TP Final

Complejidad Temporal, Estructura de Datos y Algoritmos

Profesor

Ing. Salinas Mauro

Alumno

Paez Martín (64311)

Fecha: 20/11/2022

ÍNDICE

1. Introducción

El presente informe se confeccionó con el objetivo de presentar el trabajo final de la materia Complejidad Temporal, Estructura de Datos y Algoritmos, de la carrera Ingeniería Informática.

Partiendo de un análisis sobre el proyecto presentado por la cátedra, se brindan sugerencias y una nueva implementación. Se trata de una aplicación de computadora que emula al famoso juego de mesa “Who is who”, en el cual se debe inferir el personaje elegido por el oponente mediante una seguidilla de preguntas.

Es de fundamental interés la implementación de cuatro métodos en particular: aquel que se ocupa de crear el árbol de decisión, para que la máquina sepa que pregunta debe hacer en cada caso, y tres consultas, que permiten recabar información del estado del árbol en cada momento de la partida.

2. Metodología

Como regla general, en cada sección comienza con la presentación de las clases provistas por la cátedra, o sea, las originales. Por lo tanto, para interpretar el contenido hay que mirar el código sin modificaciones. Luego se dan sugerencias de cambio, seguidas por las nuevas implementaciones. Como es de esperar, para comprender estas últimas a fondo, es necesario analizar el nuevo código.

Cabe aclarar, que la primera parte del informe consta de la explicación teórica en la que se basa el algoritmo que permite armar el árbol de decisión.

Se unificó el código en un solo idioma, el inglés, esto incluye renombrar las clases del proyecto. Para ello, se priorizó el largo de las palabras por sobre la traducción exacta. Cada sección del informe aborda una clase distinta y se titula con el nuevo nombre. En algunos casos la correspondencia entre nombres no es directa ya que se aplicó herencia o interfaces. Sin embargo, el orden establecido de la información guiará al lector hacia la comprensión de la totalidad del proyecto.

Por último, si bien en cada sección se presentarán diagramas UML, se deja en el anexo del informe los diagramas completos que detallan las relaciones entre las diferentes clases, tanto de la nueva implementación como de la original.

3. Back-End

El programa está claramente separado en Back-End y Front-End. El primero contempla la abstracción más baja, o sea, el manejo a archivos y algoritmos para manipular la información empleando estructuras de datos convencionales: listas, vectores, hashsets, etc. Estas clases, cada una con un rol bien distinguido, se encuentran en la carpeta Low Abstraction. El producto final es una estructura de datos, internamente más compleja, pero más simple de utilizar para el usuario, nombrada RowData.

RowData, a través de la jerarquía de clases encontradas dentro de la carpeta Sorter, se fracciona para convertirse en IData, o sea, la información refinada que almacena el árbol de decisión. Este, a su vez, es utilizado por la computadora, o, mejor dicho, la clase Machine, para competir contra el usuario.

3.1 ÁRBOL DE DECISIÓN

El árbol de decisión contiene preguntas en los nodos padres. Cada una se puede responder por “Si” y por “No”, siendo los hijos la siguiente pregunta. Esta característica asegura que cada padre tenga exactamente dos hijos. Esta deducción fue ampliamente aprovechada a lo largo del proyecto.

Como las hojas no tienen hijos, una vez en ellas, ya no hay más preguntas por hacer. Por lo tanto, contienen la deducción final, o sea, los posibles personajes del oponente, en este caso, del usuario (el árbol refleja la inteligencia de la máquina).

3.2 PROBABILIDAD VS GANANCIA

Probablemente la primera dificultad que hay que sortear a la hora de analizar el código, se trata de abstraer dos variables diferentes. Ambas reflejan probabilidades distintas, pero estrechamente vinculadas.

La primera de ellas es la probabilidad de que uno de los personajes almacenados en una hoja en particular, sea el que realmente eligió el usuario. Esta información, que puede ser útil para otras aplicaciones, no aporta nada al juego. El motivo es simple, cada nodo hoja solo va a tener un único valor. Esto se debe a que el juego está pensado para que las características de cada personaje permitan generar preguntas que los identifiquen inequívocamente.

