Bad Smells

Class	Analysis and Design Object Oriented
Date of delivery	@Nov 22, 2020 11:59 PM
Days left	✅ Task Complete
Finished	☑️

 Juventino Aguilar Correa 3°C

Los siguientes *Bad Smells* son obtenidos mi proyecto de Matemáticas Discretas, que consta de un programa que a partir de una *matriz de incidencia* (matriz que representa las aristas y que nodos conectan en un grafo) ingresada por el usuario, genera una *matriz de adyacencia* (que representa las conexiones entre los nodos del grafo) y también nos da la opción de generar caminos y circuitos con los nodos y aristas del grafo.

- El código de este programa (el original y posterior a realizar técnicas de refactoring) se encuentra en éste misma carpeta *Bad Smells*

Long Method

Un método o función, se considera largo cuando contiene muchas líneas de código. Generalmente cuando éste tiene más de 10 líneas.

Método `generarCircuitoSimple()`

```
260  void generarCircuitoSimple(int aristaA, int anterior, int actual, int final, vector<int> &caminosST)
261  {
262      vector<int> posible = caminosST;
263      posible.push_back(actual);
264
265      if (actual == final and aristaA != -1)
266      {
267          imprimirCaminoV(posible);
268          cout << "\n=========================\n";
269          return;
270      }
271      for (int i = 0; i < r; i++)
272      {
273          auto it = find(posible.begin(), posible.end(), i);
274          //Verificar que haya un 1 en la casilla (indica arista entre esos nodos)
275          //Indica que al nodo que se quiera ir no sea el anterior
276          //Que sea el final o que no se haya aniadido en el vector
277          if (mA[actual][i] != 0 and i!= anterior and (i == final or (it == posible.end())) )
278          {
279              generarCircuitoSimple(aristaA+1,actual,i, final, posible);
280          }
281      }
282      return;
283  }
```

- El método encargado de generar circuitos simples cuenta con 20 líneas de código, por lo cual ya se le puede considerar como long method, ésto porque si observamos bien dentro del método estamos realizando 3 cosas diferentes:

  1. Primero sería el crear el vector `posible`, el cuál es una copia de los circuitos que hasta esa momento se pueden generar.
  2. Después tenemos un condicional `if`, en el cual imprimiremos el circuito posible, si el nodo actual es el nodo final, y la arista no sea la inicial.
  3. Por último tenemos un ciclo `for`, que en sí es lo principal del método, ya que desde él se manda llamar recursivamente el método para generar los siguientes circuitos posibles desde cada nodo conectado al inicial.

- Como ahora tenemos definidos fragmentos concisos del método, podemos utilizar la técnica extract method, en donde vamos a separar cada fragmento del código en otros métodos, y los sustituimos por una llamada a éstos nuevos métodos. Ahora los 3 métodos nos quedarán de la siguiente manera:

```
260    void generarCircuitoSimple(int aristaA, int anterior, int actual, int final, vector<int> &caminosST){
261        vector<int> posible = caminosST;
262        posible.push_back(actual);
263
264        condicionalCircuitoS(actual, final, aristaA, posible);
265        iteradorCircuitoS(posible, actual, anterior, final, arista);
266        return;
267    }
268
269    void iteradorCircuitoS(vector<int> &posible, int actual, int anterior, int final, int arista){
270        for (int i = 0; i < r; i++)
271        {
272            auto it = find(posible.begin(), posible.end(), i);
273            //Verificar que haya un 1 en la casilla (indica arista entre esos nodos)
274            //Indica que al nodo que se quiera ir no sea el anterior
275            //Que sea el final o que no se haya aniadido en el vector
276            if (mA[actual][i] != 0 and i!= anterior and (i == final or (it == posible.end())) )
277            {
278                generarCircuitoSimple(aristaA+1,actual,i, final, posible);
279            }
280        }
281    }
282
283    void condicionalCircuitoS(int actual, int final, int aristaA, vector<int> &posible){
284        if (actual == final and aristaA != -1)
285        {
286            imprimirCaminoV(posible);
287            cout << "\n=========================\n";
288            return;
289        }
290    }
```

- De ésta forma, ninguno de los métodos es considerado un long method.

