger.debug("Login exitoso, creando ventana principal...")
69              self.main_controller = MainController()
70              self.main_controller.view.show()
71              self.login_controller.view.close()
72          except Exception as e:
73              self.logger.error(f"Error al crear ventana principal: {str(e)}", exc_info=True)
74              QMessageBox.critical(None, "Error Fatal", f"Error al iniciar la aplicación: {str(e)}
75
76  def main():
77      if not verificar_plantilla_cv():
78          print("La aplicación continuará, pero la generación de CVs no estará disponible.")
79      # Configurar logging
80      setup_logging()
81      logger = logging.getLogger(__name__)
82
83      application = Application()
84      return application.start()
85
86  if __name__ == '__main__':
87      sys.exit(main())
```

controllers__init__.py

```
1   # app/controllers/__init__.py
2   from .login_controller import LoginController
3   from .main_controller import MainController
4
5   __all__ = ['LoginController', 'MainController']
```

controllers\login_controller.py

```python
# app/controllers/login_controller.py
from PyQt5.QtWidgets import QMessageBox
from app.views.login_view import LoginView
from app.config import Config
from app.models.database import Database

class LoginController:
    def __init__(self):
        self.view = LoginView()
        self.db = Database()
        self.setup_connections()

    def setup_connections(self):
        self.view.login_btn.clicked.connect(self.validar_login)
        # Añadir eventos Enter para una mejor experiencia de usuario
        self.view.usuario_input.returnPressed.connect(self.validar_login)
        self.view.password_input.returnPressed.connect(self.validar_login)

    def validar_login(self):
        try:
            usuario = self.view.usuario_input.text().strip()
            password = self.view.password_input.text().strip()

            if not usuario or not password:
                QMessageBox.warning(
                    self.view,
                    "Error",
                    "Por favor complete todos los campos"
                )
                return

            # En modo debug usar credenciales de prueba
            if Config.DEBUG:
                if usuario == Config.DEMO_USER and password == Config.DEMO_PASSWORD:
                    self.view.loginSuccess.emit()
                else:
                    QMessageBox.warning(
                        self.view,
                        "Error",
                        "Credenciales incorrectas"
                    )
            else:
                # TODO: Implementar validación con base de datos real
                pass

        except Exception as e:
            QMessageBox.critical(
                self.view,
                "Error",
                f"Error en el inicio de sesión: {str(e)}"
            )
```

controllers\main_controller.py

```
# controllers/main_controller.py
import logging
from PyQt5.QtWidgets import QMessageBox, QTableWidgetItem, QFileDialog
from PyQt5.QtCore import Qt
from datetime import datetime
from app.views.main_view import MainView
from app.models.database import Database
from app.utils.cv_generator import CVGenerator

logger = logging.getLogger(__name__)

class MainController:
    def __init__(self):
        try:
            logger.debug("Inicializando MainController...")
            self.view = MainView()
            self.db = Database()
            self.experiencias_widgets = []
            self.setup_connections()
            self.cargar_datos_iniciales()
            logger.debug("MainController inicializado correctamente")
        except Exception as e:
            logger.error(f"Error al inicializar MainController: {str(e)}")
            raise

    def setup_connections(self):
        try:
            logger.debug("Configurando conexiones...")

            # Conexiones del tab de registro
            self.view.tab_registro.btn_guardar.clicked.connect(self.guardar_registro)
            self.view.tab_registro.btn_cancelar.clicked.connect(self.cancelar_registro)
            self.view.tab_registro.btn_agregar_experiencia.clicked.connect(self.agregar_experien

            # Conexiones del tab de búsqueda
            self.view.tab_busqueda.btn_buscar.clicked.connect(self.realizar_busqueda)
            self.view.tab_busqueda.btn_limpiar.clicked.connect(self.limpiar_filtros)

            logger.debug("Conexiones configuradas correctamente")
        except Exception as e:
            logger.error(f"Error al configurar conexiones: {str(e)}")
            raise

    def cargar_datos_iniciales(self):
        try:
            self.actualizar_tabla_resultados()
        except Exception as e:
            logger.error(f"Error al cargar datos iniciales: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al cargar datos iniciales: {str(e)}

    def realizar_busqueda(self):
        try:
            filtros = {
                'texto': self.view.tab_busqueda.busqueda_input.text().strip(),
                'area': self.view.tab_busqueda.area_filtro.currentText(),
                'tipo_obra': self.view.tab_busqueda.tipo_obra_filtro.currentText(),
```

```
                'cargo': self.view.tab_busqueda.cargo_filtro.text().strip(),
                'ex_eimsa': self.view.tab_busqueda.check_eimsa_filtro.isChecked(),
                'años_exp': self.view.tab_busqueda.años_exp_filtro.currentText()
            }
            resultados = self.db.buscar_registros(filtros)
            self.actualizar_tabla_resultados(resultados)
        except Exception as e:
            logger.error(f"Error en búsqueda: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al realizar la búsqueda: {str(e)}")

    def actualizar_tabla_resultados(self, resultados=None):
        try:
            if resultados is None:
                resultados = self.db.obtener_todos_registros()

            tabla = self.view.tab_busqueda.tabla_resultados
            tabla.setRowCount(0)

            for row, datos in enumerate(resultados):
                tabla.insertRow(row)

                # Convertir Row a diccionario si es necesario
                if not isinstance(datos, dict):
                    datos = dict(datos)

                items = [
                    datos.get('codigo', ''),
                    f"{datos.get('tipo_documento', '')} {datos.get('numero_documento', '')}",
                    f"{datos.get('nombres', '')} {datos.get('apellidos', '')}",
                    datos.get('profesion', ''),
                    datos.get('area', ''),
                    datos.get('cargo', ''),
                    datos.get('años_experiencia', '0'),
                    datos.get('ciudad_residencia', ''),
                    'Sí' if datos.get('trabajado_eimsa') else 'No',
                    datos.get('fecha_registro', '')
                ]

                for col, valor in enumerate(items):
                    item = QTableWidgetItem(str(valor))
                    item.setFlags(item.flags() & ~Qt.ItemIsEditable)
                    tabla.setItem(row, col, item)

            tabla.resizeColumnsToContents()

        except Exception as e:
            logger.error(f"Error al actualizar tabla: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al actualizar tabla: {str(e)}")

    def limpiar_filtros(self):
        try:
            tab_busqueda = self.view.tab_busqueda
            tab_busqueda.busqueda_input.clear()
            tab_busqueda.area_filtro.setCurrentIndex(0)
            tab_busqueda.tipo_obra_filtro.setCurrentIndex(0)
            tab_busqueda.cargo_filtro.clear()
            tab_busqueda.check_eimsa_filtro.setChecked(False)
            tab_busqueda.años_exp_filtro.setCurrentIndex(0)
```

```
            self.actualizar_tabla_resultados()
        except Exception as e:
            logger.error(f"Error al limpiar filtros: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al limpiar filtros: {str(e)}")

    def ver_detalle(self):
        try:
            registro_seleccionado = self.obtener_registro_seleccionado()
            if registro_seleccionado:
                # Cambiar a la pestaña de registro en modo lectura
                self.view.tab_widget.setCurrentWidget(self.view.tab_registro)
                self.cargar_datos_registro(registro_seleccionado, modo_lectura=True)
        except Exception as e:
            logger.error(f"Error al ver detalle: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al ver detalle: {str(e)}")

    def editar_registro(self):
        try:
            registro_seleccionado = self.obtener_registro_seleccionado()
            if registro_seleccionado:
                # Cambiar a la pestaña de registro en modo edición
                self.view.tab_widget.setCurrentWidget(self.view.tab_registro)
                self.cargar_datos_registro(registro_seleccionado, modo_lectura=False)
        except Exception as e:
            logger.error(f"Error al editar registro: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al editar registro: {str(e)}")

    def obtener_registro_seleccionado(self):
        tabla = self.view.tab_busqueda.tabla_resultados
        indices_seleccionados = tabla.selectedItems()
        if indices_seleccionados:
            fila = indices_seleccionados[0].row()
            codigo = tabla.item(fila, 0).text()
            return self.db.obtener_registro(codigo)
        return None

    def guardar_registro(self):
        try:
            datos = self.obtener_datos_formulario()
            if self.validar_datos(datos):
                self.db.guardar_registro(datos)
                QMessageBox.information(self.view, "Éxito", "Registro guardado correctamente")
                self.view.tab_widget.setCurrentWidget(self.view.tab_busqueda)
                self.actualizar_tabla_resultados()
        except Exception as e:
            logger.error(f"Error al guardar registro: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al guardar registro: {str(e)}")

    def cancelar_registro(self):
        self.view.tab_widget.setCurrentWidget(self.view.tab_busqueda)

    def generar_cv(self):
        try:
            registro_seleccionado = self.obtener_registro_seleccionado()
            if registro_seleccionado:
                # Generar nombre del archivo
                nombre_archivo = f"CV_{registro_seleccionado.get('nombres', 'sin_nombre')}_{date
                ruta_salida, _ = QFileDialog.getSaveFileName(
```

```
                    self.view,
                    "Guardar CV",
                    nombre_archivo,
                    "Documentos Word (*.docx)"
                )

                if ruta_salida:
                    try:
                        generator = CVGenerator()
                        experiencias = self.db.obtener_experiencias(registro_seleccionado['codig
                        if generator.generar(registro_seleccionado, experiencias, ruta_salida):
                            QMessageBox.information(
                                self.view,
                                "Éxito",
                                f"CV generado correctamente en:\n{ruta_salida}"
                            )
                            # Abrir el archivo generado
                            from os import startfile
                            startfile(ruta_salida)
                        else:
                            QMessageBox.warning(
                                self.view,
                                "Advertencia",
                                "No se pudo generar el CV. Verifique que los datos estén complet
                            )
                    except Exception as e:
                        QMessageBox.critical(
                            self.view,
                            "Error",
                            f"Error al generar CV: {str(e)}"
                        )
            except Exception as e:
                logger.error(f"Error al generar CV: {str(e)}")
                QMessageBox.critical(self.view, "Error", f"Error al generar CV: {str(e)}")

        def obtener_datos_formulario(self):
            """Obtiene todos los datos del formulario de registro"""
            try:
                tab_registro = self.view.tab_registro
                return {
                    'codigo': tab_registro.codigo_input.text(),
                    'tipo_documento': tab_registro.tipo_doc_combo.currentText(),
                    'numero_documento': tab_registro.num_doc_input.text(),
                    'nombres': tab_registro.nombres_input.text(),
                    'apellidos': tab_registro.apellidos_input.text(),
                    'profesion': tab_registro.profesion_input.text(),
                    'fecha_nacimiento': tab_registro.fecha_nac_input.date().toString('yyyy-MM-dd'),
                    'lugar_nacimiento': tab_registro.lugar_nac_input.text(),
                    'registro_cip': tab_registro.registro_cip_input.text(),
                    'area': tab_registro.area_combo.currentText(),
                    'cargo': tab_registro.cargo_input.text(),
                    'residencia': tab_registro.residencia_input.text(),
                    'telefono': tab_registro.telefono_input.text(),
                    'correo': tab_registro.correo_input.text(),
                    'trabajado_eimsa': tab_registro.check_eimsa.isChecked(),
                    'experiencias': self.obtener_experiencias_formulario()
                }
            except Exception as e:
```

```
            logger.error(f"Error al obtener datos del formulario: {str(e)}")
            raise

    def obtener_experiencias_formulario(self):
        """Obtiene la lista de experiencias del formulario"""
        experiencias = []
        for widget in self.experiencias_widgets:
            experiencias.append(widget.obtener_datos())
        return experiencias

    def agregar_experiencia(self):
        try:
            dialogo = ExperienciaWidget(parent=self.view)
            if dialogo.exec_():  # Usar exec_ en lugar de exec() para Qt
                datos = dialogo.obtener_datos()
                self.view.tab_registro.experiencias_layout.addWidget(
                    ExperienciaWidget(datos=datos, parent=self.view.tab_registro)
                )
        except Exception as e:
            logger.error(f"Error al agregar experiencia: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al agregar experiencia: {str(e)}")

    def eliminar_experiencia(self, widget):
        """Elimina una experiencia del formulario"""
        try:
            widget.setParent(None)
            widget.deleteLater()
            self.experiencias_widgets.remove(widget)
        except Exception as e:
            logger.error(f"Error al eliminar experiencia: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al eliminar experiencia: {str(e)}")

    def actualizar_experiencia(self, datos):
        """Actualiza los datos de una experiencia"""
        try:
            # Aquí puedes agregar lógica adicional si necesitas
            # hacer algo cuando se actualiza una experiencia
            pass
        except Exception as e:
            logger.error(f"Error al actualizar experiencia: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al actualizar experiencia: {str(e)}

    def cargar_datos_registro(self, datos, modo_lectura=False):
        """Carga los datos en el formulario de registro"""
        try:
            tab_registro = self.view.tab_registro

            # Limpiar experiencias existentes
            for widget in self.experiencias_widgets:
                widget.setParent(None)
                widget.deleteLater()
            self.experiencias_widgets.clear()

            # Cargar datos básicos
            tab_registro.codigo_input.setText(datos.get('codigo', ''))
            tab_registro.tipo_doc_combo.setCurrentText(datos.get('tipo_documento', 'DNI'))
            tab_registro.num_doc_input.setText(datos.get('numero_documento', ''))
            tab_registro.nombres_input.setText(datos.get('nombres', ''))
```

```
            tab_registro.apellidos_input.setText(datos.get('apellidos', ''))
            tab_registro.profesion_input.setText(datos.get('profesion', ''))
            # ... (continuar con el resto de campos)

            # Cargar experiencias
            experiencias = self.db.obtener_experiencias(datos['codigo'])
            for exp in experiencias:
                widget = ExperienciaWidget(datos=exp, parent=self.view.tab_registro)
                widget.eliminado.connect(lambda w=widget: self.eliminar_experiencia(w))
                widget.actualizado.connect(self.actualizar_experiencia)
                self.experiencias_widgets.append(widget)
                self.view.tab_registro.experiencias_layout.addWidget(widget)

            # Configurar modo lectura si es necesario
            if modo_lectura:
                self.set_modo_lectura(True)

        except Exception as e:
            logger.error(f"Error al cargar datos en registro: {str(e)}")
            QMessageBox.critical(self.view, "Error", f"Error al cargar datos: {str(e)}")

    def set_modo_lectura(self, enabled=True):
        """Configura el formulario en modo lectura"""
        tab_registro = self.view.tab_registro
        for widget in tab_registro.findChildren((QLineEdit, QComboBox, QDateEdit)):
            widget.setReadOnly(enabled)
            widget.setEnabled(not enabled)
        tab_registro.btn_guardar.setVisible(not enabled)
        # Deshabilitar botones de experiencia en modo lectura
        for exp_widget in self.experiencias_widgets:
            exp_widget.set_modo_lectura(enabled)

    def __init__(self):
        try:
            logger.debug("Inicializando MainController...")
            self.view = MainView()
            self.db = Database()
            self.experiencias_widgets = []  # Lista para mantener referencia a widgets de experi
            self.setup_connections()
            self.cargar_datos_iniciales()
            logger.debug("MainController inicializado correctamente")
        except Exception as e:
            logger.error(f"Error al inicializar MainController: {str(e)}", exc_info=True)
            raise
```

models__init__.py

```
from .code_generator import CodeGenerator
from .obra_types import TipoObra, CalculadorExperiencia, CategoriaObra, ExperienciaObra

__all__ = [
    'CodeGenerator',
    'TipoObra',
    'CalculadorExperiencia',
    'CategoriaObra',
    'ExperienciaObra'
]
```

models\code_generator.py

```python
# utils/code_generator.py
from datetime import datetime
import re
from typing import Optional

class CodeGenerator:
    def __init__(self):
        self._counter = 0
        self._used_codes = set()

    def generate_code(self, tipo: str = 'O') -> str:
        """
        Genera un código único con el formato O/A-000001-EI-PE
        Args:
            tipo: 'O' para Obra, 'A' para Administración
        Returns:
            Código único generado
        """
        if tipo not in ['O', 'A']:
            raise ValueError("Tipo debe ser 'O' para Obra o 'A' para Administración")

        self._counter += 1
        code = f"{tipo}-{self._counter:06d}-EI-PE"

        while code in self._used_codes:
            self._counter += 1
            code = f"{tipo}-{self._counter:06d}-EI-PE"

        self._used_codes.add(code)
        return code

    def validate_code(self, code: str) -> bool:
        """
        Valida el formato del código
        """
        pattern = r'^[OA]-\d{6}-EI-PE$'
        return bool(re.match(pattern, code))

class ExperienceCalculator:
    @staticmethod
    def calculate_years(fecha_inicio: str, fecha_fin: Optional[str] = None) -> float:
        """
        Calcula los años de experiencia entre dos fechas
        """
        inicio = datetime.strptime(fecha_inicio, '%Y-%m-%d')
        fin = datetime.strptime(fecha_fin, '%Y-%m-%d') if fecha_fin else datetime.now()

        diff = fin - inicio
        return round(diff.days / 365.25, 2)

    @staticmethod
    def calculate_total_experience(experiencias: list) -> float:
        """
        Calcula la experiencia total sumando todas las experiencias
        """
        total = 0.0
```

```
        for exp in experiencias:
            total += ExperienceCalculator.calculate_years(
                exp['fecha_inicio'],
                exp.get('fecha_fin')
            )
        return round(total, 2)

class MontoValidator:
    @staticmethod
    def format_monto(monto: float) -> str:
        """
        Formatea un monto en millones
        """
        if monto < 1_000_000:
            return f"{monto:,.2f}"

        millones = monto / 1_000_000
        return f"{millones:,.2f}MM"

