Trabajo I
Paradigmas de la Programación

Nombre: Víctor Núñez
Profesor: Juan Calderón
Fecha: 11 de Abril del 2025

Introducción

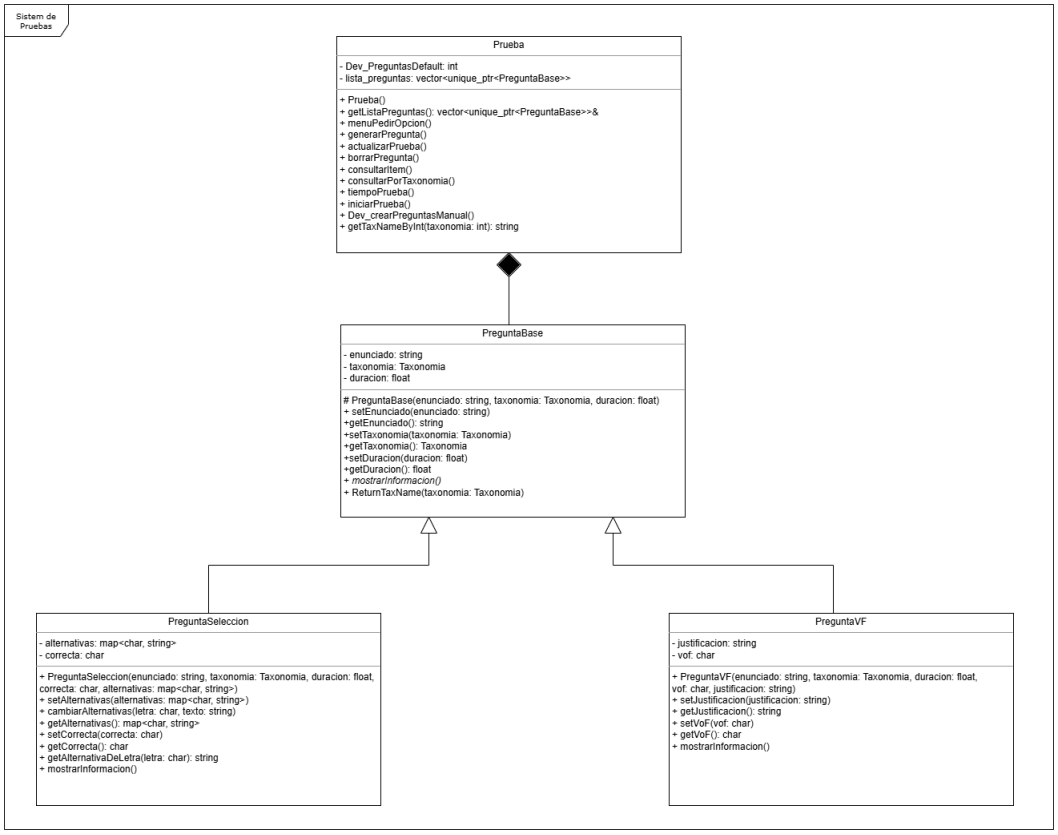
El presente informe detalla el desarrollo de una aplicación orientada a la creación, gestión y ejecución de preguntas evaluativas, implementada en el lenguaje de programación C++. Este trabajo se enmarca dentro del estudio de la programación orientada a objetos, abordando conceptos clave como la herencia, el polimorfismo, la agregación y el uso de punteros inteligentes (`unique_ptr`) para una gestión segura de la memoria.

El sistema desarrollado permite al usuario interactuar con una interfaz de consola para generar preguntas de selección múltiple o de verdadero/falso, almacenarlas, modificarlas, eliminarlas y consultarlas según distintos criterios, como su taxonomía o duración. Para ello, se diseñaron y relacionaron varias clases, destacando `PreguntaBase` como clase abstracta base, y sus derivadas `PreguntaSeleccion` y `PreguntaVF`, que encapsulan el comportamiento específico de cada tipo de pregunta. La clase `Prueba` actúa como núcleo del programa, gestionando las preguntas mediante un vector de punteros inteligentes a objetos de tipo `PreguntaBase`.

