

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.

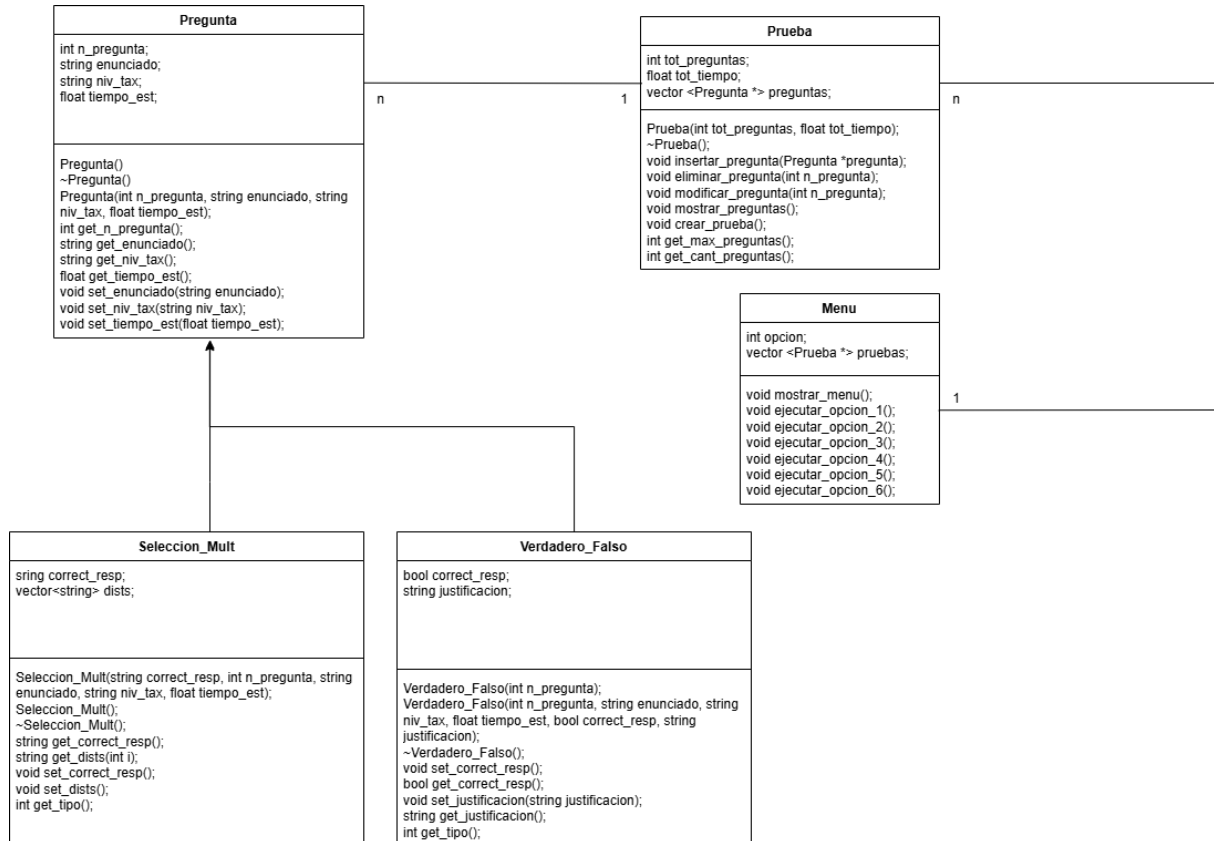


Figura 1: Diagrama de clases del sistema desarrollado.

# 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
9             tiempo_est);
10        virtual ~Pregunta();
11        // Métodos getters y setters
12        virtual void set_correct_resp()=0;
13        virtual int get_tipo();
14        void set_tiempo_est(float tiempo_est);
15        float get_tiempo_est();
16    };
17
```

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         string get_dists(int i);
13         int get_tipo() { return 2; }
14     };

```

### 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         void set_justificacion(string justificacion);
13         string get_justificacion();
14         int get_tipo() { return 1; }
15     };

```

## 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          void recalcular_tiempo_total();
7      public:

```

```

8     Prueba(int tot_preguntas, float tot_tiempo);
9     ~Prueba();
10    void insertar_pregunta(Pregunta *pregunta);
11    void eliminar_pregunta(int n_pregunta);
12    void modificar_pregunta(int n_pregunta);
13    void mostrar_preguntas();
14    int get_max_preguntas();
15    int get_cant_preguntas();
16    float get_tiempo_total() const { return tot_tiempo; }
17 };