    @staticmethod
    def get_range(monto: float) -> str:
        """
        Obtiene el rango del monto del proyecto
        """
        if monto < 5_000_000:
            return "< 5 MM"
        elif monto < 20_000_000:
            return "5 - 20 MM"
        elif monto < 50_000_000:
            return "20 - 50 MM"
        else:
            return "> 50 MM"
```

models\database.py

```python
# app/models/database.py
import sqlite3
from pathlib import Path
from app.config import Config
import os
import logging
from datetime import datetime

class MontoRango:
    MENOR_5MM = 1
    ENTRE_5_20MM = 2
    ENTRE_20_50MM = 3
    ENTRE_50_100MM = 4

    @staticmethod
    def obtener_rango(monto):
        """Determina el rango de un monto en millones"""
        try:
            monto = float(monto)
            if monto < 5_000_000:
                return MontoRango.MENOR_5MM
            elif monto < 20_000_000:
                return MontoRango.ENTRE_5_20MM
            elif monto < 50_000_000:
                return MontoRango.ENTRE_20_50MM
            else:
                return MontoRango.ENTRE_50_100MM
        except:
            return MontoRango.MENOR_5MM

    @staticmethod
    def obtener_texto_rango(rango):
        rangos = {
            MontoRango.MENOR_5MM: "< 5 MM",
            MontoRango.ENTRE_5_20MM: "5 - 20 MM",
            MontoRango.ENTRE_20_50MM: "20 - 50 MM",
            MontoRango.ENTRE_50_100MM: "50 - 100 MM"
        }
        return rangos.get(rango, "No especificado")

logger = logging.getLogger(__name__)

class Database:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            try:
                cls._instance = super().__new__(cls)
                db_path = Path(Config.DB_PATH)
                db_path.parent.mkdir(parents=True, exist_ok=True)
                cls._instance.conn = sqlite3.connect(str(db_path))
                cls._instance.conn.row_factory = sqlite3.Row
                logger.info(f"Conexión a base de datos establecida: {db_path}")
                cls._instance.crear_tablas()
            except Exception as e:
```

```
                logger.error(f"Error al crear instancia de base de datos: {str(e)}")
                raise
        return cls._instance

    def crear_tablas(self):
        """Crea todas las tablas necesarias en la base de datos"""
        try:
            with self.conn:
                # Tabla de datos personales
                self.conn.execute('''
                CREATE TABLE IF NOT EXISTS datos_personales (
                    codigo TEXT PRIMARY KEY,
                    tipo_documento TEXT,
                    numero_documento TEXT,
                    nombres TEXT NOT NULL,
                    apellidos TEXT,
                    profesion TEXT NOT NULL,
                    fecha_nacimiento TEXT,
                    lugar_nacimiento TEXT,
                    registro_cip TEXT,
                    area TEXT NOT NULL,
                    cargo TEXT NOT NULL,
                    residencia TEXT,
                    telefono TEXT,
                    correo TEXT,
                    trabajo_previo_eimsa BOOLEAN DEFAULT 0,
                    fecha_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                )
                ''')

                # Tabla de imágenes actualizada para almacenar BLOB
                self.conn.execute('''
                CREATE TABLE IF NOT EXISTS imagenes_cv (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    codigo_personal TEXT,
                    experiencia_id INTEGER,
                    nombre_archivo TEXT,
                    contenido_imagen BLOB,
                    tipo_imagen TEXT,
                    fecha_subida TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    FOREIGN KEY (codigo_personal) REFERENCES datos_personales(codigo),
                    FOREIGN KEY (experiencia_id) REFERENCES experiencia_laboral(id)
                )
                ''')


                # Tabla de experiencia laboral
                self.conn.execute('''
                CREATE TABLE IF NOT EXISTS experiencia_laboral (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    codigo_personal TEXT NOT NULL,
                    empresa TEXT NOT NULL,
                    obra TEXT NOT NULL,
                    detalle_obra TEXT,
                    tipo_obra TEXT NOT NULL,
                    monto_proyecto TEXT,
                    cargo TEXT NOT NULL,
                    fecha_inicio TEXT NOT NULL,
```

```
                    fecha_fin TEXT,
                    propietario TEXT,
                    funciones TEXT,
                    FOREIGN KEY (codigo_personal) REFERENCES datos_personales(codigo) ON DELETE
                )
                ''')

                # Tabla para documentos
                self.conn.execute('''
                CREATE TABLE IF NOT EXISTS documentos (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    codigo_personal TEXT,
                    experiencia_id INTEGER NULL,
                    tipo_documento TEXT,
                    nombre_archivo TEXT,
                    ruta_archivo TEXT,
                    descripcion TEXT,
                    fecha_subida TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    FOREIGN KEY (codigo_personal) REFERENCES datos_personales(codigo),
                    FOREIGN KEY (experiencia_id) REFERENCES experiencia_laboral(id)
                )
                ''')


                # Tabla de tipos de obra actualizada
                self.conn.execute('''
                CREATE TABLE IF NOT EXISTS tipos_obra (
                    id TEXT PRIMARY KEY,
                    categoria TEXT NOT NULL,
                    nombre TEXT NOT NULL,
                    descripcion TEXT
                )
                ''')

                # Insertar tipos de obra predefinidos
                tipos_obra = [
                    # Industria
                    ('PCAL', 'Industria', 'Planta de Cemento y Cal', 'Plantas de procesamiento')
                    ('NIND', 'Industria', 'Naves Industriales', 'Construcción de naves'),
                    ('HIDRO', 'Industria', 'Hidroeléctrica', 'Centrales hidroeléctricas'),
                    ('TERMO', 'Industria', 'Central Termoeléctrica', 'Centrales térmicas'),
                    ('PELEC', 'Industria', 'Planta Eléctrica', 'Plantas eléctricas'),
                    ('PUENT', 'Industria', 'Puentes', 'Construcción de puentes'),
                    ('PETRO', 'Industria', 'Planta Petróleo', 'Plantas petroleras'),
                    ('CIVIL', 'Industria', 'Obras Civiles', 'Obras civiles'),

                    # Minería
                    ('PGAS', 'Minería', 'Planta de Gas', 'Plantas de gas'),
                    ('ESTRU', 'Minería', 'Construcción Estructuras', 'Estructuras mineras'),
                    ('MPESA', 'Minería', 'Montaje de Estructura Pesada', 'Montajes pesados'),
                    ('CHANC', 'Minería', 'Chancadora/Fajas (seca)', 'Sistemas de chancado'),
                    ('MOLIN', 'Minería', 'Molinos/Celda de Flotación', 'Sistemas de molienda'),
                    ('TUNEL', 'Minería', 'Túneles', 'Construcción de túneles'),
                    ('DESMO', 'Minería', 'Desmontaje', 'Trabajos de desmontaje'),

                    # Servicios
                    ('ELEC', 'Servicios', 'Electricidad & Instrumentación', 'Sistemas eléctricos
                    ('TUBE', 'Servicios', 'Tubería Agua/Relaves', 'Sistemas de tubería'),
```

```
                    ('DESAL', 'Servicios', 'Planta Desalinizadora', 'Desalinización'),
                    ('TRUCK', 'Servicios', 'Truckshop - Almacenes', 'Talleres y almacenes')
                ]

                # Primero, limpiar la tabla de tipos de obra
                self.conn.execute('DELETE FROM tipos_obra')

                # Insertar los nuevos tipos de obra
                self.conn.executemany('''
                INSERT INTO tipos_obra (id, categoria, nombre, descripcion)
                VALUES (?, ?, ?, ?)
                ''', tipos_obra)

                logger.info("Todas las tablas creadas/verificadas exitosamente")

        except Exception as e:
            logger.error(f"Error al crear tablas: {str(e)}")
            raise

    def inicializar_tipos_obra(self):
        try:
            tipos_obra = [
                ('IND_CEM', 'Planta de Cemento y Cal', 'Industria de cemento y cal'),
                ('NAV_IND', 'Naves Industriales', 'Construcción de naves industriales'),
                ('HIDRO', 'Hidroeléctrica', 'Centrales hidroeléctricas'),
                ('PUENTE', 'Puentes', 'Construcción de puentes'),
                ('PET', 'Plantas de Petróleo', 'Infraestructura petrolera'),
                ('TUB', 'Tuberías', 'Sistemas de tuberías'),
                ('GAS', 'Planta de Gas', 'Plantas procesadoras de gas'),
                ('MIN', 'Minería', 'Proyectos mineros'),
                ('TUNEL', 'Túneles', 'Construcción de túneles'),
                ('DESM', 'Desmontaje', 'Trabajos de desmontaje'),
                ('CIV', 'Obras Civiles', 'Construcciones civiles'),
                ('ELEC', 'Electricidad e Instrumentación', 'Sistemas eléctricos'),
                ('DESAL', 'Planta Desalinizadora', 'Plantas de desalinización'),
                ('TRUCK', 'Truck Shop', 'Talleres y almacenes')
            ]

            with self.conn:
                for tipo in tipos_obra:
                    self.conn.execute('''
                        INSERT OR IGNORE INTO tipos_obra (id, nombre, descripcion)
                        VALUES (?, ?, ?)
                    ''', tipo)
            logger.debug("Tipos de obra inicializados correctamente")
        except Exception as e:
            logger.error(f"Error al inicializar tipos de obra: {str(e)}")
            raise

    def guardar_registro(self, datos):
        try:
            with self.conn:
                cursor = self.conn.cursor()
                logger.debug(f"Iniciando guardado de registro para código: {datos.get('codigo',

                # Si hay monto_proyecto, convertir a decimal
                if 'monto_proyecto' in datos:
                    monto_str = datos['monto_proyecto']
```

```
                    # Limpiar el string de moneda
                    monto_num = ''.join(filter(str.isdigit, monto_str))
                    if monto_num:
                        datos['monto_decimal'] = float(monto_num)

                # Extraer las experiencias
                experiencias = datos.pop('experiencias', [])

                if datos['codigo']:  # Actualización
                    cursor.execute('''
                        UPDATE datos_personales SET
                            nombres=?, profesion=?, fecha_nacimiento=?,
                            lugar_nacimiento=?, registro_cip=?, area=?,
                            cargo=?, residencia=?, telefono=?, correo=?
                        WHERE codigo=?
                    ''', (
                        datos['nombres'], datos['profesion'], datos['fecha_nacimiento'],
                        datos['lugar_nacimiento'], datos['registro_cip'], datos['area'],
                        datos['cargo'], datos['residencia'], datos['telefono'],
                        datos['correo'], datos['codigo']
                    ))
                    logger.debug(f"Actualizado registro existente: {datos['codigo']}")
                else:  # Nuevo registro
                    datos['codigo'] = self.generar_codigo(datos['area'])
                    cursor.execute('''
                        INSERT INTO datos_personales (
                            codigo, nombres, profesion, fecha_nacimiento,
                            lugar_nacimiento, registro_cip, area, cargo,
                            residencia, telefono, correo, fecha_registro
                        ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP)
                    ''', (
                        datos['codigo'], datos['nombres'], datos['profesion'],
                        datos['fecha_nacimiento'], datos['lugar_nacimiento'],
                        datos['registro_cip'], datos['area'], datos['cargo'],
                        datos['residencia'], datos['telefono'], datos['correo']
                    ))
                    logger.debug(f"Creado nuevo registro: {datos['codigo']}")

                # Manejar las experiencias
                experiencias_actuales = set()
                logger.debug("Iniciando procesamiento de experiencias")

                for exp in experiencias:
                    if 'id' in exp:  # Experiencia existente
                        experiencias_actuales.add(exp['id'])
                        cursor.execute('''
                            UPDATE experiencia_laboral SET
                                empresa=?, obra=?, tipo_obra=?,
                                detalle_obra=?, monto_proyecto=?, cargo=?,
                                fecha_inicio=?, fecha_fin=?, propietario=?, funciones=?
                            WHERE id=? AND codigo_personal=?
                        ''', (
                            exp['empresa'], exp['obra'], exp['tipo_obra'],
                            exp['detalle_obra'], exp['monto_proyecto'], exp['cargo'],
                            exp['fecha_inicio'], exp['fecha_fin'], exp['propietario'],
                            ';'.join(exp['funciones']) if isinstance(exp['funciones'], list) els
                            exp['id'], datos['codigo']
                        ))
```

```
                        exp_id = exp['id']
                        logger.debug(f"Actualizada experiencia ID: {exp_id}")
                    else:  # Nueva experiencia
                        cursor.execute('''
                            INSERT INTO experiencia_laboral (
                                codigo_personal, empresa, obra, tipo_obra,
                                detalle_obra, monto_proyecto, cargo,
                                fecha_inicio, fecha_fin, propietario, funciones
                            ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
                        ''', (
                            datos['codigo'], exp['empresa'], exp['obra'],
                            exp['tipo_obra'], exp['detalle_obra'], exp['monto_proyecto'],
                            exp['cargo'], exp['fecha_inicio'], exp['fecha_fin'],
                            exp['propietario'], ';'.join(exp['funciones']) if isinstance(exp['fu
                        ))
                        exp_id = cursor.lastrowid
                        experiencias_actuales.add(exp_id)
                        logger.debug(f"Creada nueva experiencia ID: {exp_id}")

                    # Manejar documentos/imágenes
                    if 'documentos' in exp and exp['documentos']:
                        logger.debug(f"Procesando documentos para experiencia ID: {exp_id}")
                        for doc in exp['documentos']:
                            with open(doc['ruta'], 'rb') as img_file:
                                contenido_imagen = img_file.read()

                            cursor.execute('''
                                INSERT INTO imagenes_cv (
                                    codigo_personal, experiencia_id, nombre_archivo,
                                    contenido_imagen, tipo_imagen
                                ) VALUES (?, ?, ?, ?, ?)
                            ''', (
                                datos['codigo'],
                                exp_id,
                                doc['nombre'],
                                contenido_imagen,
                                'experiencia'
                            ))
                            logger.debug(f"Guardada imagen: {doc['nombre']}")

                # Eliminar experiencias que ya no existen
                if experiencias_actuales:
                    cursor.execute('''
                        DELETE FROM experiencia_laboral
                        WHERE codigo_personal = ? AND id NOT IN ({})
                    '''.format(','.join('?' * len(experiencias_actuales))),
                        [datos['codigo']] + list(experiencias_actuales))
                    logger.debug("Eliminadas experiencias obsoletas")

                logger.info(f"Registro guardado exitosamente: {datos['codigo']}")
                return datos['codigo']

            except Exception as e:
                logger.error(f"Error al guardar registro: {str(e)}")
                raise Exception(f"Error al guardar en la base de datos: {str(e)}")

        def obtener_registro(self, codigo):
            try:
```

```
            cursor = self.conn.cursor()
            logger.debug(f"Buscando registro con código: {codigo}")

            # Obtener datos personales
            cursor.execute('SELECT * FROM datos_personales WHERE codigo = ?', (codigo,))
            datos_personales = cursor.fetchone()

            if datos_personales:
                datos = dict(datos_personales)
                logger.debug(f"Datos personales encontrados para código: {codigo}")

                # Obtener experiencias
                cursor.execute('''
                    SELECT * FROM experiencia_laboral
                    WHERE codigo_personal = ?
                    ORDER BY fecha_inicio DESC
                ''', (codigo,))

                experiencias = []
                for exp in cursor.fetchall():
                    exp_dict = dict(exp)
                    logger.debug(f"Procesando experiencia ID: {exp_dict['id']}")

                    # Obtener imágenes para esta experiencia
                    cursor.execute('''
                        SELECT id, nombre_archivo, contenido_imagen, tipo_imagen
                        FROM imagenes_cv
                        WHERE codigo_personal = ? AND experiencia_id = ?
                    ''', (codigo, exp_dict['id']))

                    documentos = []
                    for img in cursor.fetchall():
                        # Guardar temporalmente la imagen
                        temp_path = os.path.join(Config.TEMP_DIR, img['nombre_archivo'])
                        with open(temp_path, 'wb') as f:
                            f.write(img['contenido_imagen'])

                        documentos.append({
                            'nombre': img['nombre_archivo'],
                            'ruta': temp_path,
                            'tipo': img['tipo_imagen']
                        })
                        logger.debug(f"Imagen procesada: {img['nombre_archivo']}")
                    exp_dict['documentos'] = documentos
                    experiencias.append(exp_dict)

                datos['experiencias'] = experiencias
                logger.info(f"Registro obtenido exitosamente: {codigo}")
                return datos

            logger.warning(f"No se encontró registro para código: {codigo}")
            return None

        except Exception as e:
            logger.error(f"Error al obtener registro {codigo}: {str(e)}")
            raise Exception(f"Error al obtener registro: {str(e)}")

    def generar_codigo(self, area):
```

```
        try:
            cursor = self.conn.cursor()
            cursor.execute('''
                SELECT codigo FROM datos_personales
                WHERE codigo LIKE 'O-%'
                ORDER BY codigo DESC LIMIT 1
            ''')
            result = cursor.fetchone()

            if result:
                ultimo_num = int(result[0].split('-')[1])
                nuevo_num = ultimo_num + 1
            else:
                nuevo_num = 1

            nuevo_codigo = f"O-{nuevo_num:06d}-EI-PE"
            logger.debug(f"Generado nuevo código: {nuevo_codigo}")
            return nuevo_codigo
        except Exception as e:
            logger.error(f"Error al generar código: {str(e)}")
            raise

    def obtener_todos_registros(self):
        """Obtiene todos los registros de la base de datos"""
        try:
            cursor = self.conn.cursor()
            cursor.execute('''
                SELECT dp.*, COUNT(el.id) as total_experiencias
                FROM datos_personales dp
                LEFT JOIN experiencia_laboral el ON dp.codigo = el.codigo_personal
                GROUP BY dp.codigo
                ORDER BY dp.fecha_registro DESC
            ''')
            logger.debug("Obteniendo todos los registros")
            return [dict(row) for row in cursor.fetchall()]
        except Exception as e:
            logger.error(f"Error al obtener todos los registros: {str(e)}")
            raise

    def buscar_registros(self, filtros):
        """
        Busca registros según los filtros proporcionados
        filtros: diccionario con los criterios de búsqueda
        """
        try:
            logger.debug(f"Iniciando búsqueda con filtros: {filtros}")
            query = """
                SELECT DISTINCT dp.*
                FROM datos_personales dp
                LEFT JOIN experiencia_laboral el ON dp.codigo = el.codigo_personal
                WHERE 1=1
            """
            params = []

            if filtros.get('nombres'):
                query += " AND dp.nombres LIKE ?"
                params.append(f"%{filtros['nombres']}%")

```