A lo largo del informe, se analizan en detalle las decisiones de diseño tomadas, se presentan fragmentos relevantes del código fuente y se incluye un diagrama UML que ilustra las relaciones entre las clases. El objetivo principal ha sido aplicar los conocimientos adquiridos en el área de diseño orientado a objetos para construir un sistema funcional, modular y extensible, respetando buenas prácticas de programación.

1. Desarollo de Solución

1.1. Diagrama UML



El diseño del programa sigue un enfoque orientado a objetos que permite organizar y reutilizar el código de manera eficiente y escalable. A continuación, se explica la relación entre las clases, tal como se refleja en el diagrama UML, y las decisiones de diseño tomadas para lograr un sistema flexible y bien estructurado.

- **PreguntaBase:** Es una clase abstracta que actúa como clase base para todos los tipos de preguntas. Contiene atributos y métodos comunes, como el enunciado, la taxonomía y la duracion estimada para responder. Además, incluye métodos getter y setter, así como un método virtual puro llamado `mostrarInformacion()`, lo que obliga a sus subclases a implementar su propia versión de este método. Esta abstracción promueve la reutilización de código y garantiza una interfaz uniforme para todas las preguntas.

- **PreguntaVF y PreguntaSeleccion:** Ambas clases heredan de `PreguntaBase` y representan tipos específicos de preguntas. La clase `PreguntaVF` incorpora los atributos `vof` (que indica si la respuesta es verdadera o falsa) y `justificacion` (utilizada para respuestas falsas). Por su parte, la clase `PreguntaSeleccion` añade un `map`(de `char` a `string`) de alternativas asociadas a letras, así como una letra que representa la respuesta correcta. Esta especialización permite adaptar el comportamiento de las preguntas según su tipo, sin afectar la estructura general.
- **Prueba:** Es la clase que maneja y contiene a las preguntas, la vida útil de las estas dependen de la existencia de Prueba, si desaparece, también desaparecen las preguntas, por esto es una composición en el diagrama UML. Además, es la encargada de moestrar el menú y gestionar el input del usuario. Dada su importancia dentro del programa, esta clase tendrá su propia sección explicando a más detalle su funcionamiento.

Todo este diseño permite una estructura clara, modular y extensible. Por ejemplo, para incorporar nuevos tipos de preguntas en el futuro, basta con crear una nueva clase que herede de `PreguntaBase` e implemente sus métodos virtuales. El uso de polimorfismo y encapsulamiento garantiza que la lógica del programa no se vea afectada, mientras que el uso de punteros inteligentes asegura una administración adecuada de los recursos.

En resumen, las relaciones entre clases se diseñaron para:

- Separar responsabilidades entre tipos de preguntas.
- Permitir extensibilidad sin modificar código existente.
- Facilitar la reutilización y mantenimiento del código.
- Utilizar características de C++ como `unique_ptr` y `dynamic_cast`.

1.2. Prueba

Como se mencionó anteriormente, la clase `Prueba` es la encargada de gestionar una colección de preguntas, las cuales se almacenan en un `std::vector` de `std::unique_ptr<PreguntaBase>`. El uso de `unique_ptr` garantiza una gestión segura y automática de la memoria, evitando fugas y simplificando la destrucción de objetos. Este tipo de punteros, llamados punteros inteligentes, permiten administrar la memoria de forma automática, evitando fugas y errores comunes relacionados con la manipulación manual de punteros. En particular, `unique_ptr` es un puntero inteligente que garantiza propiedad exclusiva: el objeto al que apunta solo puede ser gestionado por una instancia de `unique_ptr` a la vez. Cuando un `unique_ptr` se destruye (por ejemplo, al eliminarse del vector), libera automáticamente la memoria del objeto que contiene. Al almacenar punteros a la clase base, se puede aprovechar el **polimorfismo** para tratar distintas clases derivadas como objetos de tipo `PreguntaBase`, permitiendo recorrer, mostrar y actualizar preguntas sin conocer su tipo exacto.