```

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 tiempo estimado: " << endl;
6         cin.ignore();
7         getline(cin, input);
8         if(!input.empty()){
9             this->preguntas[n_pregunta - 1]->set_tiempo_est(stof(input));
10            this->recalcular_tiempo_total();
11        }
12        // ... más código de modificación
13    } else { // Seleccion_Mult
14        // ... (obtener input para enunciado, nivel tax, tiempo)
15        cout << "Modificar el tiempo estimado: " << endl;
16        cin.ignore();
17        getline(cin, input);
18        if(!input.empty()){
19            this->preguntas[n_pregunta - 1]->set_tiempo_est(stof(input));
20            this->recalcular_tiempo_total();
21        }
22        // ... más código de modificación
23    }
24 }

```

La implementación del método `recalcular_tiempo_total()` asegura que el tiempo total se mantenga sincronizado con la suma de los tiempos individuales de cada pregunta:

```

1 void Prueba::recalcular_tiempo_total() {
2     float nuevo_tiempo = 0;
3     for (Pregunta* pregunta : preguntas) {
4         nuevo_tiempo += pregunta->get_tiempo_est();
5     }
6     this->tot_tiempo = nuevo_tiempo;
7 }

```

El sistema actualiza automáticamente el tiempo total en los siguientes casos:

- Al insertar una nueva pregunta:

```

1 void Prueba::insertar_pregunta(Pregunta *pregunta) {
2     this->preguntas.push_back(pregunta);
3     this->recalcular_tiempo_total();
4 }

```

- Al eliminar una pregunta existente:

```

1 void Prueba::eliminar_pregunta(int n_pregunta) {
2     this->preguntas.erase(this->preguntas.begin() + n_pregunta
3         - 1);
4     this->recalcular_tiempo_total();
5 }

```

- Al modificar el tiempo estimado de una pregunta:

```

1 // En el método modificar_pregunta
2 if(!input.empty()) {
3     this->preguntas[n_pregunta - 1]->set_tiempo_est(stof(input)
4         );
5     this->recalcular_tiempo_total();
6 }

```

Esta implementación ofrece varias ventajas:

- **Consistencia:** El tiempo total siempre refleja la suma real de los tiempos de las preguntas
- **Automatización:** No requiere intervención manual para mantener el tiempo actualizado
- **Encapsulamiento:** El método `recalcular_tiempo_total()` es privado, asegurando que solo se llame cuando es necesario
- **Flexibilidad:** Permite modificar tiempos individuales manteniendo la coherencia del tiempo total

## 2.4. Clase Menu

La clase `Menu` implementa la interfaz de usuario y la lógica principal del programa, definida en `utils.h` e implementada en `utils.cpp`. 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   string input;
5   limpiar_pantalla();
6   cout << "Crear prueba" << endl;
7
8   do {
9       cout << "Ingrese el numero de preguntas (debe ser positivo): ";
10      getline(cin, input);
11      try { tot_preguntas = stoi(input); } catch (...) { tot_preguntas =
12          0; }
13  } while (tot_preguntas <= 0);
14
15  do {
16      cout << "Ingrese el tiempo total estimado (o 0 para usar el valor
17          por defecto): ";
18      getline(cin, input);
19      try {
20          tot_tiempo = stof(input);
21          if (tot_tiempo < 0) tot_tiempo = 0;
22      } catch (...) { tot_tiempo = 0; }
23  } while (tot_tiempo < 0);
24
25  if (tot_preguntas > 0) {
26      if (tot_tiempo <= 0) {
27          tot_tiempo = tot_preguntas * 3.0f; // Tiempo por defecto
28          cout << "Usando tiempo por defecto: " << tot_tiempo << " minutos"
29              << endl;
30      }
31      Prueba *prueba = new Prueba(tot_preguntas, tot_tiempo);
32      this->pruebas.push_back(prueba);
33      cout << "Prueba creada exitosamente." << endl;
34  }
35  }

```

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(); // Implementado en clase base y
    derivadas
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     }

```

## 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:



- Implementaci'ón de un sistema funcional para la gesti'ón de pruebas
- Incorporaci'ón efectiva de los niveles taxon'ómicos de Bloom
- Desarrollo de una interfaz de usuario intuitiva
- Implementaci'ón de todas las operaciones CRUD requeridas

### **3.2. Reflexi'ón sobre el Trabajo**

El desarrollo de este proyecto permiti'o:

- Profundizar en la aplicaci'ón pr'actica de conceptos de POO
- Comprender la importancia de un buen dise'ño de software
- Desarrollar habilidades en la gesti'ón de memoria din'amica
- Implementar soluciones modulares y extensibles