```
            if filtros.get('area') and filtros['area'] != 'Todas las áreas':
                query += " AND dp.area = ?"
                params.append(filtros['area'])

            if filtros.get('cargo'):
                query += " AND dp.cargo LIKE ?"
                params.append(f"%{filtros['cargo']}%")

            if filtros.get('residencia'):
                query += " AND dp.residencia LIKE ?"
                params.append(f"%{filtros['residencia']}%")

            cursor = self.conn.cursor()
            cursor.execute(query, params)
            resultados = cursor.fetchall()
            logger.debug(f"Búsqueda completada. Resultados encontrados: {len(resultados)}")
            return resultados
        except Exception as e:
            logger.error(f"Error en búsqueda de registros: {str(e)}")
            raise

    def eliminar_registro(self, codigo):
        """Elimina un registro y todos sus datos relacionados"""
        try:
            logger.debug(f"Iniciando eliminación de registro: {codigo}")
            with self.conn:
                # Eliminar imágenes primero
                self.conn.execute("""
                    DELETE FROM imagenes_cv
                    WHERE codigo_personal = ?
                """, (codigo,))

                # Eliminar experiencias
                self.conn.execute("""
                    DELETE FROM experiencia_laboral
                    WHERE codigo_personal = ?
                """, (codigo,))

                # Eliminar datos personales
                self.conn.execute("""
                    DELETE FROM datos_personales
                    WHERE codigo = ?
                """, (codigo,))

            logger.info(f"Registro eliminado exitosamente: {codigo}")
            return True
        except Exception as e:
            logger.error(f"Error al eliminar registro {codigo}: {str(e)}")
            raise

    def verificar_duplicado(self, nombres, registro_cip):
        """Verifica si ya existe un registro con el mismo nombre o CIP"""
        try:
            cursor = self.conn.cursor()
            cursor.execute("""
                SELECT codigo FROM datos_personales
                WHERE nombres = ? OR registro_cip = ?
            """, (nombres, registro_cip))
```

```
521
522             resultado = cursor.fetchone()
523             if resultado:
524                 logger.warning(f"Encontrado registro duplicado: {resultado['codigo']}")
525             return resultado is not None
526         except Exception as e:
527             logger.error(f"Error al verificar duplicados: {str(e)}")
528             raise
529
530     def obtener_experiencias_por_tipo(self, tipo_obra):
531         """Obtiene todas las experiencias de un tipo de obra específico"""
532         try:
533             cursor = self.conn.cursor()
534             cursor.execute("""
535                 SELECT dp.nombres, el.*
536                 FROM experiencia_laboral el
537                 JOIN datos_personales dp ON el.codigo_personal = dp.codigo
538                 WHERE el.tipo_obra = ?
539                 ORDER BY el.fecha_inicio DESC
540             """, (tipo_obra,))
541
542             logger.debug(f"Obteniendo experiencias para tipo de obra: {tipo_obra}")
543             return cursor.fetchall()
544         except Exception as e:
545             logger.error(f"Error al obtener experiencias por tipo: {str(e)}")
546             raise
547
548     def actualizar_imagenes(self, experiencia_id, imagenes):
549         """Actualiza las imágenes de una experiencia"""
550         try:
551             logger.debug(f"Actualizando imágenes para experiencia ID: {experiencia_id}")
552             with self.conn:
553                 # Eliminar imágenes existentes
554                 self.conn.execute("""
555                     DELETE FROM imagenes_cv
556                     WHERE experiencia_id = ?
557                 """, (experiencia_id,))
558
559                 # Insertar nuevas imágenes
560                 for img in imagenes:
561                     with open(img['ruta'], 'rb') as f:
562                         contenido = f.read()
563
564                     self.conn.execute("""
565                         INSERT INTO imagenes_cv (
566                             codigo_personal, experiencia_id,
567                             nombre_archivo, contenido_imagen, tipo_imagen
568                         ) VALUES (?, ?, ?, ?, ?)
569                     """, (
570                         img['codigo_personal'],
571                         experiencia_id,
572                         img['nombre'],
573                         contenido,
574                         img['tipo']
575                     ))
576
577             logger.info(f"Imágenes actualizadas para experiencia ID: {experiencia_id}")
578         except Exception as e:
```

```
            logger.error(f"Error al actualizar imágenes: {str(e)}")
            raise

    def exportar_datos(self, ruta_archivo):
        """Exporta todos los datos a un archivo SQL"""
        try:
            logger.debug(f"Iniciando exportación de datos a: {ruta_archivo}")
            with open(ruta_archivo, 'w') as f:
                for line in self.conn.iterdump():
                    f.write(f'{line}\n')
            logger.info(f"Datos exportados exitosamente a: {ruta_archivo}")
            return True
        except Exception as e:
            logger.error(f"Error al exportar datos: {str(e)}")
            raise

    def importar_datos(self, ruta_archivo):
        """Importa datos desde un archivo SQL"""
        try:
            logger.debug(f"Iniciando importación de datos desde: {ruta_archivo}")
            with open(ruta_archivo, 'r') as f:
                sql = f.read()
            self.conn.executescript(sql)
            logger.info(f"Datos importados exitosamente desde: {ruta_archivo}")
            return True
        except Exception as e:
            logger.error(f"Error al importar datos: {str(e)}")
            raise

    def backup_database(self):
        """Realiza una copia de seguridad de la base de datos"""
        try:
            fecha = datetime.now().strftime('%Y%m%d_%H%M%S')
            backup_path = os.path.join(Config.DATA_DIR, f'backup_{fecha}.db')

            # Crear una copia de la base de datos
            with open(backup_path, 'wb') as backup_file:
                backup_file.write(self.conn.execute('vacuum into ?', (backup_path,)))

            logger.info(f"Backup creado exitosamente: {backup_path}")
            return backup_path
        except Exception as e:
            logger.error(f"Error al crear backup: {str(e)}")
            raise

    def obtener_experiencias(self, codigo_personal):
        """Obtiene todas las experiencias de un personal específico"""
        try:
            cursor = self.conn.cursor()
            cursor.execute('''
            SELECT * FROM experiencia_laboral
            WHERE codigo_personal = ?
            ORDER BY fecha_inicio DESC
            ''', (codigo_personal,))

            return [dict(row) for row in cursor.fetchall()]
        except Exception as e:
            logger.error(f"Error al obtener experiencias: {str(e)}")
```

```
637                 raise
```

models\experience_filter.py

```python
# models/experience_filter.py
from typing import List, Dict, Any
from datetime import datetime
from dataclasses import dataclass

@dataclass
class FilterCriteria:
    area: str = None
    cargo: str = None
    tipo_obra: str = None
    rango_monto: str = None
    ex_eimsa: bool = False
    años_experiencia: int = None
    residencia: str = None

class ExperienceFilter:
    def __init__(self, database_connection):
        self.db = database_connection

    def apply_filters(self, criteria: FilterCriteria) -> List[Dict[str, Any]]:
        """
        Aplica los filtros especificados a la búsqueda de experiencias
        """
        query = """
        SELECT DISTINCT
            p.*,
            e.tipo_obra,
            e.monto_proyecto,
            e.cargo as ultimo_cargo
        FROM
            datos_personales p
        LEFT JOIN experiencia_laboral e ON
            p.codigo = e.codigo_personal
        WHERE 1=1
        """
        params = []

        if criteria.area:
            query += " AND p.area = ?"
            params.append(criteria.area)

        if criteria.cargo:
            query += " AND (p.cargo_referencia LIKE ? OR e.cargo LIKE ?)"
            params.extend([f"%{criteria.cargo}%", f"%{criteria.cargo}%"])

        if criteria.tipo_obra:
            query += " AND e.tipo_obra = ?"
            params.append(criteria.tipo_obra)

        if criteria.rango_monto:
            ranges = {
                "< 5 MM": (0, 5_000_000),
                "5 - 20 MM": (5_000_000, 20_000_000),
                "20 - 50 MM": (20_000_000, 50_000_000),
                "> 50 MM": (50_000_000, float('inf'))
            }
```

```
            min_val, max_val = ranges[criteria.rango_monto]
            query += " AND e.monto_proyecto >= ? AND e.monto_proyecto < ?"
            params.extend([min_val, max_val])

        if criteria.ex_eimsa:
            query += " AND p.trabajo_previo_eimsa = 1"

        if criteria.residencia:
            query += " AND p.lugar_residencia LIKE ?"
            params.append(f"%{criteria.residencia}%")

        cursor = self.db.execute(query, params)
        results = cursor.fetchall()

        # Filtrar por años de experiencia si se especifica
        if criteria.años_experiencia:
            filtered_results = []
            for result in results:
                total_exp = self._calculate_total_experience(result['codigo'])
                if total_exp >= criteria.años_experiencia:
                    result['años_experiencia'] = total_exp
                    filtered_results.append(result)
            return filtered_results

        return results

    def _calculate_total_experience(self, codigo_personal: str) -> float:
        """
        Calcula la experiencia total para un profesional
        """
        query = """
        SELECT
            fecha_inicio,
            fecha_fin
        FROM
            experiencia_laboral
        WHERE
            codigo_personal = ?
        """
        cursor = self.db.execute(query, (codigo_personal,))
        experiences = cursor.fetchall()

        total = 0.0
        for exp in experiences:
            inicio = datetime.strptime(exp['fecha_inicio'], '%Y-%m-%d')
            fin = datetime.strptime(exp['fecha_fin'], '%Y-%m-%d') if exp['fecha_fin'] else datet
            total += (fin - inicio).days / 365.25

        return round(total, 2)

    def get_experience_summary(self, codigo_personal: str) -> Dict[str, Any]:
        """
        Obtiene un resumen de la experiencia por tipo de obra
        """
        query = """
        SELECT
            tipo_obra,
            COUNT(*) as total_proyectos,
```

```
            SUM(
                (julianday(COALESCE(fecha_fin, date('now'))) -
                 julianday(fecha_inicio)) / 365.25
            ) as años_experiencia,
            MAX(monto_proyecto) as mayor_monto
        FROM
            experiencia_laboral
        WHERE
            codigo_personal = ?
        GROUP BY
            tipo_obra
        """
        cursor = self.db.execute(query, (codigo_personal,))
        return {row['tipo_obra']: dict(row) for row in cursor.fetchall()}
```

models\experience_manager.py

```python
# models/experience_manager.py
from typing import List, Dict, Any, Optional
from datetime import datetime
from dataclasses import dataclass
import sqlite3

@dataclass
class ExperienciaLaboral:
    empresa: str
    obra: str
    tipo_obra: str
    cargo: str
    fecha_inicio: str
    fecha_fin: Optional[str]
    propietario: str
    detalle_obra: str
    monto_proyecto: float
    codigo_personal: str
    id: Optional[int] = None

class ExperienceManager:
    def __init__(self, database_connection):
        self.db = database_connection

    def add_experience(self, experiencia: ExperienciaLaboral) -> int:
        """
        Agrega una nueva experiencia laboral
        Returns:
            ID de la experiencia creada
        """
        query = """
        INSERT INTO experiencia_laboral (
            codigo_personal, empresa, obra, tipo_obra,
            cargo, fecha_inicio, fecha_fin, propietario,
            detalle_obra, monto_proyecto
        ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """
        cursor = self.db.execute(query, (
            experiencia.codigo_personal,
            experiencia.empresa,
            experiencia.obra,
            experiencia.tipo_obra,
            experiencia.cargo,
            experiencia.fecha_inicio,
            experiencia.fecha_fin,
            experiencia.propietario,
            experiencia.detalle_obra,
            experiencia.monto_proyecto
        ))
        self.db.commit()
        return cursor.lastrowid

    def update_experience(self, experiencia: ExperienciaLaboral) -> bool:
        """
        Actualiza una experiencia existente
        """
```

```
57            if not experiencia.id:
58                raise ValueError("ID de experiencia no proporcionado")
59
60            query = """
61            UPDATE experiencia_laboral SET
62                empresa = ?,
63                obra = ?,
64                tipo_obra = ?,
65                cargo = ?,
66                fecha_inicio = ?,
67                fecha_fin = ?,
68                propietario = ?,
69                detalle_obra = ?,
70                monto_proyecto = ?
71            WHERE id = ? AND codigo_personal = ?
72            """
73            cursor = self.db.execute(query, (
74                experiencia.empresa,
75                experiencia.obra,
76                experiencia.tipo_obra,
77                experiencia.cargo,
78                experiencia.fecha_inicio,
79                experiencia.fecha_fin,
80                experiencia.propietario,
81                experiencia.detalle_obra,
82                experiencia.monto_proyecto,
83                experiencia.id,
84                experiencia.codigo_personal
85            ))
86            self.db.commit()
87            return cursor.rowcount > 0
88
89        def get_experiences(self, codigo_personal: str) -> List[Dict[str, Any]]:
90            """
91            Obtiene todas las experiencias de un profesional
92            """
93            query = """
94            SELECT
95                *,
96                (julianday(COALESCE(fecha_fin, date('now'))) -
97                 julianday(fecha_inicio)) / 365.25 as duracion_años
98            FROM
99                experiencia_laboral
100           WHERE
101               codigo_personal = ?
102           ORDER BY
103               fecha_inicio DESC
104           """
105           cursor = self.db.execute(query, (codigo_personal,))
106           return cursor.fetchall()
107
108       def delete_experience(self, id: int, codigo_personal: str) -> bool:
109           """
110           Elimina una experiencia laboral
111           """
112           query = """
113           DELETE FROM experiencia_laboral
114           WHERE id = ? AND codigo_personal = ?
```

```
        """
        cursor = self.db.execute(query, (id, codigo_personal))
        self.db.commit()
        return cursor.rowcount > 0

    def get_experience_by_type(self, tipo_obra: str) -> List[Dict[str, Any]]:
        """
        Obtiene todas las experiencias de un tipo específico
        """
        query = """
        SELECT
            e.*,
            p.nombres,
            p.profesion,
            (julianday(COALESCE(e.fecha_fin, date('now'))) -
             julianday(e.fecha_inicio)) / 365.25 as duracion_años
        FROM
            experiencia_laboral e
        JOIN
            datos_personales p ON e.codigo_personal = p.codigo
        WHERE
            e.tipo_obra = ?
        ORDER BY
            e.fecha_inicio DESC
        """
        cursor = self.db.execute(query, (tipo_obra,))
        return cursor.fetchall()
```

models\experiencia.py

```python
# app/models/experiencia.py
from dataclasses import dataclass
from datetime import datetime
from typing import List, Optional
from .database import Database

@dataclass
class Experiencia:
    codigo_personal: str
    empresa: str
    obra: str
    tipo_obra: str
    cargo: str
    fecha_inicio: str
    fecha_fin: Optional[str]
    detalle_obra: Optional[str] = None
    monto_proyecto: Optional[str] = None
    propietario: Optional[str] = None
    funciones: Optional[str] = None
    id: Optional[int] = None

    def guardar(self) -> bool:
        db = Database()
        try:
            with db.conn:
                cursor = db.conn.cursor()
                cursor.execute('''
                    INSERT INTO experiencia_laboral (
                        codigo_personal, empresa, obra, detalle_obra,
                        tipo_obra, monto_proyecto, cargo, fecha_inicio,
                        fecha_fin, propietario, funciones
                    ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
                ''', (
                    self.codigo_personal, self.empresa, self.obra,
                    self.detalle_obra, self.tipo_obra, self.monto_proyecto,
                    self.cargo, self.fecha_inicio, self.fecha_fin,
                    self.propietario, self.funciones
                ))
                self.id = cursor.lastrowid
                return True
        except Exception as e:
            print(f"Error al guardar experiencia: {e}")
            return False

    def actualizar(self) -> bool:
        if not self.id:
            return False

        db = Database()
        try:
            with db.conn:
                db.conn.execute('''
                    UPDATE experiencia_laboral
                    SET empresa=?, obra=?, detalle_obra=?, tipo_obra=?,
                        monto_proyecto=?, cargo=?, fecha_inicio=?,
                        fecha_fin=?, propietario=?, funciones=?
```

```
                    WHERE id=?
                ''', (
                    self.empresa, self.obra, self.detalle_obra,
                    self.tipo_obra, self.monto_proyecto, self.cargo,
                    self.fecha_inicio, self.fecha_fin, self.propietario,
                    self.funciones, self.id
                ))
                return True
        except Exception as e:
            print(f"Error al actualizar experiencia: {e}")
            return False

    @staticmethod
    def obtener_por_personal(codigo_personal: str) -> List['Experiencia']:
        db = Database()
        cursor = db.conn.cursor()

        cursor.execute('''
            SELECT * FROM experiencia_laboral
            WHERE codigo_personal = ?
            ORDER BY fecha_inicio DESC
        ''', (codigo_personal,))

        resultados = cursor.fetchall()
        return [Experiencia(**dict(row)) for row in resultados]

    def calcular_duracion(self) -> int:
        """Calcula la duración en meses de la experiencia"""
        inicio = datetime.strptime(self.fecha_inicio, '%Y-%m-%d')
        fin = datetime.strptime(self.fecha_fin, '%Y-%m-%d') if self.fecha_fin else datetime.now(

        return ((fin.year - inicio.year) * 12 + (fin.month - inicio.month))
```

models\obra_types.py

