

UNIVERSIDAD ANDRÉS BELLO

FACULTAD DE INGENIERÍA

Paradigmas de la Programación

# **Trabajo 1**

## **La taxonomía de Bloom**

Autores:

Raphaël Maufroy

José Salazar Cabello

Profesor: Juan Calderón Maureira

Fecha: Abril 2025

Año Académico: 2025, Semestre 1

## 0.1. Diagrama de Clases

A continuación, se presenta el diagrama de clases correspondiente a la solución planteada. Este diagrama permite visualizar de manera general las relaciones entre las distintas clases desarrolladas, sus atributos principales y los métodos más relevantes que posee cada una.

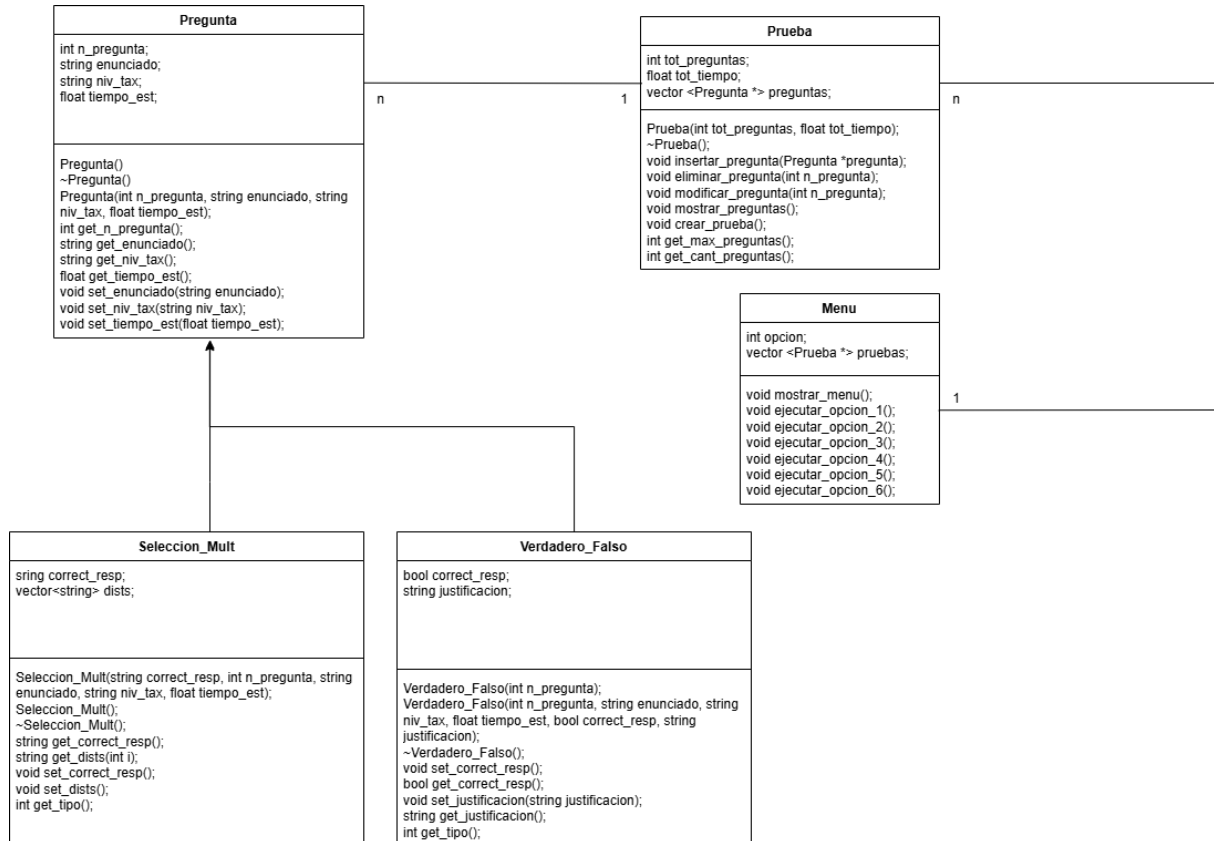


Figura 1: Diagrama de clases del sistema desarrollado.

# 1. Introducción

El presente trabajo tiene como objetivo desarrollar un sistema en C++ que permita a un usuario crear y gestionar pruebas escritas utilizando la Taxonomía de Bloom como referencia principal.

Dentro del desarrollo del trabajo se buscó aplicar conceptos fundamentales de programación orientada a objetos, como lo son la herencia, el polimorfismo y el manejo de memoria dinámica. Además, se consideró diseñar una solución modular y estructurada que permita un fácil mantenimiento y comprensión del código.

La funcionalidad central del sistema consiste en gestionar pruebas compuestas por distintos tipos de preguntas, tales como preguntas de Verdadero o Falso y preguntas de Selección Múltiple. Cada una de estas preguntas se encuentra asociada a un nivel taxonómico, lo que permite categorizar y evaluar las habilidades que se desean medir en el evaluado.