A continuación, se empieza a describir el funcionamiento de `Prueba`, empezando por su constructor.

```
Prueba::Prueba()
{
    iniciarPrueba();
}
```

Este constructor tiene como propósito iniciar el flujo principal del programa, y para ello invoca a la función `iniciarPrueba()`. Dicha función contiene un ciclo `while` que se encarga de desplegar un menú en pantalla, el cual permite al usuario seleccionar diferentes opciones.

```
    while (opcion != 8)
    {

        opcion = menuPedirOpcion();
        std::cout << "\n";
        std::cout << separador << std::endl;
        switch (opcion)
        {
        case 1:
            generarPregunta();
            break;

        case 2:
            mostrarPreguntas();
            break;

        case 3:
            actualizarPrueba();
            break;

        case 4:
            borrarPregunta();
            break;

        case 5:
            consultarItem();
            break;

        case 6:
            consultarPorTaxonomia();
            break;

        case 7:
            std::cout << "El tiempo total de la prueba es de: " << tiempoPrueba() / 60
<< " minutos. \n";
            break;

```

```
        case 8:
            std::cout << "Terminando programa..." << std::endl;
            break;
        }
    }

```

Cada opción activa una funcionalidad distinta del sistema, según la entrada proporcionada por el usuario. Las distintas funcionalidades a las que se accede mediante el menú están implementadas en funciones cuyos nombres son autoexplicativos, lo que facilita la comprensión del código. No obstante, en este informe se profundizará únicamente en las funciones `generarPregunta()`, `borrarPregunta()` y `actualizarPrueba()`, por considerarlas representativas del funcionamiento general del sistema y clave para la gestión dinámica de las preguntas.[1]

- `generarPregunta()` es la función que permite al usuario crear nuevas preguntas de selección múltiple o de verdadero/falso.

```
    int tipo_pregunta;


    // Variables preguntas general.
    std::string enunciado;
    Taxonomia taxonomia;
    int taxonomia_int;
    float duracion;

    // Seleccion Multiple
    std::map<char, std::string> alternativas;
    std::string alternativa;
    char letras[] = { 'a', 'b', 'c', 'd', 'e' };
    char correcta;

    // VoF
    std::string justificacion;
    char vof;



    std::cout << "Escriba el tipo de pregunta.\n 1. Pregunta Seleccion Multiple. \n
    2. Pregunta Verdaro o Falso. \n" << std::endl;
    std::cout << "Ingrese aqui su opcion: ";
    std::cin >> tipo_pregunta;

```

[1]El código de las funciones completas no están en el informe, solo se muestran los segmentos relevantes.

```
    while (tipo_pregunta != 1 && tipo_pregunta != 2)
    {
        std::cout << "Opcion invalida. Intente de nuevo: ";
        std::cin >> tipo_pregunta;
    }

```

Esta función solicita al usuario que indique el tipo de pregunta a crear y, dependiendo de su elección, recopila la información correspondiente: enunciado, alternativas o justificación, nivel de taxonomía y duración. Se incluyen validaciones básicas, como evitar que el enunciado esté vacío, garantizando así la integridad de los datos ingresados.

```
    if (tipo_pregunta == 1)
    {
        this->lista_preguntas.push_back(std::make_unique<PreguntaSeleccion>(enunciado, static_cast<Taxonomia>(taxonomia_int), duracion, correcta, alternativas));
        return;
    }

    this->lista_preguntas.push_back(std::make_unique<PreguntaVF>(enunciado, static_cast<Taxonomia>(taxonomia_int), duracion, vof, justificacion));

```

Finalmente, la nueva pregunta es almacenada en el vector `lista_preguntas`, utilizando un `unique_ptr`, lo que asegura una gestión de memoria automática y segura.

- `borrarPregunta()` permite al usuario eliminar una pregunta previamente creada. Para ello, se muestra un listado numerado de todas las preguntas almacenadas, y el usuario debe ingresar el número correspondiente a la que desea eliminar. La función incluye una opción para cancelar la operación ingresando el número 0, así como validaciones para evitar intentos de acceso fuera de rango.

```
        void Prueba::borrarPregunta()
    {
        int numero_pregunta;

        while (true)
        {
            int valido = this->lista_preguntas.size();
            mostrarPreguntas();
            std::cout << "Ingrese el numero de la pregunta que desea borrar (ingrese 0 para volver al menu): ";
            std::cin.ignore();
            std::cin >> numero_pregunta;

```

```
            if (numero_pregunta == 0)
            {
                return;
            }

            if (numero_pregunta > valido)
            {
                std::cout << separador << std::endl;
                std::cout << "Numero de pregunta fuera de rango, intente de nuevo"
<< std::endl;
                continue;
            }

            this->lista_preguntas.erase(this->lista_preguntas.begin() + (
numero_pregunta - 1));
            std::cout << "Pregunta " << numero_pregunta << " borrada con exito! \n"
;

        }
    }

```

Al eliminarse una pregunta, el puntero correspondiente es removido del vector `lista_preguntas`, liberando automáticamente la memoria ocupada gracias al uso de `unique_ptr`.

