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.

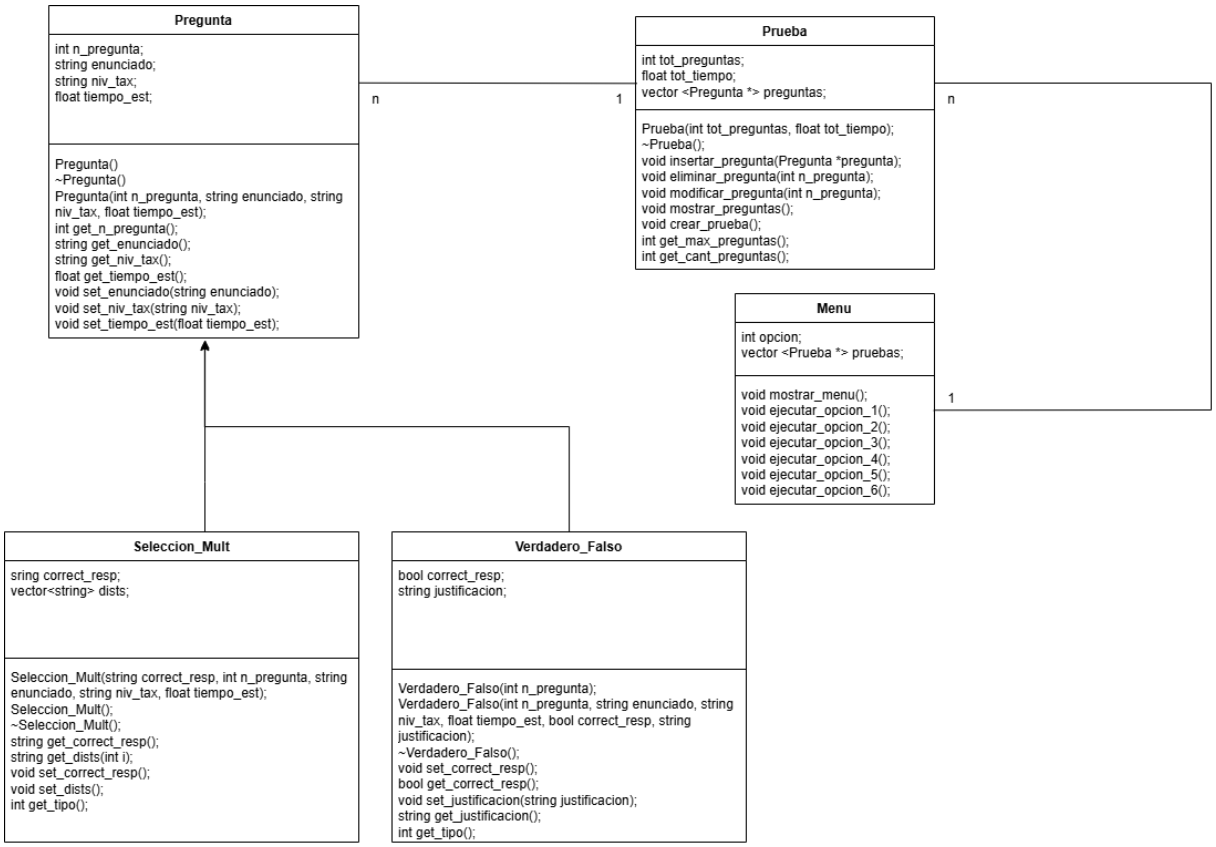

Figura 1: Diagrama de clases del sistema desarrollado.

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:

```
class Pregunta {
  private:
    int n_pregunta;
    string enunciado;
    string niv_tax;
    float tiempo_est;
  public:
    Pregunta(int n_pregunta, string enunciado, string niv_tax, float
        tiempo_est);
    virtual ~Pregunta();
    // Métodos getters y setters
    virtual void set_correct_resp()=0;
    virtual int get_tipo();
    void set_tiempo_est(float tiempo_est);
    float get_tiempo_est();
};
```

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:

```
class Seleccion_Mult : public Pregunta {
  private:
    string correct_resp;
    vector<string> dists;
  public:
    Seleccion_Mult(string correct_resp, int n_pregunta,
                   string enunciado, string niv_tax,
                   float tiempo_est);
    ~Seleccion_Mult();
    void set_correct_resp();
    void set_dists();
    string get_dists(int i);
    int get_tipo() { return 2; }
};
```

2.2.2. Clase Verdadero_Falso

Esta clase maneja las preguntas de verdadero o falso:

```
class Verdadero_Falso : public Pregunta {
  private:
    bool correct_resp;
    string justificacion;
  public:
    Verdadero_Falso(int n_pregunta, string enunciado,
                    string niv_tax, float tiempo_est,
                    bool correct_resp, string justificacion);
    ~Verdadero_Falso();
    void set_correct_resp();
    bool get_correct_resp();
    void set_justificacion(string justificacion);
    string get_justificacion();
    int get_tipo() { return 1; }
};
```

2.3. Clase Prueba

La clase `Prueba` gestiona un conjunto de preguntas:

```
class Prueba {
  private:
    int tot_preguntas;
    float tot_tiempo;
    vector<Pregunta *> preguntas;
    void recalcular_tiempo_total();
  public:
```

```
    Prueba(int tot_preguntas, float tot_tiempo);
    ~Prueba();
    void insertar_pregunta(Pregunta *pregunta);
    void eliminar_pregunta(int n_pregunta);
    void modificar_pregunta(int n_pregunta);
    void mostrar_preguntas();
    int get_max_preguntas();
    int get_cant_preguntas();
    float get_tiempo_total() const { return tot_tiempo; }
};
```