En este informe se presentará:

- La descripción detallada de la solución implementada
- Las decisiones de diseño tomadas en el desarrollo
- La estructura y funcionamiento del sistema
- Las conclusiones y reflexiones sobre el trabajo realizado

## 2. Descripción de la solución

La solución se estructura en cuatro clases principales:

### 2.1. Clase Pregunta

La clase base `Pregunta` define la estructura común para todos los tipos de preguntas:

```
1 class Pregunta {
2     private:
3         int n_pregunta;
4         string enunciado;
5         string niv_tax;
6         float tiempo_est;
7     public:
8         Pregunta(int n_pregunta, string enunciado, string niv_tax, float
          tiempo_est);
9         virtual ~Pregunta();
10        // Métodos getters y setters
11        virtual void set_correct_resp()=0;
12        virtual int get_tipo()=0;
13 };
```

Los atributos principales incluyen:

- `n_pregunta`: Identificador único de la pregunta
- `enunciado`: Texto de la pregunta
- `niv_tax`: Nivel taxonómico según Bloom

- tiempo\_est: Tiempo estimado para responder

## 2.2. Clases Derivadas

### 2.2.1. Clase Seleccion\_Mult

Esta clase implementa las preguntas de selección múltiple:

```

1 class Seleccion_Mult : public Pregunta {
2     private:
3         string correct_resp;
4         vector<string> dists;
5     public:
6         Seleccion_Mult(string correct_resp, int n_pregunta,
7                         string enunciado, string niv_tax,
8                         float tiempo_est);
9         ~Seleccion_Mult();
10        void set_correct_resp();
11        void set_dists();
12        int get_tipo() { return 2; }
13 };

```

### 2.2.2. Clase Verdadero\_Falso

Esta clase maneja las preguntas de verdadero o falso:

```

1 class Verdadero_Falso : public Pregunta {
2     private:
3         bool correct_resp;
4         string justificacion;
5     public:
6         Verdadero_Falso(int n_pregunta, string enunciado,
7                         string niv_tax, float tiempo_est,
8                         bool correct_resp, string justificacion);
9         ~Verdadero_Falso();
10        void set_correct_resp();
11        bool get_correct_resp();
12        string get_justificacion();
13        int get_tipo() { return 1; }
14 };

```

## 2.3. Clase Prueba

La clase Prueba gestiona un conjunto de preguntas:

```

1 class Prueba {
2     private:
3         int tot_preguntas;
4         float tot_tiempo;
5         vector<Pregunta *> preguntas;
6     public:
7         Prueba(int tot_preguntas, float tot_tiempo);
8         ~Prueba();
9         void insertar_pregunta(Pregunta *pregunta);
10        void eliminar_pregunta(int n_pregunta);
11        void modificar_pregunta(int n_pregunta);

```

```

12     void mostrar_preguntas();
13     int get_max_preguntas();
14     int get_cant_preguntas();
15 };

```

Un ejemplo de implementación de uno de sus métodos principales es:

```

1 void Prueba::modificar_pregunta(int n_pregunta) {
2     int tipo = this->preguntas[n_pregunta - 1]->get_tipo();
3     if (tipo == 1) { // Verdadero_Falso
4         string input;
5         cout << "Modificar el enunciado: " << endl;
6         cin.ignore();
7         getline(cin, input);
8         if(!input.empty()) {
9             this->preguntas[n_pregunta - 1]->set_enunciado(input);
10        }
11        // ... más código de modificación
12    }
13 }

```

## 2.4. Clase Menu

La clase Menu implementa la interfaz de usuario y la lógica principal del programa. Esta clase es responsable de la interacción con el usuario y la gestión de las pruebas:

```

1 class Menu {
2     private:
3         int opcion;
4         vector<Prueba *> pruebas;
5     public:
6         void mostrar_menu();
7         void ejecutar_opcion_1(); // Crear prueba
8         void ejecutar_opcion_2(); // Insertar pregunta
9         void ejecutar_opcion_3(); // Eliminar pregunta
10        void ejecutar_opcion_4(); // Modificar pregunta
11        void ejecutar_opcion_5(); // Mostrar preguntas
12        void ejecutar_opcion_6(); // Eliminar prueba
13 };

```

La implementación del menú sigue un patrón de diseño modular, donde cada opción está encapsulada en su propio método. Por ejemplo, la creación de una prueba:

```

1 void Menu::ejecutar_opcion_1() {
2     int tot_preguntas;
3     float tot_tiempo;
4     limpiar_pantalla();
5     cout << "Crear prueba" << endl;
6     cout << "Ingrese el numero de preguntas: ";
7     cin >> tot_preguntas;
8     cout << "Ingrese el tiempo total estimado: ";
9     cin >> tot_tiempo;
10    if (tot_preguntas > 0) {
11        if (tot_tiempo > 0) {
12            Prueba *prueba = new Prueba(tot_preguntas, tot_tiempo);
13            this->pruebas.push_back(prueba);
14        } else {

```

```

15     // Tiempo por defecto: 3 minutos por pregunta
16     Prueba *prueba = new Prueba(tot_preguntas, tot_preguntas * 3);
17     this->pruebas.push_back(prueba);
18 }
19 }
20 }