```python
from enum import Enum
from typing import List, Dict, Optional
from dataclasses import dataclass

@dataclass
class ExperienciaObra:
    """Clase para almacenar la experiencia en un tipo de obra"""
    años: float
    meses: int
    proyectos: List[str]

class CategoriaObra(Enum):
    INDUSTRIA = "Industria"
    MINERIA = "Minería"
    ENERGIA = "Energía"
    INFRAESTRUCTURA = "Infraestructura"
    SERVICIOS = "Servicios"

class TipoObra:
    def __init__(self):
        self.tipos = {
            # Industria
            "CEMENTO_CAL": {
                "nombre": "Planta de Cemento y Cal",
                "categoria": CategoriaObra.INDUSTRIA
            },
            "NAVES_INDUSTRIALES": {
                "nombre": "Naves Industriales",
                "categoria": CategoriaObra.INDUSTRIA
            },

            # Minería
            "CHANCADORAS": {
                "nombre": "Chancadoras/Fajas",
                "categoria": CategoriaObra.MINERIA
            },
            "MOLINOS": {
                "nombre": "Molinos/Celdas de Flotación",
                "categoria": CategoriaObra.MINERIA
            },

            # Energía
            "HIDROELECTRICA": {
                "nombre": "Hidroeléctrica",
                "categoria": CategoriaObra.ENERGIA
            },
            "TERMOELECTRICA": {
                "nombre": "Central Termoeléctrica",
                "categoria": CategoriaObra.ENERGIA
            },

            # Infraestructura
            "TUBERIAS": {
                "nombre": "Tubería Agua/Relaves",
                "categoria": CategoriaObra.INFRAESTRUCTURA
            },
```

```
            "TRUCK_SHOP": {
                "nombre": "Truck Shop/Almacenes",
                "categoria": CategoriaObra.INFRAESTRUCTURA
            },

            # Servicios
            "ELECTR_INSTRUM": {
                "nombre": "Electricidad e Instrumentación",
                "categoria": CategoriaObra.SERVICIOS
            },
            "MONTAJE": {
                "nombre": "Montaje de Estructuras Pesadas",
                "categoria": CategoriaObra.SERVICIOS
            }
        }

    def get_categorias(self) -> List[CategoriaObra]:
        """Obtiene todas las categorías disponibles"""
        return list(set(tipo["categoria"] for tipo in self.tipos.values()))

    def get_tipos_por_categoria(self, categoria: CategoriaObra) -> List[str]:
        """Obtiene los tipos de obra de una categoría específica"""
        return [
            codigo for codigo, datos in self.tipos.items()
            if datos["categoria"] == categoria
        ]

    def get_nombre(self, codigo: str) -> Optional[str]:
        """Obtiene el nombre de un tipo de obra por su código"""
        if codigo in self.tipos:
            return self.tipos[codigo]["nombre"]
        return None

class CalculadorExperiencia:
    def __init__(self):
        self.experiencias: Dict[str, ExperienciaObra] = {}

    def agregar_experiencia(self, tipo_obra: str, años: float, proyecto: str):
        """
        Agrega experiencia para un tipo de obra específico
        """
        if tipo_obra not in self.experiencias:
            self.experiencias[tipo_obra] = ExperienciaObra(
                años=años,
                meses=int(años * 12),
                proyectos=[proyecto]
            )
        else:
            exp = self.experiencias[tipo_obra]
            exp.años += años
            exp.meses = int(exp.años * 12)
            exp.proyectos.append(proyecto)

    def get_experiencia_total(self) -> float:
        """
        Calcula la experiencia total en años
        """
        return sum(exp.años for exp in self.experiencias.values())
```

```
115
116       def get_experiencia_por_categoria(self, categoria: CategoriaObra) -> Dict[str, ExperienciaOb
117           """
118           Obtiene la experiencia agrupada por categoría
119           """
120           tipos_obra = TipoObra()
121           resultado = {}
122
123           for tipo, exp in self.experiencias.items():
124               if tipos_obra.tipos[tipo]["categoria"] == categoria:
125                   resultado[tipo] = exp
126
127           return resultado
128
```

models\persona.py

```python
# models/persona.py
from dataclasses import dataclass
from datetime import date
from typing import List, Optional
from enum import Enum

class TipoDocumento(Enum):
    DNI = "DNI"
    RUC = "RUC"
    CE = "CE"  # Carnet de Extranjería
    CPP = "CPP" # Carnet de Permiso Temporal
    PASAPORTE = "PASAPORTE"

@dataclass
class DatosPersonales:
    # Campos requeridos (sin valores por defecto)
    codigo: str
    tipo_documento: TipoDocumento
    numero_documento: str
    nombres: str
    apellidos: str
    profesion: str
    area: str
    cargo_actual: str

    # Campos opcionales (con valores por defecto)
    fecha_nacimiento: Optional[date] = None
    lugar_nacimiento: Optional[str] = None
    nacionalidad: str = "Peruana"
    telefono: Optional[str] = None
    correo: Optional[str] = None
    direccion: Optional[str] = None
    ciudad_residencia: Optional[str] = None
    pais_residencia: str = "Perú"
    numero_colegiatura: Optional[str] = None
    años_experiencia: float = 0.0
    fecha_registro: date = date.today()
    trabajado_eimsa: bool = False
    estado_cv: str = "Activo"

@dataclass
class Experiencia:
    # Campos requeridos
    empresa: str
    cargo: str
    fecha_inicio: date
    obra: str
    tipo_obra: str
    propietario: str

    # Campos opcionales
    id: Optional[int] = None
    fecha_fin: Optional[date] = None
    detalle_obra: str = ""
    ubicacion: str = ""
    monto_proyecto: float = 0.0
```

```
    moneda_proyecto: str = "PEN"
    funciones: List[str] = None
    logros: List[str] = None
    tecnologias_usadas: List[str] = None
    documentos: List[str] = None

    def __post_init__(self):
        if self.funciones is None:
            self.funciones = []
        if self.logros is None:
            self.logros = []
        if self.tecnologias_usadas is None:
            self.tecnologias_usadas = []
        if self.documentos is None:
            self.documentos = []

    @property
    def duracion_meses(self) -> int:
        fin = self.fecha_fin or date.today()
        return (fin.year - self.fecha_inicio.year) * 12 + (fin.month - self.fecha_inicio.month)

@dataclass
class CV:
    datos_personales: DatosPersonales
    experiencias: List[Experiencia]
    habilidades: List[str] = None
    certificaciones: List[str] = None
    idiomas: List[dict] = None
    educacion: List[dict] = None

    def __post_init__(self):
        if self.habilidades is None:
            self.habilidades = []
        if self.certificaciones is None:
            self.certificaciones = []
        if self.idiomas is None:
            self.idiomas = []
        if self.educacion is None:
            self.educacion = []
```

models\personal.py

```python
# app/models/personal.py
from dataclasses import dataclass
from datetime import datetime
from typing import List, Optional
from .database import Database

@dataclass
class Personal:
    codigo: str
    nombres: str
    tipo_documento: str
    numero_documento: str
    profesion: str
    area: str
    cargo: str
    fecha_nacimiento: Optional[str] = None
    lugar_nacimiento: Optional[str] = None
    registro_cip: Optional[str] = None
    residencia: Optional[str] = None
    telefono: Optional[str] = None
    correo: Optional[str] = None
    fecha_registro: Optional[str] = None

    @staticmethod
    def generar_codigo(area: str) -> str:
        db = Database()
        cursor = db.conn.cursor()

        # Obtener el último código para el área
        cursor.execute('''
            SELECT codigo FROM datos_personales
            WHERE codigo LIKE ?
            ORDER BY codigo DESC LIMIT 1
        ''', (f'O-____-EI-PE',))

        ultimo_codigo = cursor.fetchone()

        if ultimo_codigo:
            # Extraer el número y aumentar en 1
            num = int(ultimo_codigo[0].split('-')[1]) + 1
        else:
            num = 1

        return f'O-{num:05d}-EI-PE'

    def guardar(self) -> bool:
        db = Database()
        try:
            with db.conn:
                db.conn.execute('''
                    INSERT INTO datos_personales (
                        codigo, nombres, profesion, fecha_nacimiento,
                        lugar_nacimiento, registro_cip, area, cargo,
                        residencia, telefono, correo
                    ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
                ''', (
```

```
                    self.codigo, self.nombres, self.profesion,
                    self.fecha_nacimiento, self.lugar_nacimiento,
                    self.registro_cip, self.area, self.cargo,
                    self.residencia, self.telefono, self.correo
                ))
                return True
        except Exception as e:
            print(f"Error al guardar personal: {e}")
            return False

    def actualizar(self) -> bool:
        db = Database()
        try:
            with db.conn:
                db.conn.execute('''
                    UPDATE datos_personales
                    SET nombres=?, profesion=?, fecha_nacimiento=?,
                        lugar_nacimiento=?, registro_cip=?, area=?,
                        cargo=?, residencia=?, telefono=?, correo=?
                    WHERE codigo=?
                ''', (
                    self.nombres, self.profesion, self.fecha_nacimiento,
                    self.lugar_nacimiento, self.registro_cip, self.area,
                    self.cargo, self.residencia, self.telefono,
                    self.correo, self.codigo
                ))
                return True
        except Exception as e:
            print(f"Error al actualizar personal: {e}")
            return False

    @staticmethod
    def buscar(texto: str, area: Optional[str] = None) -> List['Personal']:
        db = Database()
        cursor = db.conn.cursor()

        query = '''
            SELECT * FROM datos_personales
            WHERE (nombres LIKE ? OR codigo LIKE ?)
        '''
        params = [f'%{texto}%', f'%{texto}%']

        if area and area != 'Todas las áreas':
            query += ' AND area = ?'
            params.append(area)

        cursor.execute(query, params)
        resultados = cursor.fetchall()

        return [Personal(**dict(row)) for row in resultados]

    @staticmethod
    def obtener_por_codigo(codigo: str) -> Optional['Personal']:
        db = Database()
        cursor = db.conn.cursor()

        cursor.execute('SELECT * FROM datos_personales WHERE codigo = ?', (codigo,))
        resultado = cursor.fetchone()
```

```
115
116             if resultado:
117                 return Personal(**dict(resultado))
118             return None
```

models\project_filters.py

```python
# app/models/project_filters.py

from enum import Enum
from typing import List, Dict, Optional
from dataclasses import dataclass
from decimal import Decimal

class MontoRango(Enum):
    """Rangos de montos de proyecto definidos"""
    MENOR_5MM = "< 5 MM"
    ENTRE_5_20MM = "5 - 20 MM"
    ENTRE_20_50MM = "20 - 50 MM"
    ENTRE_50_100MM = "50 - 100 MM"
    MAYOR_100MM = "> 100 MM"

@dataclass
class ProyectoMonto:
    """Clase para manejar el monto de un proyecto"""
    valor: Decimal
    moneda: str

    def get_rango(self) -> MontoRango:
        """Determina el rango del monto del proyecto"""
        monto_mm = float(self.valor) / 1_000_000

        if monto_mm < 5:
            return MontoRango.MENOR_5MM
        elif monto_mm < 20:
            return MontoRango.ENTRE_5_20MM
        elif monto_mm < 50:
            return MontoRango.ENTRE_20_50MM
        elif monto_mm < 100:
            return MontoRango.ENTRE_50_100MM
        else:
            return MontoRango.MAYOR_100MM

class ProjectFilters:
    """Clase para manejar los filtros de búsqueda de proyectos y personal"""

    def __init__(self, db_connection):
        self.db = db_connection

    def filter_by_residence(self, residence: str) -> List[Dict]:
        """
        Filtra personal por lugar de residencia
        """
        query = """
        SELECT * FROM datos_personales
        WHERE LOWER(residencia) LIKE LOWER(?)
        """
        return self.db.execute(query, (f"%{residence}%",)).fetchall()

    def filter_by_name(self, name: str) -> List[Dict]:
        """
        Filtra personal por nombre o apellido
        """
```

```
            query = """
            SELECT * FROM datos_personales
            WHERE LOWER(nombres) LIKE LOWER(?)
            """
            return self.db.execute(query, (f"%{name}%",)).fetchall()

        def filter_by_area(self, area: str) -> List[Dict]:
            """
            Filtra personal por área
            """
            query = """
            SELECT * FROM datos_personales
            WHERE area = ?
            """
            return self.db.execute(query, (area,)).fetchall()

        def filter_by_reference_position(self, position: str) -> List[Dict]:
            """
            Filtra personal por cargo de referencia
            """
            query = """
            SELECT DISTINCT dp.*
            FROM datos_personales dp
            JOIN experiencia_laboral el ON dp.codigo = el.codigo_personal
            WHERE LOWER(el.cargo) LIKE LOWER(?)
            """
            return self.db.execute(query, (f"%{position}%",)).fetchall()

        def filter_by_project_amount(self, rango: MontoRango) -> List[Dict]:
            """
            Filtra proyectos por rango de monto
            """
            ranges = {
                MontoRango.MENOR_5MM: (0, 5_000_000),
                MontoRango.ENTRE_5_20MM: (5_000_000, 20_000_000),
                MontoRango.ENTRE_20_50MM: (20_000_000, 50_000_000),
                MontoRango.ENTRE_50_100MM: (50_000_000, 100_000_000),
                MontoRango.MAYOR_100MM: (100_000_000, float('inf'))
            }

            min_val, max_val = ranges[rango]

            query = """
            SELECT DISTINCT dp.*
            FROM datos_personales dp
            JOIN experiencia_laboral el ON dp.codigo = el.codigo_personal
            WHERE CAST(REPLACE(REPLACE(el.monto_proyecto, 'S/', ''), '$', '') AS DECIMAL)
            BETWEEN ? AND ?
            """
            return self.db.execute(query, (min_val, max_val)).fetchall()

        def filter_previous_eimsa(self) -> List[Dict]:
            """
            Filtra personal que trabajó anteriormente en EIMSA
            """
            query = """
            SELECT * FROM datos_personales
            WHERE trabajo_previo_eimsa = 1
```

```
            """
            return self.db.execute(query).fetchall()

    # Actualización de la estructura de la base de datos
    DB_UPDATES = """
    -- Agregar campo para trabajo previo en EIMSA
    ALTER TABLE datos_personales ADD COLUMN trabajo_previo_eimsa BOOLEAN DEFAULT 0;

    -- Actualizar tabla de experiencia_laboral para manejar montos como decimal
    ALTER TABLE experiencia_laboral ADD COLUMN monto_decimal DECIMAL(15,2);
    """
```

models\tipo_obra.py

```python
# app/models/tipo_obra.py
from dataclasses import dataclass
from typing import List
from .database import Database
from typing import Optional


@dataclass
class TipoObra:
    id: str
    nombre: str
    descripcion: Optional[str] = None

    @staticmethod
    def obtener_todos() -> List['TipoObra']:
        db = Database()
        cursor = db.conn.cursor()

        cursor.execute('SELECT * FROM tipos_obra ORDER BY nombre')
        resultados = cursor.fetchall()

        return [TipoObra(**dict(row)) for row in resultados]

    @staticmethod
    def obtener_por_id(id: str) -> Optional['TipoObra']:
        db = Database()
        cursor = db.conn.cursor()

        cursor.execute('SELECT * FROM tipos_obra WHERE id = ?', (id,))
        resultado = cursor.fetchone()

        if resultado:
            return TipoObra(**dict(resultado))
        return None
```

utils__init__.py

```
# app/utils/__init__.py
from .cv_generator import CVGenerator
from .validators import validate_email, validate_phone, validate_date
from .formatters import format_currency, format_date
from .code_generator import generate_code

__all__ = [
    'CVGenerator',
    'validate_email',
    'validate_phone',
    'validate_date',
    'format_currency',
    'format_date',
    'generate_code'
]
```

utils\code_generator.py

```python
# app/utils/code_generator.py
import random
import string
from datetime import datetime
from typing import Optional
from app.models.database import Database

def generate_code(area: str = 'O', company: str = 'EI',
                  country: str = 'PE') -> str:
    """
    Genera un código único para un nuevo registro
    Formato: O-000001-EI-PE
    """
    db = Database()
    cursor = db.conn.cursor()

    # Obtener el último código
    cursor.execute('''
        SELECT codigo FROM datos_personales
        WHERE codigo LIKE ?
        ORDER BY codigo DESC LIMIT 1
    ''', (f'{area}-%',))

    result = cursor.fetchone()

    if result:
        # Extraer el número del último código y aumentar en 1
        last_number = int(result[0].split('-')[1])
        new_number = last_number + 1
    else:
        new_number = 1

    # Generar el nuevo código
    return f"{area}-{new_number:06d}-{company}-{country}"

def generate_temp_code() -> str:
    """
    Genera un código temporal para archivos o registros temporales
    """
    timestamp = datetime.now().strftime('%Y%m%d%H%M%S')
    random_chars = ''.join(random.choices(string.ascii_uppercase + string.digits, k=4))
    return f"TEMP-{timestamp}-{random_chars}"
```

utils\cv_generator.py

```
from docx import Document
from docx.shared import Pt, Inches, Cm
from docx.enum.text import WD_ALIGN_PARAGRAPH
from docx.enum.section import WD_ORIENT
from datetime import datetime
import os
from app.config import Config
from docx.oxml import parse_xml
from PIL import Image

class CVGenerator:
    def __init__(self):
        self.template_path = Config.CV_TEMPLATE_PATH
        if not os.path.exists(self.template_path):
            raise FileNotFoundError(f"No se encontró la plantilla en: {self.template_path}")

    def _agregar_imagen(self, doc, ruta_imagen, nombre):
        """Método para agregar imagen con formato correcto"""
        try:
            # Agregar párrafo para la imagen
            img_paragraph = doc.add_paragraph()
            img_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER