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

```
void Prueba::modificar_pregunta(int n_pregunta) {
  int tipo = this->preguntas[n_pregunta - 1]->get_tipo();
  if (tipo == 1) {  // Verdadero_Falso
    string input;
    cout << "Modificar el tiempo estimado: "<< endl;
    cin.ignore();
    getline(cin,input);
    if(!input.empty()){
        this->preguntas[n_pregunta - 1]->set_tiempo_est(stof(input));
        this->recalcular_tiempo_total();
    }
    // ... más código de modificación
  } else { // Seleccion_Mult
    // ... (obtener input para enunciado, nivel tax, tiempo)
    cout << "Modificar el tiempo estimado: "<< endl;
    cin.ignore();
    getline(cin,input);
    if(!input.empty()){
        this->preguntas[n_pregunta - 1]->set_tiempo_est(stof(input));
        this->recalcular_tiempo_total();
    }
    // ... más código de modificación
  }
}
```

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:

```
void Prueba::recalcular_tiempo_total() {
    float nuevo_tiempo = 0;
    for (Pregunta* pregunta : preguntas) {
        nuevo_tiempo += pregunta->get_tiempo_est();
    }
    this->tot_tiempo = nuevo_tiempo;
}
```

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

- Al insertar una nueva pregunta:

```
    void Prueba::insertar_pregunta(Pregunta *pregunta) {
        this->preguntas.push_back(pregunta);
        this->recalcular_tiempo_total();
    }
```

- Al eliminar una pregunta existente:

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

- Al modificar el tiempo estimado de una pregunta:

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

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:

```
class Menu {
  private:
    int opcion;
    vector<Prueba *> pruebas;
  public:
    void mostrar_menu();
    void ejecutar_opcion_1();  // Crear prueba
    void ejecutar_opcion_2();  // Insertar pregunta
    void ejecutar_opcion_3();  // Eliminar pregunta
    void ejecutar_opcion_4();  // Modificar pregunta
    void ejecutar_opcion_5();  // Mostrar preguntas
    void ejecutar_opcion_6();  // Eliminar prueba
};
```

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:

```
void Menu::ejecutar_opcion_1() {
  int tot_preguntas;
  float tot_tiempo;
```

```
  string input;
  limpiar_pantalla();
  cout << "Crear prueba" << endl;

  do {
    cout << "Ingrese el numero de preguntas (debe ser positivo): ";
    getline(cin, input);
    try { tot_preguntas = stoi(input); } catch (...) { tot_preguntas =
       0; }
  } while (tot_preguntas <= 0);

  do {
    cout << "Ingrese el tiempo total estimado (o 0 para usar el valor
       por defecto): ";
    getline(cin, input);
    try {
      tot_tiempo = stof(input);
      if (tot_tiempo < 0) tot_tiempo = 0;
    } catch (...) { tot_tiempo = 0; }
  } while (tot_tiempo < 0);

  if (tot_preguntas > 0) {
    if (tot_tiempo <= 0) {
      tot_tiempo = tot_preguntas * 3.0f; // Tiempo por defecto
      cout << "Usando tiempo por defecto: " << tot_tiempo << " minutos"
         << endl;
    }
    Prueba *prueba = new Prueba(tot_preguntas, tot_tiempo);
    this->pruebas.push_back(prueba);
    cout << "Prueba creada exitosamente." << endl;
  }
}
```

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:

```
    vector<Prueba *> pruebas;

    // En ejecutar_opcion_6 (eliminar prueba):
    delete this->pruebas[id_prueba-1];
    this->pruebas.erase(this->pruebas.begin() + id_prueba - 1);
```

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

```
    if (id_prueba > 0 && id_prueba <= this->pruebas.size()) {
      if (id_pregunta > 0 &&
          id_pregunta <= this->pruebas[id_prueba-1]->
             get_cant_preguntas()) {
        // Operación válida
      }
    }
```

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:

```
    virtual int get_tipo();  // Implementado en clase base y
        derivadas
    int get_tipo() { return 1; } // En Verdadero_Falso
    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:

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

3. Conclusi'on

El desarrollo de este trabajo nos permiti'o implementar de manera pr'actica los principales conceptos de la programaci'on orientada a objetos en C++. Logramos crear un sistema modular y estructurado que gestiona eficientemente pruebas y preguntas siguiendo la taxonom'ia de Bloom.

El uso de herencia y polimorfismo result'o fundamental para manejar los diferentes tipos de preguntas de manera uniforme, mientras que el uso de memoria din'amica nos permiti'o una gesti'on flexible de las pruebas y sus componentes.

La soluci'on desarrollada demuestra la aplicabilidad de los conceptos de POO en problemas reales, resultando en un c'odigo mantenible, extensible y bien estructurado que cumple con todos los requerimientos establecidos.

3.1. Cumplimiento de Objetivos

Se alcanzaron satisfactoriamente los siguientes objetivos:

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

3.2. Reflexi'on sobre el Trabajo

El desarrollo de este proyecto permiti'o:

- 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