Comments

- Cuando un método esta repleto de comentarios.

Método `iteradorCircuitoS()`

```
269 void iteradorCircuitoS(vector<int> &posible, int actual, int anterior, int final, int arista){
270     for (int i = 0; i < r; i++)
271     {
272         auto it = find(posible.begin(), posible.end(), i);
273         //Verificar que haya un 1 en la casilla (indica arista entre esos nodos)
274         //Indica que al nodo que se quiera ir no sea el anterior
275         //Que sea el final o que no se haya aniadido en el vector
276         if (mA[actual][i] != 0 and i!= anterior and (i == final or (it == posible.end())) )
277         {
278             generarCircuitoSimple(aristaA+1,actual,i, final, posible);
279         }
280     }
281 }
```

- En éste método (resultado del punto anterior) tenemos muchas líneas de comentararios (en relación al total de líneas del método) que explican la condicional, la cual sin estos sería difícil de comprender. Por lo anterior debemos modificar la estructura del código para que los comentarios sean innecesarios.

- Para realizar ésto podemos usar el método extract variable, en la cual cada parte de la expresión del condicional la convertimos en una variable, donde su nombre estará basado en el comentario que la explica.

```
269 void iteradorCircuitoS(vector<int> &posible, int actual, int anterior, int final, int arista){
270     for (int i = 0; i < r; i++)
271     {
272         auto it = find(posible.begin(), posible.end(), i);
273 
274         const bool existeAristaEntreNodos = mA[actual][i] != 0;
275         const bool esNodoAnterior = i!= anterior;
276         const bool esNodoFinal = i == final;
277         const bool nodoEnVector = (it == posible.end());
278 
279         if ( existeAristaEntreNodos and esNodoAnterior and ( esNodoFinal or nodoEnVector) )
280         {
281             generarCircuitoSimple(aristaA+1,actual,i, final, posible);
282         }
283     }
284 }
```

Duplicate Code

- Sucede cuando tenemos código muy parecido o igual en varias partes del programa.

Métodos `condicionalCaminoS()` y `condicionalCircuito()`

```
299 bool condicionalCircuitoS(int actual, int final, int aristaA, vector<int> &posible){
300     if (actual == final and aristaA != -1)
301     {
302         imprimirCaminoV(posible);
303         cout << "\n=========================\n";
304         return true;
305     }
306     return false;
307 }
```

```
263 bool condicionalCaminoS(int actual, int final, vector<int> &posible){
264     if (actual == final)
265     {
266         imprimirCaminoV(posible);
267         cout << "\n=========================\n";
268         return true;
269     }
270     return false;
271 }
```

- En éste ejemplo, en ambos métodos tenemos exactamente las mismas tres líneas de código, lo cual es considerado duplicate code.Lo que hacemosen estas líneas es imprimir un camino posible junto con otros caractéres que fungen de separadores. Para no tener código duplicado podemos apoyarnos del extract method, con el cuál generaremos un método para

esta operación que será llamado desde ambos sustituyendo esas tres líneas.

```
290 bool condicionalCaminoS(int actual, int final, vector<int> &posible){
291     if (actual == final)
292     {
293         imprimirCaminoPosible(posible);
294     }
295     return false;
296 }
297
298 bool condicionalCircuitoS(int actual, int final, int aristaA, vector<int> &posible){
299     if (actual == final and aristaA != -1)
300     {
301         imprimirCaminoPosible(posible);
302     }
303     return false;
304 }
305
306 void imprimirCaminoPosible(vector<int> &posible){
307     imprimirCaminoV(posible);
308     cout << "\n=========================\n";
309     return true;
310 }
```

- Ahora simplemente llamando al nuevo método `imprimirCaminoPosible()` desde ambos métodos realizamos las mismas operaciones, logrando así tener menos líneas repetidas en nuestro código.
- Puede parecer que esto nos hizo utilizar muchas más líneas de código, pero si vemos el programa completo nos daremos cuenta de que al igual que éstos métodos existen otros dos muy parecidos que realizan la misma acción al imprimir un camino, por lo cual ahorraremos aún más líneas.