Por motivos didácticos, se creó un clon del personaje Alex, llamado Alexander. Al ser idénticos, comparten la misma hoja, y es posible observar que el algoritmo calcula para ambos el 50% de probabilidad.

El segundo cálculo se relaciona con el orden de las preguntas, o sea, como se arma el árbol de decisión. Si por ejemplo, cambiamos los valores en el archivo que contiene la información de cada personaje, el árbol se adapta dinámicamente. Esto se debe a que es capaz de calcular cual es la mejor métrica en cada momento, o sea, sobre que característica conviene indagar en cada situación. Para ello realiza un cálculo matemático que involucra la Impureza de Gini.

3.3 IMPUREZA DE GINI

La impureza de Gini es comúnmente utilizada en árboles de clasificación. Mediante un algoritmo que la aplica de forma ponderada, se logra decidir cual es la pregunta que conviene realizar en cada ocasión. En otras palabras, cuando el juego comienza, la máquina ya conoce todas las posibilidades, luego, el mecanismo será metódico y preestablecido.

Lo que la hace inteligente, es el hecho de que, si cambiamos las características de cada personaje, la máquina sabrá por sí sola adaptar el árbol a su forma más conveniente.

3.3.1 CASO DE ESTUDIO

Tenemos 4 alumnos, y, para cada uno, hay información del tiempo de estudio, la motivación, su asistencia a clases y las notas que obtuvieron. Entonces, queremos analizar cual de los tres atributos es mejor para determinar si un alumno aprobará.

Nota	Motivación (M)	Asistencia (A)	Tiempo (T)
Aprobó (A)	Alta	Alta	Alto
Aprobó (A)	Alta	Alta	Bajo
Desaprobó (D)	Baja	Alta	Bajo
Desaprobó (D)	Baja	Alta	Bajo

Tabla 1. Datos de 4 alumnos.

3.3.2 Cálculos y Resultados

M	Tot(T)	Nota	Cant (C)	C/T (P)	P^2	∑ (S)	1-S	Pond	Pond	∑
Alta	2	A	2	1	1	1	0	50%	0	0
		D	0	0	0					
Baja	2	A	0	0	0	1	0	50%	0	
		D	2	1	1					

Tabla 2. Cálculo de la Impureza de Gini para la Motivación.

T	Tot(T)	Nota	Cant(C)	C/T(P)	P^2	∑ (S)	1-S	Pond	Pond	∑
Alto	1	A	1	1	1	1	0	25%	0	0.25
		D	0	0	0					
Bajo	3	A	1	1/3	0.33	0.77	0.33	75%	0.25	
		D	2	2/3	0.44					

Tabla 3. Cálculo de la Impureza de Gini para el Tiempo.

	Motivación	Asistencia	Tiempo
Impureza de Gini	0	0.5	0.25

Tabla 4. Valor de la impureza de Gini para cada atributo

3.3.3 Conclusión

La motivación es el mejor indicador, mientras que la asistencia no aporta absolutamente nada. Cuando el resultado es cero, se interpreta con una probabilidad del 100% de aprobar.

Los resultados tienen lógica, cada alumno motivado aprobó sin excepción, mientras que independientemente de que todos tuvieron buena asistencia, la mitad desaprobó.

Analizando el tiempo, solo la mitad de los aprobados dedico mucho tiempo, sin embargo, todos los desaprobados tienen ese común denominador. Más bien parecería ser una variable que incide con mayor peso en la probabilidad de que un alumno desapruebe.

Sin embargo, podemos verlo desde el punto de vista de que, hay uno que aprobó dedicando mucho tiempo y uno de los que no le dedico desaprobó. De los otros dos, uno aprobó y el otro no, a pesar de que ninguno dedicó mucho tiempo. O sea, la mitad de los datos no aportan nada y la otra mitad sí, siendo que la Impureza de Gini dio el punto medio.

Claro que la tabla de datos solamente tenía tres filas, pero con la finalidad de comprender el funcionamiento y tener una idea de su efectividad, los resultados fueron muy satisfactorios.

3.4 Spliter

La clase Praticionador brinda métodos para dividir, ponderar y contabilizar la información, previamente parseada, del archivo preguntas.csv.