            # Obtener dimensiones de la imagen
            with Image.open(ruta_imagen) as img:
                width, height = img.size

            # Calcular el ancho máximo (15 cm, manteniendo la proporción)
            max_width = Cm(15)
            aspect_ratio = height / width
            width = max_width
            height = width * aspect_ratio

            # Agregar la imagen centrada
            run = img_paragraph.add_run()
            run.add_picture(ruta_imagen, width=width)

            # Espacio después de la imagen
            doc.add_paragraph()

        except Exception as e:
            print(f"Error al agregar imagen {nombre}: {str(e)}")

    def generar(self, datos_personales, experiencias, output_path):
        try:
            # Usar la plantilla existente en lugar de crear un nuevo documento
            doc = Document(self.template_path)

            # DATOS PERSONALES
            titulo_dp = doc.add_paragraph('DATOS PERSONALES')
            titulo_dp.runs[0].font.bold = True
            titulo_dp.runs[0].font.size = Pt(16)

            # Datos personales con indentación
            campos = [
                ('APELLIDOS Y NOMBRES', datos_personales['nombres']),
```

```
                ('PROFESION', datos_personales['profesion']),
                ('FECHA DE NACIMIENTO', datos_personales['fecha_nacimiento']),
                ('LUGAR DE NACIMIENTO', datos_personales['lugar_nacimiento']),
                ('REGISTRO C.I.P.', datos_personales.get('registro_cip', '')),
                ('RESIDENCIA', datos_personales.get('residencia', '')),
                ('TELÉFONO', datos_personales.get('telefono', '')),
                ('CORREO', datos_personales.get('correo', ''))
            ]

            # Tabla invisible para alineación
            table = doc.add_table(rows=len(campos), cols=3)
            table.allow_autofit = True

            # Ajustar ancho y quitar bordes
            for row in table.rows:
                for cell in row.cells:
                    cell._tc.get_or_add_tcPr().append(parse_xml(r'<w:tcW xmlns:w="http://schemas
                    cell._element.get_or_add_tcPr().append(parse_xml(r'<w:tcBorders xmlns:w="htt

            # Agregar datos con indentación
            for i, (label, valor) in enumerate(campos):
                cells = table.rows[i].cells

                # Primera celda: etiqueta
                label_para = cells[0].paragraphs[0]
                label_para.paragraph_format.left_indent = Inches(0.5)
                run = label_para.add_run(label)
                run.font.bold = True

                # Segunda celda: dos puntos centrados
                cells[1].paragraphs[0].alignment = WD_ALIGN_PARAGRAPH.CENTER
                cells[1].paragraphs[0].add_run(' : ')

                # Tercera celda: valor
                cells[2].paragraphs[0].add_run(valor)

            doc.add_paragraph()  # Espacio entre secciones

            # EXPERIENCIA PROFESIONAL
            titulo_exp = doc.add_paragraph('EXPERIENCIA PROFESIONAL')
            titulo_exp.runs[0].font.bold = True
            titulo_exp.runs[0].font.size = Pt(16)

            # Experiencias
            for exp in experiencias:
                campos_exp = [
                    ('Empresa', exp['empresa']),
                    ('Obra', exp['obra']),
                    ('Detalle de la obra', exp['detalle_obra']),
                    ('Monto del proyecto', exp['monto_proyecto']),
                    ('Cargo', exp['cargo']),
                    ('Periodo', f"{exp['fecha_inicio']} - {exp['fecha_fin']}"),
                    ('Propietario', exp['propietario'])
                ]

                # Tabla invisible para cada experiencia
                exp_table = doc.add_table(rows=len(campos_exp), cols=3)
                exp_table.allow_autofit = True
```

```
115
116                     # Quitar bordes
117                     for row in exp_table.rows:
118                         for cell in row.cells:
119                             cell._tc.get_or_add_tcPr().append(parse_xml(r'<w:tcW xmlns:w="http://sch
120                             cell._element.get_or_add_tcPr().append(parse_xml(r'<w:tcBorders xmlns:w=
121
122                     # Agregar datos de experiencia
123                     for i, (label, valor) in enumerate(campos_exp):
124                         cells = exp_table.rows[i].cells
125
126                         # Etiqueta con indentación
127                         label_para = cells[0].paragraphs[0]
128                         label_para.paragraph_format.left_indent = Inches(0.5)
129                         run = label_para.add_run(label)
130                         run.font.bold = True
131
132                         # Dos puntos centrados
133                         cells[1].paragraphs[0].alignment = WD_ALIGN_PARAGRAPH.CENTER
134                         cells[1].paragraphs[0].add_run(' : ')
135
136                         # Valor
137                         cells[2].paragraphs[0].add_run(valor)
138
139                     # Funciones
140                     if exp.get('funciones'):
141                         func_para = doc.add_paragraph()
142                         func_para.paragraph_format.left_indent = Inches(0.5)
143                         func_para.add_run('Funciones:').bold = True
144
145                         funciones = exp['funciones']
146                         if isinstance(funciones, str):
147                             funciones = funciones.split(';')
148
149                         for funcion in funciones:
150                             if funcion.strip():
151                                 bullet_para = doc.add_paragraph()
152                                 bullet_para.paragraph_format.left_indent = Inches(0.7)
153                                 bullet_para.paragraph_format.space_after = Pt(0)
154                                 bullet_para.paragraph_format.space_before = Pt(0)
155                                 check = bullet_para.add_run('✓')
156                                 check.font.size = Pt(9)
157                                 bullet_para.add_run(' ' + funcion.strip())
158
159                     # Imágenes de esta experiencia
160                     if 'documentos' in exp and exp['documentos']:
161                         for img in exp['documentos']:
162                             if os.path.exists(img['ruta']):
163                                 self._agregar_imagen(doc, img['ruta'], img['nombre'])
164
165                     doc.add_paragraph()  # Espacio entre experiencias
166
167                 # Documentos generales después de todas las experiencias
168                 if 'imagenes' in datos_personales and datos_personales['imagenes']:
169                     titulo_img = doc.add_paragraph('DOCUMENTOS ADJUNTOS')
170                     titulo_img.runs[0].font.bold = True
171                     titulo_img.runs[0].font.size = Pt(16)
172
```

```
                for img in datos_personales['imagenes']:
                    if os.path.exists(img['ruta']):
                        self._agregar_imagen(doc, img['ruta'], img['nombre'])

            doc.save(output_path)
            return True

        except Exception as e:
            print(f"Error generando CV: {e}")
            return False
```

utils\formatters.py

```python
# app/utils/formatters.py
from datetime import datetime
import locale

def format_currency(amount: str) -> str:
    """
    Formatea un monto monetario
    """
    try:
        # Limpiar el string de caracteres no numéricos
        clean_amount = ''.join(filter(lambda x: x.isdigit() or x == '.', amount))
        value = float(clean_amount)

        # Formatear con separadores de miles
        locale.setlocale(locale.LC_ALL, '')
        return locale.currency(value, grouping=True, symbol='')
    except:
        return amount

def format_date(date_str: str, input_format: str = '%Y-%m-%d',
                output_format: str = '%d/%m/%Y') -> str:
    """
    Formatea una fecha al formato deseado
    """
    try:
        if date_str:
            date_obj = datetime.strptime(date_str, input_format)
            return date_obj.strftime(output_format)
        return ""
    except:
        return date_str

def format_phone(phone: str) -> str:
    """
    Formatea un número telefónico
    """
    if not phone:
        return ""

    # Limpiar el número
    clean_number = ''.join(filter(str.isdigit, phone))

    if len(clean_number) == 9:  # Número celular Perú
        return f"{clean_number[:3]} {clean_number[3:6]} {clean_number[6:]}"
    return phone
```

utils\setup_check.py

```
# En utils/setup_check.py
import os
from app.config import Config

def verificar_plantilla_cv():
    if not os.path.exists(Config.CV_TEMPLATE_PATH):
        print(f"ADVERTENCIA: No se encontró la plantilla de CV en {Config.CV_TEMPLATE_PATH}")
        print("Creando directorio de templates...")
        os.makedirs(os.path.dirname(Config.CV_TEMPLATE_PATH), exist_ok=True)
        print(f"Por favor, coloque su plantilla de CV en: {Config.CV_TEMPLATE_PATH}")
        return False
    return True
```

utils\validators.py

```python
# app/utils/validators.py
import re
from datetime import datetime
from typing import Tuple

def validate_email(email: str) -> Tuple[bool, str]:
    """
    Valida el formato de un correo electrónico
    """
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    if not email:
        return True, ""  # El email es opcional
    if re.match(pattern, email):
        return True, ""
    return False, "Formato de correo electrónico inválido"

def validate_phone(phone: str) -> Tuple[bool, str]:
    """
    Valida el formato de un número telefónico
    """
    pattern = r'^\+?[0-9]{9,15}$'
    if not phone:
        return True, ""  # El teléfono es opcional
    if re.match(pattern, phone):
        return True, ""
    return False, "Formato de teléfono inválido"

def validate_date(date_str: str) -> Tuple[bool, str]:
    """
    Valida el formato de una fecha (YYYY-MM-DD)
    """
    try:
        if date_str:
            datetime.strptime(date_str, '%Y-%m-%d')
        return True, ""
    except ValueError:
        return False, "Formato de fecha inválido (YYYY-MM-DD)"

def validate_required(value: str, field_name: str) -> Tuple[bool, str]:
    """
    Valida que un campo requerido no esté vacío
    """
    if not value or not value.strip():
        return False, f"El campo {field_name} es requerido"
    return True, ""
```

views__init__.py

```
# app/views/__init__.py
from .login_view import LoginView
from .main_view import MainView
from .personal_view import RegistroPersonalView
from .busqueda_view import BusquedaView
from .experiencia_view import ExperienciaView
from .detalle_view import DetalleView

__all__ = [
    'LoginView',
    'MainView',
    'RegistroPersonalView',
    'BusquedaView',
    'ExperienciaView',
    'DetalleView'
]
```

views\busqueda_view.py

```
# app/views/busqueda_view.py
from PyQt5.QtWidgets import (QWidget, QVBoxLayout, QHBoxLayout, QLineEdit,
    QPushButton, QTableWidget, QTableWidgetItem, QLabel, QComboBox,
    QCheckBox, QGroupBox)
from PyQt5.QtCore import pyqtSignal, Qt
from PyQt5.QtGui import QColor
from app.config import Config
from app.models.database import Database  # Añadida esta importación

class BusquedaView(QWidget):
    registro_seleccionado = pyqtSignal(str)

    def __init__(self):
        super().__init__()
        self.db = Database()  # Inicializar conexión a base de datos
        self.setup_ui()

    def setup_ui(self):
        layout = QVBoxLayout()

        # Grupo de filtros
        filtros_group = QGroupBox("Filtros de Búsqueda")
        filtros_layout = QVBoxLayout()

        # Primera fila de filtros
        filtros_row1 = QHBoxLayout()

        self.busqueda_input = QLineEdit()
        self.busqueda_input.setPlaceholderText("Buscar por nombre o código...")

        self.area_filter = QComboBox()
        self.area_filter.addItems(['Todas las areas'] + Config.AREAS)

        filtros_row1.addWidget(QLabel("Buscar:"))
        filtros_row1.addWidget(self.busqueda_input)
        filtros_row1.addWidget(QLabel("Área:"))
        filtros_row1.addWidget(self.area_filter)

        # Segunda fila de filtros
        filtros_row2 = QHBoxLayout()

        self.tipo_obra_filter = QComboBox()
        self.tipo_obra_filter.addItems(['Todos los tipos'] + Config.TIPOS_OBRA)

        self.monto_filter = QComboBox()
        self.monto_filter.addItems(['Todos los montos', '< 5 MM', '5 - 20 MM', '20 - 50 MM', '50

        filtros_row2.addWidget(QLabel("Tipo de Obra:"))
        filtros_row2.addWidget(self.tipo_obra_filter)
        filtros_row2.addWidget(QLabel("Rango de Monto:"))
        filtros_row2.addWidget(self.monto_filter)

        # Checkbox para EIMSA
        self.eimsa_check = QCheckBox("Ex-empleados EIMSA")

        # Añadir filtros al layout
```

```
57            filtros_layout.addLayout(filtros_row1)
58            filtros_layout.addLayout(filtros_row2)
59            filtros_layout.addWidget(self.eimsa_check)
60            filtros_group.setLayout(filtros_layout)
61            layout.addWidget(filtros_group)
62
63            # Botones de acción
64            actions_layout = QHBoxLayout()
65            self.btn_buscar = QPushButton("Buscar")
66            self.btn_limpiar = QPushButton("Limpiar Filtros")
67            self.btn_ver_detalle = QPushButton("Ver Detalle")
68            self.btn_generar_cv = QPushButton("Generar CV")
69            self.btn_editar = QPushButton("Editar")
70
71            # Estilo para los botones
72            self.btn_buscar.setStyleSheet("background-color: #3498db; color: white;")
73            self.btn_limpiar.setStyleSheet("background-color: #3498db; color: white;")
74            self.btn_ver_detalle.setStyleSheet("background-color: #95a5a6; color: white;")
75            self.btn_generar_cv.setStyleSheet("background-color: #95a5a6; color: white;")
76            self.btn_editar.setStyleSheet("background-color: #95a5a6; color: white;")
77
78            actions_layout.addWidget(self.btn_buscar)
79            actions_layout.addWidget(self.btn_limpiar)
80            actions_layout.addWidget(self.btn_ver_detalle)
81            actions_layout.addWidget(self.btn_generar_cv)
82            actions_layout.addWidget(self.btn_editar)
83
84            layout.addLayout(actions_layout)
85
86            # Tabla de resultados
87            self.label_resultados = QLabel("Resultados:")
88            layout.addWidget(self.label_resultados)
89            self.tabla_resultados = QTableWidget()
90            self.tabla_resultados.setColumnCount(8)
91            self.tabla_resultados.setHorizontalHeaderLabels([
92                "Código", "Nombre", "Profesión", "Área",
93                "Tipo de Obra", "Monto", "Ex-EIMSA",
94                "Última Actualización"
95            ])
96
97            self.tabla_resultados.setAlternatingRowColors(True)
98            self.tabla_resultados.setSelectionBehavior(QTableWidget.SelectRows)
99            self.tabla_resultados.setSelectionMode(QTableWidget.SingleSelection)
100           self.tabla_resultados.horizontalHeader().setStretchLastSection(True)
101
102           layout.addWidget(self.tabla_resultados)
103
104           self.setLayout(layout)
105
106           # Conectar señales
107           self.tabla_resultados.itemSelectionChanged.connect(self.on_selection_change)
108           self.busqueda_input.returnPressed.connect(self.btn_buscar.click)
109
110       def on_selection_change(self):
111           """Habilita/deshabilita botones según selección"""
112           tiene_seleccion = len(self.tabla_resultados.selectedItems()) > 0
113           self.btn_ver_detalle.setEnabled(tiene_seleccion)
114           self.btn_generar_cv.setEnabled(tiene_seleccion)
```

```
            self.btn_editar.setEnabled(tiene_seleccion)

        def cargar_registros(self, registros=None):
            try:
                if registros is None:
                    registros = self.db.obtener_todos_registros()

                self.tabla_resultados.setRowCount(0)
                for row, datos in enumerate(registros):
                    self.tabla_resultados.insertRow(row)
                    items = [
                        datos['codigo'],
                        datos['nombres'],
                        datos['profesion'],
                        datos['area'],
                        datos.get('tipo_obra', ''),
                        datos.get('monto_proyecto', ''),
                        '✓' if datos.get('trabajo_previo_eimsa') else '',
                        datos.get('fecha_registro', '')
                    ]

                    for col, valor in enumerate(items):
                        item = QTableWidgetItem(str(valor))
                        item.setFlags(item.flags() & ~Qt.ItemIsEditable)
                        self.tabla_resultados.setItem(row, col, item)

                self.tabla_resultados.resizeColumnsToContents()
            except Exception as e:
                print(f"Error al cargar registros: {e}")

        def mostrar_resultados(self, resultados):
            """Muestra los resultados en la tabla"""
            self.tabla_resultados.setRowCount(0)
            for row, datos in enumerate(resultados):
                self.tabla_resultados.insertRow(row)
                items = [
                    datos.get('codigo', ''),
                    datos.get('nombres', ''),
                    datos.get('profesion', ''),
                    datos.get('area', ''),
                    datos.get('cargo', ''),
                    datos.get('monto_proyecto', ''),
                    '✓' if datos.get('trabajo_previo_eimsa') else '',
                    datos.get('fecha_registro', '')
                ]

                for col, valor in enumerate(items):
                    item = QTableWidgetItem(str(valor))
                    item.setFlags(item.flags() & ~Qt.ItemIsEditable)
                    self.tabla_resultados.setItem(row, col, item)

            self.tabla_resultados.resizeColumnsToContents()

        def obtener_codigo_seleccionado(self):
            items = self.tabla_resultados.selectedItems()
            if items:
                row = items[0].row()
                return self.tabla_resultados.item(row, 0).text()
```

```
            return None

        def limpiar_filtros(self):
            self.busqueda_input.clear()
            self.area_filter.setCurrentIndex(0)
            self.tipo_obra_filter.setCurrentIndex(0)
            self.monto_filter.setCurrentIndex(0)
            self.eimsa_check.setChecked(False)
```

views\detalle_view.py