Switch Staments

- Tenemos un switch muy complejo o con casos muy extendidos

Método `menuCaminos()`

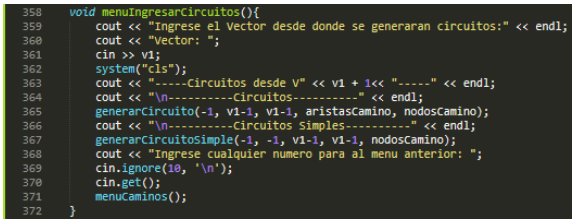
- En éste método tenemos un condicional `switch`, el cual se extiende demasiado en 2 de sus casos, lo cual genera que nuestro método sea demasiado largo.

- Para solucionar éste problema lo que podemos hacer es que las líneas de código de cada caso sean ejecutadas en otros métodos específicos de cada caso, ésto lo logramos con la técnica extract method..

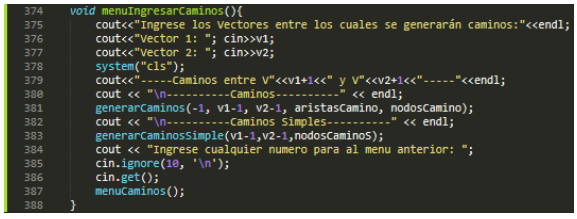
```
void menuCaminos(){
    int opt = -1, v1, v2;
    system("cls");
    cout<<"-----Generar Caminos o Circuitos-----\nQue desea generar:"<<endl;
    cout<<"\t1.Caminos\n\t2.Circuitos\n\t0. Volver al Menu Principal"<<endl;
    cout<<"Opcion: ";cin>>opt;
    cout<<endl;
    switch (opt)
    {
    case 0:
        menuPrincipal();
        break;
    case 1:
        cout<<"Ingrese los Vectores entre los cuales se generarán caminos:"<<endl;
        cout<<"Vector 1: "; cin>>v1;
        cout<<"Vector 2: "; cin>>v2;
        system("cls");
        cout<<"-----Caminos entre V"<<v1+1<<" y V"<<v2+1<<"-----"<<endl;
        cout << "\n----------Caminos----------" << endl;
        generarCaminos(-1, v1-1, v2-1, aristasCamino, nodosCamino);
        cout << "\n----------Caminos Simples----------" << endl;
        generarCaminosSimple(v1-1,v2-1,nodosCaminos);
        cout << "Ingrese cualquier numero para al menu anterior: ";
        cin.ignore(10, '\n');
        cin.get();
        menuCaminos();
        break;
    case 2:
        cout << "Ingrese el Vector desde donde se generaran circuitos:" << endl;
        cout << "Vector: ";
        cin >> v1;
        system("cls");
        cout << "-----Circuitos desde V" << v1 + 1<< "-----" << endl;
        cout << "\n----------Circuitos----------" << endl;
        generarCircuito(-1, v1-1, v1-1, aristasCamino, nodosCamino);
        cout << "\n----------Circuitos Simples----------" << endl;
        generarCircuitoSimple(-1, -1, v1-1, v1-1, nodosCamino);
        cout << "Ingrese cualquier numero para al menu anterior: ";
        cin.ignore(10, '\n');
        cin.get();
        menuCaminos();
        break;
    default:
        system("cls");
        menuCaminos();
        break;
    }
}
```