Esta clase es la encargada de realizar el cálculo de la impureza de Gini, aplicando el concepto en árboles de clasificación. La “information gain” (ganancia) se calcula como la impureza de Gini del nodo actual, menos la suma ponderada de las impurezas de sus hijos.

En otras palabras, haciendo una analogía con el ejemplo visto: la “Motivación” es al “Color de pelo”, como “Alta/Baja” lo es a “EsNegro/NoEsNegro”. Esto se calcula para el conjunto actual de filas, así como para los subconjuntos resultantes de dividirlo en función a una pregunta determinada, obteniendo así, las impurezas del nodo actual y sus hijos respectivamente.

Como las preguntas son del estilo: “¿tiene pelo negro?”, las respuestas posibles son siempre dos. Entonces, se logra lo que obtuvimos en el ejemplo, al convertir las notas en “Aprobado/Desaprobado”. De este modo el cálculo es equivalente al de las tablas. Si se observa la suma de los porcentajes en las filas de la tabla, a la hora de ponderar, es el 100%. Del mismo modo, se pondera al nodo izquierdo con su probabilidad, computada acorde a la cantidad de filas que le corresponden respecto del total, y al derecho como uno menos el izquierdo.

3.4.1 SUGERENCIAS

El método find_best_split() retorna la ganancia, cuando en realidad, la información útil para nuestro software es la subdivisión de filas.

No provee un nivel más de abstracción, sus usuarios siguen manipulando estructuras tales como IList<IList<IList<string>>>, filas o encabezados.

Además, se le dio un atributo col a la clase Pregunta, que nada tiene que ver con su representación interna, si no que hace referencia a un índice de vectores que emplea la clase Particionador.

El método class_count esté acoplado al formato empleado para el archivo, ese comportamiento se podría delegar a clases con menor nivel de abstracción.

El método find_best_split recibe por separado todo lo que constituye a un ConjuntoDeDatos, y analizando, sus usuarios disponen del conjunto y lo particionan para usarlos como parámetros.

Se pueden usar menos sentencias de la forma de: IList<IList<IList<string>>>

```
IList<IList<IList<string>>> res= (IList<IList<IList<string>>>) partition(rows, question);
```

```
public static Dictionary<Pregunta, double> find_best_split(IList<IList<string>> rows, IList<string> header)
```

Ilustración 1. Algunos ejemplos del código en el método estático find_best_split() de la clase Particionador.

3.4.2 NUEVA IMPLEMENTACIÓN

Pasó de ser una clase con métodos estáticos, a un patrón Strategy, contenido en RawData. Ya no utiliza más estructuras tales como IList anidados, todo el manejo de la información es a través de RawData.

El concepto de ganancia se encapsuló dentro de Particionador, quien se ocupa de utilizar su estrategia para elegir la pregunta y luego retornarla, o null, en caso de que ya no haya ninguna más por hacer. Se utilizaron tuplas para devolverlas junto a la subdivisión originada a raíz de la misma.

Se implementaron las sugerencias mencionadas, reduciendo código, desacoplando el formato del archivo, así como los atributos de la clase pregunta del comportamiento interno de Particionador.