```
from PyQt5.QtWidgets import (
    QWidget, QVBoxLayout, QHBoxLayout, QLabel, QPushButton,
    QScrollArea, QFrame
)
from PyQt5.QtCore import Qt


class DetalleView(QWidget):
    def __init__(self):
        super().__init__()
        self.setup_ui()

    def setup_ui(self):
        layout = QVBoxLayout()

        # Área scrollable para el contenido
        scroll = QScrollArea()
        scroll.setWidgetResizable(True)

        # Widget contenedor
        self.content_widget = QWidget()
        self.content_layout = QVBoxLayout(self.content_widget)

        # Sección de datos personales
        self.datos_personales = QFrame()
        self.datos_personales.setFrameStyle(QFrame.StyledPanel)
        datos_layout = QVBoxLayout(self.datos_personales)

        self.lbl_titulo_datos = QLabel("Datos Personales")
        self.lbl_titulo_datos.setStyleSheet("font-size: 16px; font-weight: bold;")
        datos_layout.addWidget(self.lbl_titulo_datos)

        self.grid_datos = QVBoxLayout()
        datos_layout.addLayout(self.grid_datos)

        self.content_layout.addWidget(self.datos_personales)

        # Sección de experiencias
        self.experiencias = QFrame()
        self.experiencias.setFrameStyle(QFrame.StyledPanel)
        exp_layout = QVBoxLayout(self.experiencias)

        self.lbl_titulo_exp = QLabel("Experiencia Laboral")
        self.lbl_titulo_exp.setStyleSheet("font-size: 16px; font-weight: bold;")
        exp_layout.addWidget(self.lbl_titulo_exp)

        self.experiencias_layout = QVBoxLayout()
        exp_layout.addLayout(self.experiencias_layout)

        self.content_layout.addWidget(self.experiencias)

        scroll.setWidget(self.content_widget)
        layout.addWidget(scroll)

        # Botones de acción
        buttons_layout = QHBoxLayout()
```

```
57            self.btn_editar = QPushButton("Editar")
58            self.btn_generar_cv = QPushButton("Generar CV")
59            self.btn_volver = QPushButton("Volver")
60
61            buttons_layout.addWidget(self.btn_editar)
62            buttons_layout.addWidget(self.btn_generar_cv)
63            buttons_layout.addWidget(self.btn_volver)
64
65            layout.addLayout(buttons_layout)
66
67            self.setLayout(layout)
68
69        def mostrar_datos(self, datos_personales, experiencias):
70            """Muestra los datos del registro en los campos correspondientes"""
71            if not datos_personales:
72                raise ValueError("Datos personales no proporcionados.")
73            if experiencias is None:
74                experiencias = []
75
76            # Limpiar layouts previos
77            self._limpiar_layout(self.grid_datos)
78            self._limpiar_layout(self.experiencias_layout)
79
80            # Mostrar datos personales
81            campos = [
82                ('Código:', datos_personales.get('codigo', '')),
83                ('Apellidos y Nombres:', datos_personales.get('nombres', '')),
84                ('Profesión:', datos_personales.get('profesion', '')),
85                ('Área:', datos_personales.get('area', '')),
86                ('Cargo:', datos_personales.get('cargo', '')),
87                ('Fecha de Nacimiento:', datos_personales.get('fecha_nacimiento', '')),
88                ('Lugar de Nacimiento:', datos_personales.get('lugar_nacimiento', '')),
89                ('Registro CIP:', datos_personales.get('registro_cip', '')),
90                ('Residencia:', datos_personales.get('residencia', '')),
91                ('Teléfono:', datos_personales.get('telefono', '')),
92                ('Correo:', datos_personales.get('correo', ''))
93            ]
94
95            for label_text, valor in campos:
96                if valor:
97                    container = QWidget()
98                    layout = QHBoxLayout(container)
99                    label = QLabel(label_text)
100                   label.setMinimumWidth(150)
101                   label.setStyleSheet("font-weight: bold;")
102                   valor_label = QLabel(str(valor))
103                   layout.addWidget(label)
104                   layout.addWidget(valor_label)
105                   layout.addStretch()
106                   self.grid_datos.addWidget(container)
107
108           # Mostrar experiencias
109           for exp in experiencias:
110               exp_frame = QFrame()
111               exp_frame.setFrameStyle(QFrame.StyledPanel)
112               exp_layout = QVBoxLayout(exp_frame)
113
114               # Título de la experiencia
```

```
            titulo = QLabel(f"{exp.get('cargo', 'N/A')} en {exp.get('empresa', 'N/A')}")
            titulo.setStyleSheet("font-weight: bold; font-size: 14px;")
            exp_layout.addWidget(titulo)

            # Detalles de la experiencia
            detalles = [
                ('Obra:', exp.get('obra', 'N/A')),
                ('Periodo:', f"{exp.get('fecha_inicio', '')} - {exp.get('fecha_fin', 'Actualidad
                ('Tipo de Obra:', exp.get('tipo_obra', 'N/A')),
                ('Monto del Proyecto:', exp.get('monto_proyecto', 'N/A')),
                ('Propietario:', exp.get('propietario', 'N/A'))
            ]

            for label_text, valor in detalles:
                if valor:
                    container = QWidget()
                    layout = QHBoxLayout(container)
                    label = QLabel(label_text)
                    label.setMinimumWidth(120)
                    label.setStyleSheet("font-weight: bold;")
                    valor_label = QLabel(str(valor))
                    layout.addWidget(label)
                    layout.addWidget(valor_label)
                    layout.addStretch()
                    exp_layout.addWidget(container)

            # Funciones
            if exp.get('funciones'):
                funciones_label = QLabel("Funciones:")
                funciones_label.setStyleSheet("font-weight: bold;")
                exp_layout.addWidget(funciones_label)

                funciones = exp['funciones'].split('\n')
                for funcion in funciones:
                    if funcion.strip():
                        func_label = QLabel(f"• {funcion.strip()}")
                        exp_layout.addWidget(func_label)

            self.experiencias_layout.addWidget(exp_frame)

        # Agregar espacio al final
        self.experiencias_layout.addStretch()

    def _limpiar_layout(self, layout):
        while layout.count():
            item = layout.takeAt(0)
            if item.widget():
                item.widget().deleteLater()
            elif item.layout():
                self._limpiar_layout(item.layout())

```

views\experiencia_view.py

```
from PyQt5.QtWidgets import (QWidget, QVBoxLayout, QHBoxLayout, QLineEdit,
                             QLabel, QComboBox, QPushButton, QDateEdit,
                             QTextEdit, QDialog, QListWidget, QFileDialog,
                             QMessageBox, QGroupBox)
import os
from PyQt5.QtCore import pyqtSignal, Qt, QDate
from app.config import Config

class ExperienciaView(QDialog):
    guardado_success = pyqtSignal(dict)

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Agregar Experiencia Laboral")
        self.setup_ui()

    def setup_ui(self):
        layout = QVBoxLayout()

        # Campos de la experiencia
        self.empresa_input = QLineEdit()
        self.obra_input = QLineEdit()
        self.tipo_obra_combo = QComboBox()
        self.tipo_obra_combo.addItems(Config.TIPOS_OBRA)
        self.cargo_input = QLineEdit()

        self.fecha_inicio = QDateEdit()
        self.fecha_inicio.setCalendarPopup(True)
        self.fecha_fin = QDateEdit()
        self.fecha_fin.setCalendarPopup(True)

        self.detalle_obra = QTextEdit()
        self.propietario = QLineEdit()

        # Layout para monto con selector de moneda
        monto_container = QWidget()
        monto_layout = QHBoxLayout(monto_container)
        monto_layout.setContentsMargins(0, 0, 0, 0)

        self.moneda_combo = QComboBox()
        self.moneda_combo.addItems(['S/', '$'])
        self.monto_input = QLineEdit()
        self.monto_input.setPlaceholderText("0.00")
        self.monto_input.textChanged.connect(self.validar_monto)

        monto_layout.addWidget(self.moneda_combo)
        monto_layout.addWidget(self.monto_input)

        # Agregar campos al layout
        campos = [
            ('Empresa:', self.empresa_input),
            ('Obra:', self.obra_input),
            ('Tipo de Obra:', self.tipo_obra_combo),
            ('Cargo:', self.cargo_input),
            ('Fecha de Inicio:', self.fecha_inicio),
            ('Fecha de Fin:', self.fecha_fin),
```

```
            ('Detalle de la Obra:', self.detalle_obra),
            ('Monto del Proyecto:', monto_container),
            ('Propietario:', self.propietario)
        ]

        for label_text, widget in campos:
            container = QWidget()
            field_layout = QHBoxLayout(container)
            label = QLabel(label_text)
            label.setMinimumWidth(120)
            field_layout.addWidget(label)
            field_layout.addWidget(widget)
            layout.addWidget(container)

        # Sección de funciones
        funciones_group = QGroupBox("Funciones")
        funciones_layout = QVBoxLayout()

        nueva_funcion_layout = QHBoxLayout()
        self.nueva_funcion_input = QLineEdit()
        self.btn_agregar_funcion = QPushButton("Agregar Función")
        self.btn_agregar_funcion.clicked.connect(self.agregar_funcion)
        nueva_funcion_layout.addWidget(self.nueva_funcion_input)
        nueva_funcion_layout.addWidget(self.btn_agregar_funcion)

        self.funciones_list = QListWidget()
        self.btn_eliminar_funcion = QPushButton("Eliminar Función")
        self.btn_eliminar_funcion.clicked.connect(self.eliminar_funcion)

        funciones_layout.addLayout(nueva_funcion_layout)
        funciones_layout.addWidget(self.funciones_list)
        funciones_layout.addWidget(self.btn_eliminar_funcion)
        funciones_group.setLayout(funciones_layout)
        layout.addWidget(funciones_group)

        # Sección de imágenes
        images_group = QGroupBox("Imágenes")
        images_layout = QVBoxLayout()

        self.images_list = QListWidget()
        self.btn_agregar_imagen = QPushButton("Agregar Imagen")
        self.btn_agregar_imagen.clicked.connect(self.adjuntar_imagen)
        self.btn_eliminar_imagen = QPushButton("Eliminar Imagen")
        self.btn_eliminar_imagen.clicked.connect(self.eliminar_imagen)
        self.btn_eliminar_imagen.setEnabled(False)

        images_layout.addWidget(self.images_list)
        images_layout.addWidget(self.btn_agregar_imagen)
        images_layout.addWidget(self.btn_eliminar_imagen)

        images_group.setLayout(images_layout)
        layout.addWidget(images_group)

        # Botones de acción
        buttons_layout = QHBoxLayout()
        self.btn_guardar = QPushButton("Guardar")
        self.btn_cancelar = QPushButton("Cancelar")
        self.btn_guardar.clicked.connect(self.guardar)
```

```
115             self.btn_cancelar.clicked.connect(self.reject)
116
117             buttons_layout.addWidget(self.btn_guardar)
118             buttons_layout.addWidget(self.btn_cancelar)
119             layout.addLayout(buttons_layout)
120
121             self.setLayout(layout)
122
123             # Conexiones para imágenes
124             self.images_list.itemSelectionChanged.connect(
125                 lambda: self.btn_eliminar_imagen.setEnabled(self.images_list.currentItem() is not No
126             )
127
128         def agregar_funcion(self):
129             funcion = self.nueva_funcion_input.text().strip()
130             if funcion:
131                 self.funciones_list.addItem(funcion)
132                 self.nueva_funcion_input.clear()
133
134         def eliminar_funcion(self):
135             current_item = self.funciones_list.currentItem()
136             if current_item:
137                 self.funciones_list.takeItem(self.funciones_list.row(current_item))
138
139         def validar_monto(self, texto):
140             texto_limpio = ''.join(c for c in texto if c.isdigit() or c == '.')
141             if texto_limpio.count('.') > 1:
142                 texto_limpio = texto_limpio[:texto_limpio.rfind('.')]
143             if '.' in texto_limpio:
144                 partes = texto_limpio.split('.')
145                 texto_limpio = f"{partes[0]}.{partes[1][:2]}"
146             if texto_limpio != texto:
147                 self.monto_input.setText(texto_limpio)
148
149
150         def adjuntar_imagen(self):
151             try:
152                 archivos, _ = QFileDialog.getOpenFileNames(
153                     self,
154                     "Seleccionar Imágenes",
155                     "",
156                     "Imágenes (*.png *.jpg *.jpeg *.bmp)"
157                 )
158
159                 if archivos:
160                     for archivo in archivos:
161                         nombre = os.path.basename(archivo)
162                         # Verificar si ya existe la imagen
163                         existe = False
164                         for i in range(self.images_list.count()):
165                             if self.images_list.item(i).text() == nombre:
166                                 existe = True
167                                 break
168
169                         if not existe:
170                             self.images_list.addItem(nombre)
171                             if not hasattr(self, 'imagenes'):
172                                 self.imagenes = []
```

```
                        self.imagenes.append({
                            'nombre': nombre,
                            'ruta': archivo,
                            'tipo': 'experiencia'
                        })
            except Exception as e:
                QMessageBox.critical(self, "Error", f"Error al adjuntar imagen: {str(e)}")

    def eliminar_imagen(self):
        current_row = self.images_list.currentRow()
        if current_row >= 0:
            self.images_list.takeItem(current_row)
            if hasattr(self, 'imagenes'):
                self.imagenes.pop(current_row)

    def guardar(self):
        datos = self.obtener_datos()
        self.guardado_success.emit(datos)
        self.accept()

    def obtener_datos(self):
        funciones = []
        for i in range(self.funciones_list.count()):
            funciones.append(self.funciones_list.item(i).text())

        documentos = []
        for i in range(self.images_list.count()):
            nombre = self.images_list.item(i).text()
            # Buscar la ruta en las imágenes guardadas
            for img in getattr(self, 'imagenes', []):
                if img['nombre'] == nombre:
                    documentos.append({
                        'nombre': img['nombre'],
                        'ruta': img['ruta'],
                        'tipo': img.get('tipo', 'experiencia')
                    })
                    break

        return {
            'empresa': self.empresa_input.text().strip(),
            'obra': self.obra_input.text().strip(),
            'tipo_obra': self.tipo_obra_combo.currentText(),
            'cargo': self.cargo_input.text().strip(),
            'fecha_inicio': self.fecha_inicio.date().toString('dd/MM/yyyy'),
            'fecha_fin': self.fecha_fin.date().toString('dd/MM/yyyy'),
            'detalle_obra': self.detalle_obra.toPlainText().strip(),
            'monto_proyecto': f"{self.moneda_combo.currentText()} {self.monto_input.text()}",
            'propietario': self.propietario.text().strip(),
            'funciones': funciones,
            'documentos': documentos  # Usar la lista de documentos procesada
        }

    def cargar_datos(self, experiencia):
        try:
            self.empresa_input.setText(experiencia['empresa'])
            self.obra_input.setText(experiencia['obra'])

            index = self.tipo_obra_combo.findText(experiencia['tipo_obra'])
```

```
            if index >= 0:
                self.tipo_obra_combo.setCurrentIndex(index)

            self.cargo_input.setText(experiencia['cargo'])
            self.fecha_inicio.setDate(QDate.fromString(experiencia['fecha_inicio'], 'dd-MM-yyyy'
            if experiencia['fecha_fin']:
                self.fecha_fin.setDate(QDate.fromString(experiencia['fecha_fin'], 'dd-MM-yyyy'))

            self.detalle_obra.setPlainText(experiencia['detalle_obra'])

            # Separar monto y moneda
            monto = experiencia['monto_proyecto']
            if monto.startswith('S/'):
                self.moneda_combo.setCurrentText('S/')
                self.monto_input.setText(monto[2:].strip())
            elif monto.startswith('$'):
                self.moneda_combo.setCurrentText('$')
                self.monto_input.setText(monto[1:].strip())
            else:
                self.monto_input.setText(monto)

            self.propietario.setText(experiencia['propietario'])

            # Cargar funciones
            self.funciones_list.clear()
            if isinstance(experiencia['funciones'], list):
                for funcion in experiencia['funciones']:
                    self.funciones_list.addItem(funcion)
            elif isinstance(experiencia['funciones'], str):
                for funcion in experiencia['funciones'].split(';'):
                    if funcion.strip():
                        self.funciones_list.addItem(funcion.strip())

            # Cargar imágenes
            self.images_list.clear()
            if 'documentos' in experiencia and experiencia['documentos']:
                self.imagenes = experiencia['documentos']  # Guardamos la lista completa de imág
                for doc in experiencia['documentos']:
                    self.images_list.addItem(doc['nombre'])

        except Exception as e:
            QMessageBox.critical(self, "Error", f"Error al cargar experiencia: {str(e)}")
```

views\login_view.py

```
# app/views/login_view.py
from PyQt5.QtWidgets import (QWidget, QVBoxLayout, QHBoxLayout, QLineEdit,
                             QPushButton, QLabel)
from PyQt5.QtCore import pyqtSignal, Qt
from PyQt5.QtGui import QPixmap
from app.config import Config

class LoginView(QWidget):
    loginSuccess = pyqtSignal()

    def __init__(self):
        super().__init__()
        self.setWindowTitle(f"{Config.APP_NAME} - Login")
        self.setup_ui()

    def setup_ui(self):
        layout = QVBoxLayout()

        # Logo
        logo_label = QLabel()
        pixmap = QPixmap(Config.LOGO_PATH)
        logo_label.setPixmap(pixmap.scaled(200, 200))
        logo_label.setAlignment(Qt.AlignCenter)

        # Título
        title = QLabel(Config.APP_NAME)
        title.setStyleSheet("font-size: 24px; font-weight: bold;")
        title.setAlignment(Qt.AlignCenter)

        # Campos de login
        self.usuario_input = QLineEdit()
        self.usuario_input.setPlaceholderText("Usuario")
        self.password_input = QLineEdit()
        self.password_input.setPlaceholderText("Contraseña")
        self.password_input.setEchoMode(QLineEdit.Password)