- `actualizarPrueba()` permite modificar los atributos de una pregunta previamente creada. Primero, se muestra un listado con todas las preguntas disponibles, y el usuario debe seleccionar una para editar. Dependiendo del tipo de pregunta (selección múltiple o verdadero/falso), se presentan distintas opciones de actualización, como modificar el enunciado, la taxonomía, la duración, las alternativas o la justificación.

```
    int numero;
    int actualizar;
    int valido = this->lista_preguntas.size();

    while (true)
    {
        mostrarPreguntas();
        std::cout << "Ingrese el numero de la pregunta cuyos datos desea actualizar
: ";
        //std::cin.ignore();
        std::cin >> numero;
        int indice_pregunta = numero - 1;

        if (numero == 0)
```

```
        {
            return;
        }

        // Abajo de == 0,sino da nullptr
        PreguntaSeleccion* pregunta_seleccion = dynamic_cast<PreguntaSeleccion*>(
    this->lista_preguntas[indice_pregunta].get());
        PreguntaVF* pregunta_vf = dynamic_cast<PreguntaVF*>(this->lista_preguntas[
    indice_pregunta].get());


        if (numero > valido)
        {
            std::cout << separador << std::endl;
            std::cout << "Numero de pregunta fuera de rango, intente de nuevo" <<
    std::endl;
            continue;
        }

        this->lista_preguntas[indice_pregunta]->mostrarInformacion();
        std::cout << "Ingrese un numero segun el dato que desea actualizar: ";
        std::cin.ignore();
        std::cin >> actualizar;


```

La función incluye validaciones para asegurar que los datos nuevos sean válidos, y emplea `dynamic_cast` para determinar el tipo concreto de pregunta a actualizar. Se utiliza `dynamic_cast` para convertir un puntero de la clase base `PreguntaBase` a uno de sus clases derivadas, como `PreguntaSeleccion` o `PreguntaVF`. Esto permite acceder a funcionalidades específicas de cada tipo de pregunta. Si la conversión no es válida, el puntero resultante será `nullptr`, lo que permite manejar correctamente el tipo de pregunta seleccionado sin producir errores.

La implementación de la clase `Prueba` y sus métodos representó una parte central del desarrollo, ya que articula el flujo principal del programa y gestiona la interacción con el usuario. El análisis detallado de sus funciones permitió evidenciar cómo se integran las distintas clases del sistema y de qué manera se aplican los conceptos de diseño orientado a objetos en un contexto práctico. Habiendo abordado esta estructura fundamental del programa, es pertinente presentar ahora algunas reflexiones finales sobre el proceso de desarrollo y los aprendizajes obtenidos durante su realización.

Conclusión

El desarrollo de este programa permitió alcanzar satisfactoriamente los objetivos propuestos, entre ellos la creación de una estructura flexible y escalable para gestionar distintos tipos de preguntas, utilizando principios de la programación orientada a objetos como la herencia, el polimorfismo y el uso de punteros inteligentes para una gestión segura de la memoria.

Durante el proceso se consolidaron conceptos clave del lenguaje C++, como la abstracción mediante clases base, el uso de constructores y métodos virtuales, y la implementación de interacción con el usuario por medio de menús. Asimismo, se incorporaron mecanismos de validación que permitieron mejorar la robustez del sistema y evitar errores comunes en la entrada de datos.

Como reflexión final, el trabajo representó un desafío enriquecedor, no solo en el plano técnico sino también en términos de planificación y diseño del código. Fue necesario tomar decisiones sobre la arquitectura del programa que facilitaran su extensión futura y aseguraran una separación clara de responsabilidades entre clases. En conjunto, la experiencia favoreció el desarrollo de buenas prácticas de programación y fomentó una comprensión más profunda del paradigma orientado a objetos.