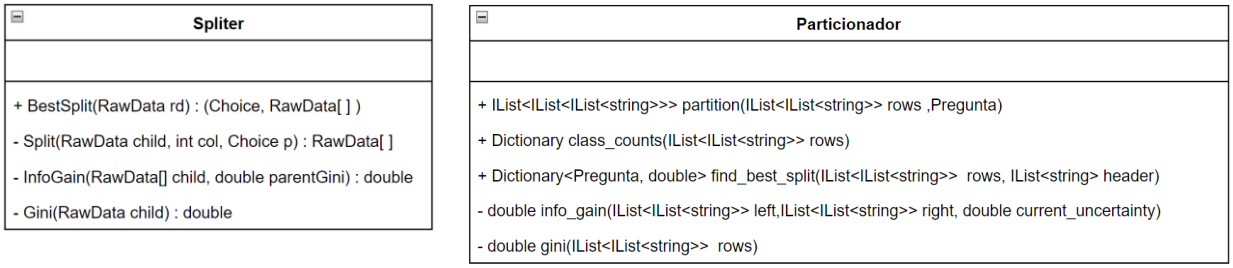


Ilustración 2. Diagrama UML de las clases Particionador ySpliter

3.5 SORTER

El Clasificador agrega un nivel de abstracción a los datos de preguntas.csv. Valiéndose del Particionador determina el contenido de los nodos. El valor del atributo ganancia se obtiene aplicando la Impureza de Gini, o sea, se analiza la probabilidad de errar en caso de intentar responderla. Por lo tanto, para identificar una hoja se compara contra cero, ya que es imposible errar si no hay más preguntas que hacer (contemplando lo anteriormente mencionado).

3.5.1 SUGERENCIAS

Los atributos ganancia y pregunta almacena, en conjunto, la misma información que el diccionario respuestas, que además es una colección con un único elemento. No tiene sentido de ser ese atributo, debería ser una variable local del constructor. Además, hay otro atributo, preguntas (en plural), que guarda la única clave del diccionario en otra colección, o sea, la información de pregunta, está en respuestas (diccionario) y en preguntas (lista).

Si querían evitar definir una clase, podrían haber usado Tuple, en vez de un diccionario, ya que es poco intuitivo a la hora de interpretar el código. Incluso el nombre respuestas, en plural, acarrea confusión.

El atributo filas, tiene la información de cada línea del archivo preguntas.csv. La información se duplica y divide en un vector de listas, que también es un atributo de instancia. En vez de guardar el objeto filas, podría usarse un atributo int y calcular de antemano el resultado de obtenerDatoHoja(), que para el caso es inmutable y además es para lo único que lo se utiliza.

El algoritmo find_best_split divide la información en dos y retorna, tanto la ganancia, como la pregunta usada para calcularla. Su único usuario, Clasificador, vuelve a particionarla para obtener el vector de listas descrito anteriormente, o sea, se realiza el mismo trabajo dos veces.

Quitaría obternerDatosDerecho() y obternerDatosIzquierdo(). No se usan más que como auxiliares de otro método, que tiene una sola línea.

Por último, se podría retornar DecisionData, al final de cuentas es lo que guardan los nodos, evitando obligar al usuario a instanciarlo en cada oportunidad, y, brindando así, un mayor nivel de abstracción.

```
        Dictionary<Pregunta, double> result = new Dictionary<Pregunta, double>();
        if (best_question == null) {
            result.Add(new Pregunta(0, "", ""),best_gain);
        }else
            result.Add(best_question,best_gain);
        return result;
    }
```

Ilustración 3. Últimas líneas del método find_best_split(). Retorna el diccionario que es almacenado en el atributo ¨respuestas´ de Clasificador. Como se observa, siempre va a devolver un único valor.

3.5.2 NUEVA IMPLEMENTACIÓN

Con los cambios planteados, la implementación anterior estaría bien. Sin embargo, no se aprovecha bien el paradigma orientado a objetos.

Dependiendo de si entregaba datos para una hoja o para un padre, el clasificador tenía métodos y atributos que nunca se usaban. Por ellos se creó la interfaz ISorter, implementada por SLeaf y SParent.

Además, se creó una fábrica de ISorter, quien llama al método que calcula la impedancia de Gini, para así poder determinar cual de los dos instanciar en cada caso. Se empleó una variación del patrón, las fábricas concretas no son hijos. Podría, incluso, haber bastado solo con el padre, pero mejor dejarlas escritas, por si es necesario reutilizar el código eventualmente.