        # Botón de login
        self.login_btn = QPushButton("Iniciar Sesión")

        layout.addWidget(logo_label)
        layout.addWidget(title)
        layout.addWidget(self.usuario_input)
        layout.addWidget(self.password_input)
        layout.addWidget(self.login_btn)

        self.setLayout(layout)

        # Estilo
        self.setStyleSheet("""
            QWidget {
                background-color: white;
            }
            QPushButton {
                background-color: #4CAF50;
                color: white;
                padding: 8px;
```

```
                border-radius: 4px;
            }
            QLineEdit {
                padding: 8px;
                border: 1px solid #ccc;
                border-radius: 4px;
            }
        """)
```

views\main_view.py

```
# views/main_view.py
from PyQt5.QtWidgets import (
    QMainWindow, QWidget, QTabWidget, QVBoxLayout, QHBoxLayout,
    QLabel, QLineEdit, QComboBox, QDateEdit, QCheckBox,
    QPushButton, QGroupBox, QScrollArea, QTableWidget,
    QTableWidgetItem, QFileDialog, QMessageBox
)
from PyQt5.QtCore import Qt, QDate
from app.views.widgets import ExperienciaWidget
from app.models.persona import TipoDocumento
from app.config import Config

class MainView(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Sistema de Gestión de CVs")
        self.setMinimumSize(1200, 800)
        # Eliminar la línea: self.setup_registro_tab()
        self.setup_ui()

    def setup_ui(self):
        central_widget = QWidget()
        self.setCentralWidget(central_widget)
        layout = QVBoxLayout(central_widget)

        # Crear tabs
        self.tab_widget = QTabWidget()
        self.tab_busqueda = self.crear_tab_busqueda()
        self.tab_registro = self.crear_tab_registro()

        self.tab_widget.addTab(self.tab_busqueda, "Búsqueda")
        self.tab_widget.addTab(self.tab_registro, "Registro/Edición")

        layout.addWidget(self.tab_widget)

    def crear_tab_registro(self):
        tab = QWidget()
        layout = QVBoxLayout(tab)
        scroll = QScrollArea()
        scroll.setWidgetResizable(True)
        content = QWidget()
        content_layout = QVBoxLayout(content)

        # Grupo de Identificación
        grupo_id = QGroupBox("Identificación")
        grupo_id_layout = QHBoxLayout()

        self.tipo_doc_combo = QComboBox()
        self.tipo_doc_combo.addItems([t.value for t in TipoDocumento])
        self.num_doc_input = QLineEdit()
        self.codigo_input = QLineEdit()
        self.codigo_input.setReadOnly(True)

        grupo_id_layout.addWidget(QLabel("Tipo Documento:"))
        grupo_id_layout.addWidget(self.tipo_doc_combo)
        grupo_id_layout.addWidget(QLabel("Número:"))
```

```
            grupo_id_layout.addWidget(self.num_doc_input)
            grupo_id_layout.addWidget(QLabel("Código:"))
            grupo_id_layout.addWidget(self.codigo_input)
            grupo_id.setLayout(grupo_id_layout)

            # Grupo de Datos Personales
            grupo_personal = QGroupBox("Datos Personales")
            grupo_personal_layout = QVBoxLayout()

            campos_personales = [
                ('Nombres:', QLineEdit()),
                ('Apellidos:', QLineEdit()),
                ('Fecha Nacimiento:', QDateEdit()),
                ('Lugar Nacimiento:', QLineEdit()),
                ('Nacionalidad:', QLineEdit()),
            ]

            for label, widget in campos_personales:
                fila = QHBoxLayout()
                fila.addWidget(QLabel(label))
                fila.addWidget(widget)
                grupo_personal_layout.addLayout(fila)

            grupo_personal.setLayout(grupo_personal_layout)

            # Grupo de Contacto
            grupo_contacto = QGroupBox("Datos de Contacto")
            grupo_contacto_layout = QVBoxLayout()

            campos_contacto = [
                ('Teléfono:', QLineEdit()),
                ('Correo:', QLineEdit()),
                ('Dirección:', QLineEdit()),
                ('Ciudad:', QLineEdit()),
                ('País:', QLineEdit('Perú')),
            ]

            for label, widget in campos_contacto:
                fila = QHBoxLayout()
                fila.addWidget(QLabel(label))
                fila.addWidget(widget)
                grupo_contacto_layout.addLayout(fila)

            grupo_contacto.setLayout(grupo_contacto_layout)

            # Grupo Profesional
            grupo_prof = QGroupBox("Datos Profesionales")
            grupo_prof_layout = QVBoxLayout()

            self.area_combo = QComboBox()
            self.area_combo.addItems(Config.AREAS)

            campos_prof = [
                ('Profesión:', QLineEdit()),
                ('Nº Colegiatura:', QLineEdit()),
                ('Área:', self.area_combo),
                ('Cargo Actual:', QLineEdit()),
            ]
```

```
115
116             for label, widget in campos_prof:
117                 fila = QHBoxLayout()
118                 fila.addWidget(QLabel(label))
119                 fila.addWidget(widget)
120                 grupo_prof_layout.addLayout(fila)
121
122             self.check_eimsa = QCheckBox("¿Ha trabajado en EIMSA?")
123             grupo_prof_layout.addWidget(self.check_eimsa)
124             grupo_prof.setLayout(grupo_prof_layout)
125
126             # Grupo de Experiencias
127             grupo_exp = QGroupBox("Experiencias Laborales")
128             tab.experiencias_layout = QVBoxLayout()  # Guardar referencia al layout
129             tab.btn_agregar_experiencia = QPushButton("Agregar Experiencia")  # Crear botón como atr
130             tab.btn_agregar_experiencia.setStyleSheet("""
131                 QPushButton {
132                     background-color: #4CAF50;
133                     color: white;
134                     padding: 8px;
135                     border-radius: 4px;
136                 }
137                 QPushButton:hover {
138                     background-color: #45a049;
139                 }
140             """)
141
142             # Crear el layout de experiencias
143             experiencias_container = QVBoxLayout()
144             experiencias_container.addWidget(tab.btn_agregar_experiencia)
145             tab.experiencias_layout = QVBoxLayout()
146             experiencias_container.addLayout(tab.experiencias_layout)
147             grupo_exp.setLayout(experiencias_container)
148
149             # Botones de acción
150             botones_layout = QHBoxLayout()
151             tab.btn_guardar = QPushButton("Guardar")
152             tab.btn_cancelar = QPushButton("Cancelar")
153             botones_layout.addWidget(tab.btn_guardar)
154             botones_layout.addWidget(tab.btn_cancelar)
155
156             # Agregar todos los grupos al layout principal
157             content_layout.addWidget(grupo_exp)
158             content_layout.addLayout(botones_layout)
159
160             scroll.setWidget(content)
161             layout.addWidget(scroll)
162             return tab
163
164         def crear_tab_busqueda(self):
165             tab = QWidget()
166             layout = QVBoxLayout(tab)
167
168             # Filtros
169             grupo_filtros = QGroupBox("Filtros de Búsqueda")
170             filtros_layout = QVBoxLayout()
171
172             # Primera fila de filtros
```

```
173             fila1 = QHBoxLayout()
174             self.busqueda_input = QLineEdit()
175             self.busqueda_input.setPlaceholderText("Buscar por nombre, documento o código...")
176             self.area_filtro = QComboBox()
177             self.area_filtro.addItems(['Todas las Áreas'] + Config.AREAS)
178 
179             fila1.addWidget(QLabel("Búsqueda:"))
180             fila1.addWidget(self.busqueda_input)
181             fila1.addWidget(QLabel("Área:"))
182             fila1.addWidget(self.area_filtro)
183 
184             # Segunda fila de filtros
185             fila2 = QHBoxLayout()
186             self.tipo_obra_filtro = QComboBox()
187             self.tipo_obra_filtro.addItems(['Todos'] + Config.TIPOS_OBRA)
188             self.cargo_filtro = QLineEdit()
189             self.cargo_filtro.setPlaceholderText("Filtrar por cargo...")
190 
191             fila2.addWidget(QLabel("Tipo de Obra:"))
192             fila2.addWidget(self.tipo_obra_filtro)
193             fila2.addWidget(QLabel("Cargo:"))
194             fila2.addWidget(self.cargo_filtro)
195 
196             # Tercera fila de filtros
197             fila3 = QHBoxLayout()
198             self.check_eimsa_filtro = QCheckBox("Ex-empleados EIMSA")
199             self.años_exp_filtro = QComboBox()
200             self.años_exp_filtro.addItems(['Cualquier experiencia', '0-2 años', '2-5 años', '5-10 añ
201 
202             fila3.addWidget(self.check_eimsa_filtro)
203             fila3.addWidget(QLabel("Años de Experiencia:"))
204             fila3.addWidget(self.años_exp_filtro)
205 
206             # Botones de filtro
207             botones_filtro = QHBoxLayout()
208             self.btn_buscar = QPushButton("Buscar")
209             self.btn_limpiar = QPushButton("Limpiar Filtros")
210 
211             botones_filtro.addWidget(self.btn_buscar)
212             botones_filtro.addWidget(self.btn_limpiar)
213 
214             # Agregar todas las filas al grupo de filtros
215             filtros_layout.addLayout(fila1)
216             filtros_layout.addLayout(fila2)
217             filtros_layout.addLayout(fila3)
218             filtros_layout.addLayout(botones_filtro)
219             grupo_filtros.setLayout(filtros_layout)
220 
221             # Tabla de resultados
222             self.tabla_resultados = QTableWidget()
223             self.tabla_resultados.setColumnCount(10)
224             self.tabla_resultados.setHorizontalHeaderLabels([
225                 "Código", "Documento", "Nombres", "Profesión", "Área",
226                 "Cargo", "Años Exp.", "Ciudad", "Ex-EIMSA", "Última Act."
227             ])
228 
229             # Botones de acción
230             acciones = QHBoxLayout()
```

```
231             self.btn_ver = QPushButton("Ver Detalle")
232             self.btn_editar = QPushButton("Editar")
233             self.btn_generar_cv = QPushButton("Generar CV")
234 
235             acciones.addWidget(self.btn_ver)
236             acciones.addWidget(self.btn_editar)
237             acciones.addWidget(self.btn_generar_cv)
238 
239             # Agregar todo al layout principal
240             layout.addWidget(grupo_filtros)
241             layout.addWidget(self.tabla_resultados)
242             layout.addLayout(acciones)
243 
244             return tab
245 
246         def crear_tab_busqueda(self):
247             tab = QWidget()
248             layout = QVBoxLayout(tab)
249 
250             # Filtros
251             grupo_filtros = QGroupBox("Filtros de Búsqueda")
252             filtros_layout = QVBoxLayout()
253 
254             # Primera fila de filtros
255             fila1 = QHBoxLayout()
256             tab.busqueda_input = QLineEdit()
257             tab.busqueda_input.setPlaceholderText("Buscar por nombre, documento o código...")
258             tab.area_filtro = QComboBox()
259             tab.area_filtro.addItems(['Todas las Áreas'] + Config.AREAS)
260 
261             fila1.addWidget(QLabel("Búsqueda:"))
262             fila1.addWidget(tab.busqueda_input)
263             fila1.addWidget(QLabel("Área:"))
264             fila1.addWidget(tab.area_filtro)
265 
266             # Segunda fila de filtros
267             fila2 = QHBoxLayout()
268             tab.tipo_obra_filtro = QComboBox()
269             tab.tipo_obra_filtro.addItems(['Todos'] + Config.TIPOS_OBRA)
270             tab.cargo_filtro = QLineEdit()
271             tab.cargo_filtro.setPlaceholderText("Filtrar por cargo...")
272 
273             fila2.addWidget(QLabel("Tipo de Obra:"))
274             fila2.addWidget(tab.tipo_obra_filtro)
275             fila2.addWidget(QLabel("Cargo:"))
276             fila2.addWidget(tab.cargo_filtro)
277 
278             # Tercera fila de filtros
279             fila3 = QHBoxLayout()
280             tab.check_eimsa_filtro = QCheckBox("Ex-empleados EIMSA")
281             tab.años_exp_filtro = QComboBox()
282             tab.años_exp_filtro.addItems(['Cualquier experiencia', '0-2 años', '2-5 años', '5-10 año
283 
284             fila3.addWidget(tab.check_eimsa_filtro)
285             fila3.addWidget(QLabel("Años de Experiencia:"))
286             fila3.addWidget(tab.años_exp_filtro)
287 
288             # Botones de filtro
```

```
        botones_filtro = QHBoxLayout()
        tab.btn_buscar = QPushButton("Buscar")
        tab.btn_limpiar = QPushButton("Limpiar Filtros")

        botones_filtro.addWidget(tab.btn_buscar)
        botones_filtro.addWidget(tab.btn_limpiar)

        # Agregar todas las filas al grupo de filtros
        filtros_layout.addLayout(fila1)
        filtros_layout.addLayout(fila2)
        filtros_layout.addLayout(fila3)
        filtros_layout.addLayout(botones_filtro)
        grupo_filtros.setLayout(filtros_layout)

        # Tabla de resultados
        tab.tabla_resultados = QTableWidget()
        tab.tabla_resultados.setColumnCount(10)
        tab.tabla_resultados.setHorizontalHeaderLabels([
            "Código", "Documento", "Nombres", "Profesión", "Área",
            "Cargo", "Años Exp.", "Ciudad", "Ex-EIMSA", "Última Act."
        ])

        # Botones de acción
        acciones = QHBoxLayout()
        tab.btn_ver = QPushButton("Ver Detalle")
        tab.btn_editar = QPushButton("Editar")
        tab.btn_generar_cv = QPushButton("Generar CV")

        acciones.addWidget(tab.btn_ver)
        acciones.addWidget(tab.btn_editar)
        acciones.addWidget(tab.btn_generar_cv)

        # Agregar todo al layout principal
        layout.addWidget(grupo_filtros)
        layout.addWidget(tab.tabla_resultados)
        layout.addLayout(acciones)

        return tab
```

views\personal_view.py

```
# app/views/personal_view.py
from PyQt5.QtWidgets import (QWidget, QVBoxLayout, QHBoxLayout, QLineEdit,
                             QLabel, QComboBox, QPushButton, QDateEdit,
                             QMainWindow, QScrollArea, QMessageBox)
from PyQt5.QtCore import pyqtSignal, Qt, QDate
from app.config import Config
from app.utils.validators import validate_email, validate_phone, validate_date
from .experiencia_view import ExperienciaView
from .widgets.experiencia_widget import ExperienciaWidget
import re

class RegistroPersonalView(QWidget):
    # Definir señales
    guardado_success = pyqtSignal()
    experiencia_agregada = pyqtSignal(dict)
    experiencia_eliminada = pyqtSignal(int)
    guardado_success = pyqtSignal(dict)  # Modificado para enviar los datos

    def __init__(self):
        super().__init__()
        self.experiencias = []
        self.setup_ui()

    def validar_datos(self):
        """
        Valida los campos del formulario antes de guardar
        """
        try:
            # Validar campos requeridos
            if not self.nombres_input.text().strip():
                QMessageBox.warning(self, "Error", "El campo Apellidos y Nombres es requerido")
                self.nombres_input.setFocus()
                return False

            if not self.profesion_input.text().strip():
                QMessageBox.warning(self, "Error", "El campo Profesión es requerido")
                self.profesion_input.setFocus()
                return False

            if not self.cargo_input.text().strip():
                QMessageBox.warning(self, "Error", "El campo Cargo es requerido")
                self.cargo_input.setFocus()
                return False

            # Validar formato de correo si se ha ingresado
            correo = self.correo_input.text().strip()
            if correo and not self._validar_formato_correo(correo):
                QMessageBox.warning(self, "Error", "El formato del correo electrónico no es váli
                self.correo_input.setFocus()
                return False

            # Validar formato de teléfono si se ha ingresado
            telefono = self.telefono_input.text().strip()
            if telefono and not self._validar_formato_telefono(telefono):
                QMessageBox.warning(self, "Error", "El formato del teléfono no es válido")
                self.telefono_input.setFocus()
```

```
                return False

            # Si todas las validaciones pasan, emitir señal con los datos
            datos = self.obtener_datos()
            self.guardado_success.emit(datos)
            return True

        except Exception as e:
            QMessageBox.critical(self, "Error", f"Error al validar datos: {str(e)}")
            return False

    def _validar_formato_correo(self, correo):
        """
        Valida el formato del correo electrónico
        """
        patron = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
        return bool(re.match(patron, correo))

    def _validar_formato_telefono(self, telefono):
        """
        Valida el formato del teléfono
        """
        # Acepta números con o sin código de país, mínimo 9 dígitos
        patron = r'^\+?[0-9]{9,15}$'
        return bool(re.match(patron, telefono))

    def obtener_datos(self):
        """
        Recopila todos los datos del formulario
        """
        return {
            'codigo': self.codigo_input.text().strip(),
            'nombres': self.nombres_input.text().strip(),
            'profesion': self.profesion_input.text().strip(),
            'fecha_nacimiento': self.fecha_nac_input.date().toString('dd-MM-yyyy'),
            'lugar_nacimiento': self.lugar_nac_input.text().strip(),
            'registro_cip': self.registro_cip_input.text().strip(),
            'area': self.area_combo.currentText(),
            'cargo': self.cargo_input.text().strip(),
            'residencia': self.residencia_input.text().strip(),
            'telefono': self.telefono_input.text().strip(),
            'correo': self.correo_input.text().strip(),
            'experiencias': self.experiencias
        }

    def agregar_experiencia(self):
        """
        Abre el diálogo para agregar una nueva experiencia laboral
        """
        try:
            dialogo = ExperienciaView(self)
            if dialogo.exec_():
                # Si el diálogo se cerró con Aceptar
                datos_experiencia = dialogo.obtener_datos()
                self.experiencias.append(datos_experiencia)
                self.actualizar_lista_experiencias()
        except Exception as e:
            QMessageBox.critical(
```