```

Las principales decisiones de diseño en la clase Menu incluyen:

1. **Gestión de Memoria Dinámica:** Utilizamos un vector de punteros a Prueba para mantener las pruebas creadas:

```

1     vector<Prueba *> pruebas;
2
3     // En ejecutar_opcion_6 (eliminar prueba):
4     delete this->pruebas[id_prueba-1];
5     this->pruebas.erase(this->pruebas.begin() + id_prueba - 1);

```

2. **Validación de Entradas:** Implementamos validaciones para asegurar la integridad de los datos:

```

1     if (id_prueba > 0 && id_prueba <= this->pruebas.size()) {
2         if (id_pregunta > 0 &&
3             id_pregunta <= this->pruebas[id_prueba-1]->
4                 get_cant_preguntas()) {
5             // Operación válida
6         }
7     }

```

3. **Interfaz de Usuario:** Diseñamos una interfaz clara y amigable:

- Mensajes informativos para cada operación
- Limpieza de pantalla entre operaciones
- Pausas para mejor legibilidad (`this_thread::sleep_for`)

4. **Modularidad:** Cada operación está encapsulada en su propio método, lo que facilita:

- Mantenimiento del código
- Depuración de errores
- Extensibilidad del sistema

## 2.5. Decisiones de Diseño

Las principales decisiones de diseño tomadas incluyen:

1. **Herencia y Polimorfismo:** Utilizamos una jerarquía de clases para manejar los diferentes tipos de preguntas. Por ejemplo, el método virtual `get_tipo()` nos permite identificar el tipo de pregunta sin necesidad de hacer casting:

```

1     virtual int get_tipo() = 0; // En clase base
2     int get_tipo() { return 1; } // En Verdadero_Falso
3     int get_tipo() { return 2; } // En Seleccion_Mult

```

2. **Gestión de Memoria Dinámica:** Implementamos un sistema de gestión de memoria usando punteros y vectores de la STL:

```
1     vector<Pregunta *> preguntas; // Vector de punteros
2     ~Prueba() {
3         for(auto p : preguntas) {
4             delete p; // Liberación de memoria
5         }
6     }
```

## 2.6. Manejo del Tiempo

En nuestro sistema, el manejo del tiempo se implementa en dos niveles:

1. **Tiempo por Pregunta:** Cada pregunta individual mantiene su propio tiempo estimado:

```
1     class Pregunta {
2     private:
3         float tiempo_est; // Tiempo estimado en minutos
4     public:
5         void set_tiempo_est(float tiempo_est);
6         float get_tiempo_est();
7     };
```

2. **Tiempo Total de la Prueba:** La clase Prueba mantiene un tiempo total:

```
1     class Prueba {
2     private:
3         float tot_tiempo; // Tiempo total en minutos
4     };
```

Observamos algunas limitaciones en la implementación actual:

- No hay una actualización automática del tiempo total cuando se modifican los tiempos individuales de las preguntas
- El tiempo total se establece al crear la prueba y no se ajusta al agregar o eliminar preguntas
- Por defecto, se asignan 3 minutos por pregunta si no se especifica un tiempo total:

```
1     // En Menu::ejecutar_opcion_1()
2     if (tot_tiempo <= 0) {
3         Prueba *prueba = new Prueba(tot_preguntas, tot_preguntas * 3)
4         ;
5     }
```

Posibles mejoras para futuras versiones:

- Implementar un método para recalcular el tiempo total basado en los tiempos individuales
- Agregar validaciones para asegurar que el tiempo total sea coherente con la suma de tiempos individuales
- Permitir ajustes dinámicos del tiempo total cuando se modifican las preguntas

### **3. Conclusión**

El desarrollo de este trabajo nos permitió implementar de manera práctica los principales conceptos de la programación orientada a objetos en C++. Logramos crear un sistema modular y estructurado que gestiona eficientemente pruebas y preguntas siguiendo la taxonomía de Bloom.

El uso de herencia y polimorfismo resultó fundamental para manejar los diferentes tipos de preguntas de manera uniforme, mientras que el uso de memoria dinámica nos permitió una gestión flexible de las pruebas y sus componentes.

La solución desarrollada demuestra la aplicabilidad de los conceptos de POO en problemas reales, resultando en un código mantenible, extensible y bien estructurado que cumple con todos los requerimientos establecidos.

#### **3.1. Cumplimiento de Objetivos**

Se alcanzaron satisfactoriamente los siguientes objetivos:

- Implementación de un sistema funcional para la gestión de pruebas
- Incorporación efectiva de los niveles taxonómicos de Bloom
- Desarrollo de una interfaz de usuario intuitiva
- Implementación de todas las operaciones CRUD requeridas

#### **3.2. Reflexión sobre el Trabajo**

El desarrollo de este proyecto permitió:

- Profundizar en la aplicación práctica de conceptos de POO
- Comprender la importancia de un buen diseño de software
- Desarrollar habilidades en la gestión de memoria dinámica
- Implementar soluciones modulares y extensibles