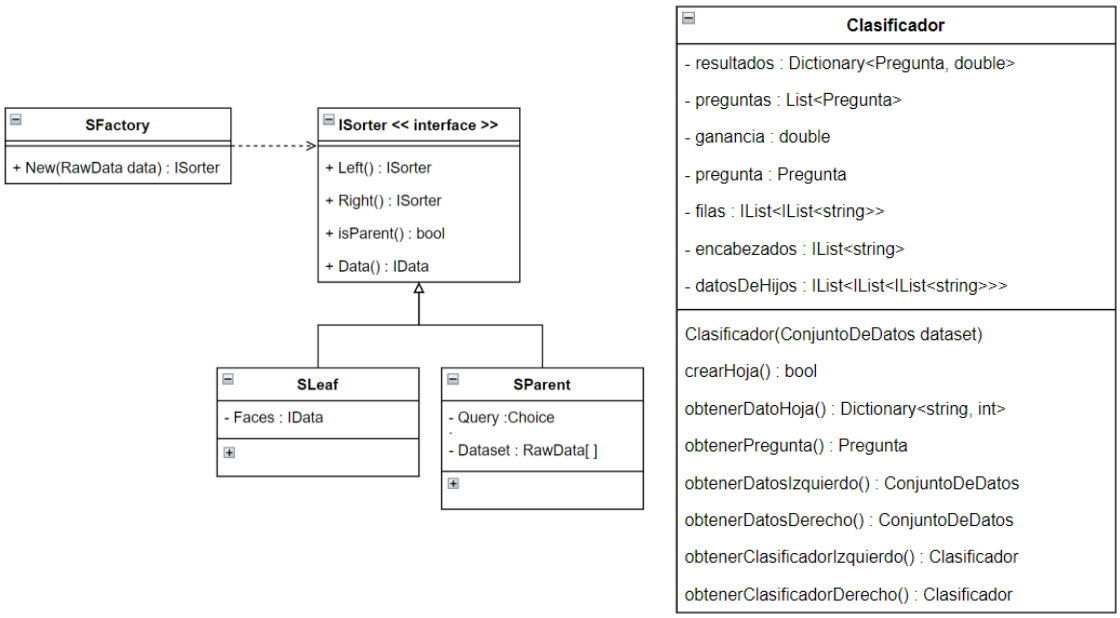


Ilustración 4. UML de los clasificadores. Los hijos implementan únicamente los métodos de la interfaz. La fábrica se graficó en su versión simplificada, 4 líneas de código. La implementación final contempla dos fábricas concretas muy simples, solo por una cuestión de emplear buenas prácticas.

3.6 RawData

ConjuntoDeDatos encapsula la información *parseada* del archivo preguntas.csv. La misma es almacenada en estructuras de datos que implementan IList.

3.6.1 Sugerencias

Hay un constructor que no se utiliza, y, además, es prácticamente código repetido.

Si bien brinda algunos métodos para manipular la información, no son aporte significativo al proyecto, básicamente la clase se limita a dar getters de estructuras anidadas de muchos IList.

3.6.2 Nueva implementación

ConjuntoDeDatos pasó a llamarse RawData. Básicamente, las capas superiores del proyecto accederán a la información del archivo a través de esta nueva estructura de datos implementada con listas y vectores. Provee todo lo necesario, y es el único que intercambia mensajes con Parser (CvsParser) y Spliter (Particionador).

El método que construía un diccionario con los nombres y ocurrencias de cada personaje, ahora es un método de instancia de RawData.

Se adicionó un método similar: ToHashSet(), útil para aquellos nodos hoja con varias predicciones posibles (en nuestro caso, por motivos didácticos, el personaje Alexander, mencionado anteriormente).

Además, utiliza a Spliter, como patrón Strategy, para la división de la información.