```
                self,
                "Error",
                f"Error al agregar experiencia: {str(e)}"
            )

    def actualizar_lista_experiencias(self):
        """
        Actualiza la visualización de la lista de experiencias
        """
        try:
            # Limpiar el layout de experiencias existente
            while self.experiencias_layout.count():
                item = self.experiencias_layout.takeAt(0)
                if item.widget():
                    item.widget().deleteLater()

            # Agregar las experiencias actualizadas
            for i, exp in enumerate(self.experiencias):
                exp_widget = QWidget()
                exp_layout = QVBoxLayout(exp_widget)

                # Mostrar información resumida de la experiencia
                titulo = QLabel(f"{exp['cargo']} en {exp['empresa']}")
                titulo.setStyleSheet("font-weight: bold;")
                periodo = QLabel(f"Periodo: {exp['fecha_inicio']} - {exp['fecha_fin'] or 'Actual

                exp_layout.addWidget(titulo)
                exp_layout.addWidget(periodo)

                # Botones de acción
                botones_layout = QHBoxLayout()
                btn_editar = QPushButton("Editar")
                btn_eliminar = QPushButton("Eliminar")

                btn_editar.clicked.connect(lambda checked, idx=i: self.editar_experiencia(idx))
                btn_eliminar.clicked.connect(lambda checked, idx=i: self.eliminar_experiencia(id

                botones_layout.addWidget(btn_editar)
                botones_layout.addWidget(btn_eliminar)
                exp_layout.addLayout(botones_layout)

                self.experiencias_layout.addWidget(exp_widget)
        except Exception as e:
            QMessageBox.critical(
                self,
                "Error",
                f"Error al actualizar lista de experiencias: {str(e)}"
            )

    def editar_experiencia(self, index):
        """
        Abre el diálogo para editar una experiencia existente
        """
        try:
            dialogo = ExperienciaView(self)
            dialogo.cargar_datos(self.experiencias[index])
            if dialogo.exec_():
                self.experiencias[index] = dialogo.obtener_datos()
```

```
                self.actualizar_lista_experiencias()
            except Exception as e:
                QMessageBox.critical(
                    self,
                    "Error",
                    f"Error al editar experiencia: {str(e)}"
                )

    def eliminar_experiencia(self, index):
        """
        Elimina una experiencia de la lista
        """
        try:
            respuesta = QMessageBox.question(
                self,
                "Confirmar eliminación",
                "¿Está seguro de que desea eliminar esta experiencia?",
                QMessageBox.Yes | QMessageBox.No
            )

            if respuesta == QMessageBox.Yes:
                self.experiencias.pop(index)
                self.actualizar_lista_experiencias()
        except Exception as e:
            QMessageBox.critical(
                self,
                "Error",
                f"Error al eliminar experiencia: {str(e)}"
            )

    def setup_validations(self):
        """Configurar validaciones de campos"""
        # Validación de correo electrónico
        self.correo_input.textChanged.connect(self.validar_correo)
        # Validación de teléfono
        self.telefono_input.textChanged.connect(self.validar_telefono)
        # No permitir fechas futuras
        self.fecha_nac_input.setMaximumDate(QDate.currentDate())

    def setup_ui(self):
        main_layout = QHBoxLayout()

        # Panel izquierdo: Datos personales
        left_panel = QWidget()
        left_layout = QVBoxLayout(left_panel)

        # Campos de datos personales
        self.codigo_input = QLineEdit()
        self.codigo_input.setReadOnly(True)
        self.nombres_input = QLineEdit()
        self.profesion_input = QLineEdit()
        self.fecha_nac_input = QDateEdit()
        self.fecha_nac_input.setCalendarPopup(True)
        self.lugar_nac_input = QLineEdit()
        self.registro_cip_input = QLineEdit()
        self.area_combo = QComboBox()
        self.area_combo.addItems(Config.AREAS)
        self.cargo_input = QLineEdit()
```

```
        self.residencia_input = QLineEdit()
        self.telefono_input = QLineEdit()
        self.correo_input = QLineEdit()

        # Agregar campos al layout izquierdo
        campos = [
            ('Código:', self.codigo_input),
            ('Apellidos y Nombres:', self.nombres_input),
            ('Profesión:', self.profesion_input),
            ('Fecha de Nacimiento:', self.fecha_nac_input),
            ('Lugar de Nacimiento:', self.lugar_nac_input),
            ('Registro CIP:', self.registro_cip_input),
            ('Área:', self.area_combo),
            ('Cargo:', self.cargo_input),
            ('Residencia:', self.residencia_input),
            ('Teléfono:', self.telefono_input),
            ('Correo:', self.correo_input)
        ]

        for label_text, widget in campos:
            container = QWidget()
            layout = QHBoxLayout(container)
            label = QLabel(label_text)
            label.setMinimumWidth(150)
            layout.addWidget(label)
            layout.addWidget(widget)
            left_layout.addWidget(container)

        # Crear layout para botones
        buttons_layout = QHBoxLayout()
        self.btn_guardar = QPushButton("Guardar")
        self.btn_cancelar = QPushButton("Cancelar")  # Nuevo botón
        self.btn_cancelar.clicked.connect(self.cancelar)  # Nueva conexión
        buttons_layout.addWidget(self.btn_guardar)
        buttons_layout.addWidget(self.btn_cancelar)
        left_layout.addLayout(buttons_layout)

        # Panel derecho: Experiencias laborales
        right_panel = QWidget()
        right_layout = QVBoxLayout(right_panel)

        # Título de experiencias
        titulo_exp = QLabel("Experiencias Laborales")
        titulo_exp.setStyleSheet("font-weight: bold; font-size: 14px;")
        right_layout.addWidget(titulo_exp)

        # Contenedor scrollable para experiencias
        scroll = QScrollArea()
        experiencias_container = QWidget()
        self.experiencias_layout = QVBoxLayout(experiencias_container)
        scroll.setWidget(experiencias_container)
        scroll.setWidgetResizable(True)
        right_layout.addWidget(scroll)

        # Botón agregar experiencia
        self.btn_agregar_experiencia = QPushButton("Agregar Experiencia")
        self.btn_agregar_experiencia.setStyleSheet("""
            QPushButton {
```

```
                background-color: #4CAF50;
                color: white;
                padding: 8px;
                border-radius: 4px;
            }
            QPushButton:hover {
                background-color: #45a049;
            }
        """)
        right_layout.addWidget(self.btn_agregar_experiencia)

        # Agregar paneles al layout principal
        main_layout.addWidget(left_panel, stretch=1)
        main_layout.addWidget(right_panel, stretch=1)

        self.setLayout(main_layout)

        # Conexiones
        self.btn_guardar.clicked.connect(self.validar_datos)
        self.btn_agregar_experiencia.clicked.connect(self.agregar_experiencia)

    def limpiar_campos(self):
        """Limpia todos los campos del formulario"""
        self.codigo_input.clear()
        self.nombres_input.clear()
        self.profesion_input.clear()
        self.fecha_nac_input.setDate(QDate.currentDate())
        self.lugar_nac_input.clear()
        self.registro_cip_input.clear()
        self.area_combo.setCurrentIndex(0)
        self.cargo_input.clear()
        self.residencia_input.clear()
        self.telefono_input.clear()
        self.correo_input.clear()

        # Limpiar experiencias
        self.experiencias = []
        while self.experiencias_layout.count():
            item = self.experiencias_layout.takeAt(0)
            if item.widget():
                item.widget().deleteLater()

    def cargar_datos(self, datos):
        """Carga los datos del registro en el formulario"""
        try:
            self.codigo_input.setText(datos['codigo'])
            self.nombres_input.setText(datos['nombres'])
            self.profesion_input.setText(datos['profesion'])
            self.lugar_nac_input.setText(datos['lugar_nacimiento'])
            self.registro_cip_input.setText(datos['registro_cip'])

            # Convertir la fecha de nacimiento a QDate
            fecha_nac = QDate.fromString(datos['fecha_nacimiento'], 'yyyy-MM-dd')
            self.fecha_nac_input.setDate(fecha_nac)

            # Seleccionar el área correcta
            index = self.area_combo.findText(datos['area'])
            if index >= 0:
```

```
                    self.area_combo.setCurrentIndex(index)

                self.cargo_input.setText(datos['cargo'])
                self.residencia_input.setText(datos['residencia'])
                self.telefono_input.setText(datos['telefono'])
                self.correo_input.setText(datos['correo'])

                # Cargar experiencias
                self.experiencias = datos.get('experiencias', [])
                self.actualizar_lista_experiencias()

            except Exception as e:
                QMessageBox.critical(self, "Error", f"Error al cargar datos: {str(e)}")

        def cancelar(self):
            """Cancela el registro y regresa a la vista de búsqueda"""
            # Subir hasta encontrar MainView
            parent = self.parent()
            while parent and not isinstance(parent, QMainWindow):
                parent = parent.parent()

            if parent:
                parent.stack.setCurrentWidget(parent.busqueda_view)
                self.limpiar_campos()


```

views\widgets__init__.py

```
1   # views/widgets/__init__.py
2   from .experiencia_widget import ExperienciaWidget
3
4   __all__ = ['ExperienciaWidget']
```

views\widgets\experiencia_widget.py

```
# views/widgets/experiencia_widget.py
from PyQt5.QtWidgets import (
    QWidget, QVBoxLayout, QHBoxLayout, QLabel,
    QLineEdit, QDateEdit, QTextEdit, QPushButton,
    QComboBox, QGroupBox, QFileDialog, QListWidget
)
from PyQt5.QtCore import Qt, QDate, pyqtSignal
from PyQt5.QtWidgets import QDialog

class ExperienciaWidget(QDialog):
    eliminado = pyqtSignal(QDialog)
    actualizado = pyqtSignal(dict)

    def __init__(self, datos=None, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Experiencia Laboral")
        self.setup_ui()
        if datos:
            self.cargar_datos(datos)

        # Agregar botones de aceptar/cancelar
        buttons_layout = QHBoxLayout()
        self.btn_aceptar = QPushButton("Aceptar")
        self.btn_cancelar = QPushButton("Cancelar")

        self.btn_aceptar.clicked.connect(self.accept)
        self.btn_cancelar.clicked.connect(self.reject)

        buttons_layout.addWidget(self.btn_aceptar)
        buttons_layout.addWidget(self.btn_cancelar)

        # Asegurarse de que el layout principal sea QVBoxLayout
        if not hasattr(self, 'layout'):
            self.setLayout(QVBoxLayout())
        self.layout().addLayout(buttons_layout)

    def setup_ui(self):
        layout = QVBoxLayout(self)
        self.setFrameStyle = lambda: None  # Para compatibilidad con QFrame

        # Crear grupo principal
        grupo = QGroupBox("Experiencia Laboral")
        grupo_layout = QVBoxLayout()

        # Primera fila: Empresa y Cargo
        fila1 = QHBoxLayout()
        self.empresa_input = QLineEdit()
        self.cargo_input = QLineEdit()

        fila1.addWidget(QLabel("Empresa:"))
        fila1.addWidget(self.empresa_input)
        fila1.addWidget(QLabel("Cargo:"))
        fila1.addWidget(self.cargo_input)

        # Segunda fila: Fechas
        fila2 = QHBoxLayout()
```

```
57            self.fecha_inicio = QDateEdit()
58            self.fecha_inicio.setCalendarPopup(True)
59            self.fecha_fin = QDateEdit()
60            self.fecha_fin.setCalendarPopup(True)
61
62            fila2.addWidget(QLabel("Fecha Inicio:"))
63            fila2.addWidget(self.fecha_inicio)
64            fila2.addWidget(QLabel("Fecha Fin:"))
65            fila2.addWidget(self.fecha_fin)
66
67            # Tercera fila: Obra y Tipo
68            fila3 = QHBoxLayout()
69            self.obra_input = QLineEdit()
70            self.tipo_obra_combo = QComboBox()
71            # Agregar tipos de obra desde Config
72            self.tipo_obra_combo.addItems(['Planta de Cemento', 'Minería', 'Hidroeléctrica'])
73
74            fila3.addWidget(QLabel("Obra:"))
75            fila3.addWidget(self.obra_input)
76            fila3.addWidget(QLabel("Tipo:"))
77            fila3.addWidget(self.tipo_obra_combo)
78
79            # Cuarta fila: Propietario y Monto
80            fila4 = QHBoxLayout()
81            self.propietario_input = QLineEdit()
82            self.monto_input = QLineEdit()
83            self.moneda_combo = QComboBox()
84            self.moneda_combo.addItems(['S/', 'USD'])
85
86            fila4.addWidget(QLabel("Propietario:"))
87            fila4.addWidget(self.propietario_input)
88            fila4.addWidget(QLabel("Monto:"))
89            fila4.addWidget(self.moneda_combo)
90            fila4.addWidget(self.monto_input)
91
92            # Detalle de la obra
93            self.detalle_obra = QTextEdit()
94            self.detalle_obra.setMaximumHeight(100)
95            detalle_container = QVBoxLayout()
96            detalle_container.addWidget(QLabel("Detalle de la Obra:"))
97            detalle_container.addWidget(self.detalle_obra)
98
99            # Funciones
100           funciones_grupo = QGroupBox("Funciones")
101           funciones_layout = QVBoxLayout()
102           self.funciones_list = QListWidget()
103           self.funcion_input = QLineEdit()
104           btn_agregar_funcion = QPushButton("Agregar Función")
105           btn_eliminar_funcion = QPushButton("Eliminar Función")
106
107           funciones_layout.addWidget(self.funcion_input)
108           funciones_layout.addWidget(btn_agregar_funcion)
109           funciones_layout.addWidget(self.funciones_list)
110           funciones_layout.addWidget(btn_eliminar_funcion)
111           funciones_grupo.setLayout(funciones_layout)
112
113           # Botones de acción
114           botones = QHBoxLayout()
```

```
        self.btn_eliminar = QPushButton("Eliminar Experiencia")
        botones.addWidget(self.btn_eliminar)

        # Conectar señales
        self.btn_eliminar.clicked.connect(lambda: self.eliminado.emit(self))
        btn_agregar_funcion.clicked.connect(self.agregar_funcion)
        btn_eliminar_funcion.clicked.connect(self.eliminar_funcion)

        # Agregar todo al layout principal
        grupo_layout.addLayout(fila1)
        grupo_layout.addLayout(fila2)
        grupo_layout.addLayout(fila3)
        grupo_layout.addLayout(fila4)
        grupo_layout.addLayout(detalle_container)
        grupo_layout.addWidget(funciones_grupo)
        grupo_layout.addLayout(botones)

        grupo.setLayout(grupo_layout)
        layout.addWidget(grupo)

    def agregar_funcion(self):
        texto = self.funcion_input.text().strip()
        if texto:
            self.funciones_list.addItem(texto)
            self.funcion_input.clear()
            self.actualizado.emit(self.obtener_datos())

    def eliminar_funcion(self):
        item = self.funciones_list.currentItem()
        if item:
            self.funciones_list.takeItem(self.funciones_list.row(item))
            self.actualizado.emit(self.obtener_datos())

    def obtener_datos(self):
        funciones = []
        for i in range(self.funciones_list.count()):
            funciones.append(self.funciones_list.item(i).text())

        return {
            'empresa': self.empresa_input.text(),
            'cargo': self.cargo_input.text(),
            'fecha_inicio': self.fecha_inicio.date().toString('dd/MM/yyyy'),
            'fecha_fin': self.fecha_fin.date().toString('dd/MM/yyyy'),
            'obra': self.obra_input.text(),
            'tipo_obra': self.tipo_obra_combo.currentText(),
            'propietario': self.propietario_input.text(),
            'monto_proyecto': f"{self.moneda_combo.currentText()} {self.monto_input.text()}",
            'detalle_obra': self.detalle_obra.toPlainText(),
            'funciones': funciones
        }

    def cargar_datos(self, datos):
        self.empresa_input.setText(datos.get('empresa', ''))
        self.cargo_input.setText(datos.get('cargo', ''))

        # Cargar fechas
        if 'fecha_inicio' in datos:
            self.fecha_inicio.setDate(QDate.fromString(datos['fecha_inicio'], 'dd/MM/yyyy'))
```

```
            if 'fecha_fin' in datos:
                self.fecha_fin.setDate(QDate.fromString(datos['fecha_fin'], 'dd/MM/yyyy'))

            self.obra_input.setText(datos.get('obra', ''))

            # Cargar tipo de obra
            tipo_obra = datos.get('tipo_obra', '')
            index = self.tipo_obra_combo.findText(tipo_obra)
            if index >= 0:
                self.tipo_obra_combo.setCurrentIndex(index)

            self.propietario_input.setText(datos.get('propietario', ''))

            # Cargar monto y moneda
            monto = datos.get('monto_proyecto', '')
            if monto:
                if monto.startswith('S/'):
                    self.moneda_combo.setCurrentText('S/')
                    self.monto_input.setText(monto[2:].strip())
                elif monto.startswith('USD'):
                    self.moneda_combo.setCurrentText('USD')
                    self.monto_input.setText(monto[3:].strip())

            self.detalle_obra.setPlainText(datos.get('detalle_obra', ''))

            # Cargar funciones
            self.funciones_list.clear()
            funciones = datos.get('funciones', [])
            if isinstance(funciones, str):
                funciones = funciones.split(';')
            for funcion in funciones:
                if funcion.strip():
                    self.funciones_list.addItem(funcion.strip())
```