- De este forma el switch nos quedaría como simples llamadas a otros métodos, en vez de tener que tener toda la ejecución ahí. Y el código de los casos `1` y `2`, ahora está en los métodos `menuIngresarCircuitos()` y `menuIngresarCaminos()`

```
void menuCaminos(){
    int opt = -1, v1, v2;
    system("cls");
    cout<<"-----Generar Caminos o Circuitos-----\nQue desea generar:"<<endl;
    cout<<"\t1.Caminos\n\t2.Circuitos\n\t0. Volver al Menu Principal"<<endl;
    cout<<"Opcion: ";cin>>opt;
    cout<<endl;
    switch (opt)
    {
    case 0:
        menuPrincipal();
        break;
    case 1:
        menuIngresarCaminos();
        break;
    case 2:
        menuIngresarCircuitos();
        break;
    default:
        system("cls");
        menuCaminos();
        break;
    }
}
```



```
void menuIngresarCircuitos(){
    cout << "Ingrese el Vector desde donde se generaran circuitos:" << endl;
    cout << "Vector: ";
    cin >> v1;
    system("cls");
    cout << "-----Circuitos desde V" << v1 + 1<< "-----" << endl;
    cout << "\n----------Circuitos----------" << endl;
    generarCircuito(-1, v1-1, v1-1, aristasCamino, nodosCamino);
    cout << "\n----------Circuitos Simples----------" << endl;
    generarCircuitoSimple(-1, -1, v1-1, v1-1, nodosCamino);
    cout << "Ingrese cualquier numero para al menu anterior: ";
    cin.ignore(10, '\n');
    cin.get();
    menuCaminos();
}
```



```
void menuIngresarCaminos(){
    cout<<"Ingrese los Vectores entre los cuales se generarán caminos:"<<endl;
    cout<<"Vector 1: "; cin>>v1;
    cout<<"Vector 2: "; cin>>v2;
    system("cls");
    cout<<"-----Caminos entre V"<<v1+1<<" y V"<<v2+1<<"-----"<<endl;
    cout << "\n----------Caminos----------" << endl;
    generarCaminos(-1, v1-1, v2-1, aristasCamino, nodosCamino);
    cout << "\n----------Caminos Simples----------" << endl;
    generarCaminosSimple(v1-1,v2-1,nodosCaminos);
    cout << "Ingrese cualquier numero para al menu anterior: ";
    cin.ignore(10, '\n');
    cin.get();
    menuCaminos();
}
```

Dead Code

- Sucede cuando una variable, parámetro, clase o método ya no es usado, porque es obsoleto

Método `imprimirCaminoA()`

```
void imprimirCaminoV(vector<int> &vimprimir){
    cout << " Representado en Vertices\n\t";
    for (int i = 0; i < vimprimir.size(); ++i)
    {
        if (i!=0){cout<<"->";}
        cout<<"V"<<vimprimir[i]+1;
    }
    cout<<endl;
}
void imprimirCaminoA(vector<int> &vimprimir)
{
    cout<<" Representado en Aristas\n\t";
    for (int i = 0; i < vimprimir.size(); ++i)
    {
        if (i!=0){cout<<"->";}
        cout<<"A"<<vimprimir[i]+1;
    }
}
```

- En el programa es fácil observar que cuando se generar Caminos o Circuitos estos son impresos en pantalla representados en vértices, y solo en algunos casos se representan en aristas (con el método `imprimirCaminoA()`), lo cuál en un inicio se utilizo para hacer pruebas con el código viendo si se respetaba las reglas de la Teoría de Grafos, donde si por una arista ya hemos cruzado en un camino, ya no debemos volver a cruzar por ella.
- Pero después de esto el método ya no tuvo otro motivo para seguir ahí ya que se cuenta con el método `imprimirCaminoV()`, por lo cuál ahora se le puede considerar como dead code.

- Para solucionar ésto basta con eliminar el método y sus llamadas desde otros, no se requiere ningún otro tipo de técnica de refactoring.