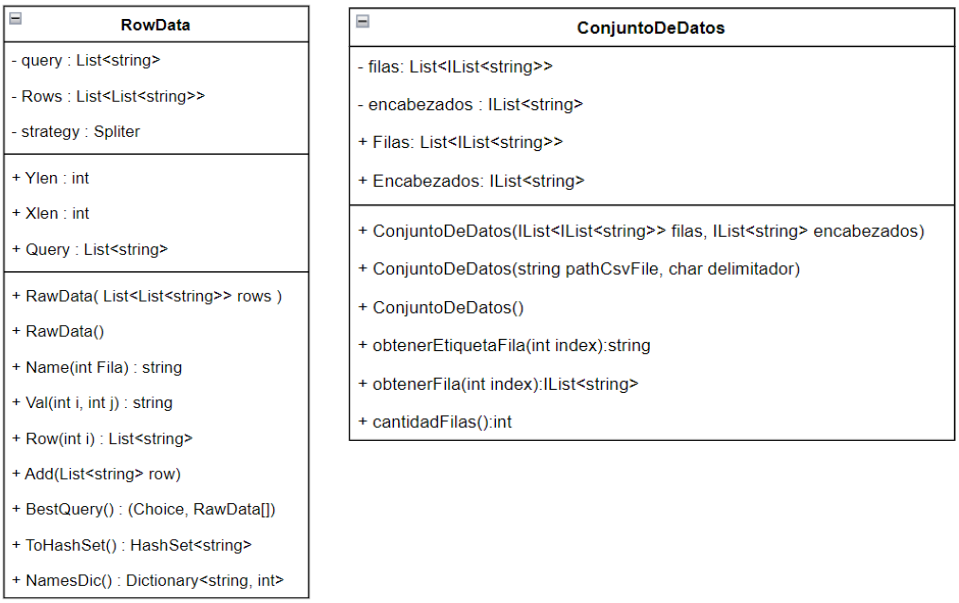


Ilustración 5. UML de RawData y ConjuntosDeDatos.

3.7 IDATA

El concepto detrás de la clase DecisionData, es tan simple como importante. Permite encapsular el dato de cada nodo, de modo que el árbol maneja por igual los diccionarios (nodos hoja) y las preguntas.

3.7.1 SUGERENCIAS

El método imprimirPredicciones() no contemplaba el caso en el que un nodo hoja tuviese más de un posible personaje. Además, del mismo modo que ocurrió en Clasificador, la implementación no era orientada a objetos, el condicional del método ToString() así como los atributos que se usan dependiendo de la instancia, dan cuenta de ello.

3.7.2 NUEVA IMPLEMENTACIÓN

Pregunta (que luego se dividió, como se explicará más adelante) pasó a implementar IDato, de hecho, no hubo que modificar código, pero con este simple cambio se evitaron los intermediarios DecisionData. Por otro lado, la clase DecisionData pasó a ser Predicciones, con el código rescatado de su predecesora.

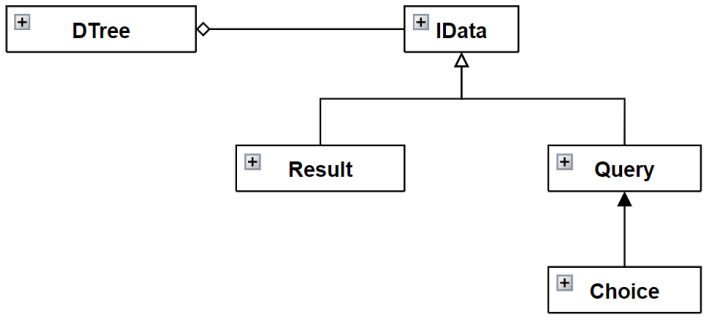


Ilustración 6. Diagrama UML con las abstracciones que emplea el árbol para almacenar la información.

3.8 QUERY, CHOICE Y RESULT

La clase Pregunta hace honor a su nombre, con la particularidad de estar asociada a un personaje en específico. Por lo tanto, no solo tiene el comportamiento de “mostrarse”, si no que también de “responderse”, ya que dentro en su representación interna contiene la respuesta, o sea, el valor de la característica particular del personaje sobre el cual se interroga.

3.8.1 SUGERENCIAS

Las preguntas son del tipo, ¿No tiene ojos azules?, lo cual complica la decisión a la hora de responder por sí o por no.

Hay código repetido en múltiples variaciones del método coincide(), de las cuales solo una se usa.

El atributo columna no es representativo de la clase, no debería estar.

El método coincide no realiza una comparación acorde a lo que se espera de la clase y contempla casteos a double que no tienen sentido de ser dentro del proyecto.

TextoParaUsuario() brinda un modo efectivo de presentar la información, y, debería reemplazar la implementación de ToString().

Es demasiado permisiva con su estado interno, y de hecho ni siquiera es aprovechada por el resto del proyecto. Los métodos que muestran la información se podrían calcular una sola vez, por eficiencia, si esas properties no tuvieran los set. Del mismo modo, no sería necesario el atributo encabezado, que, por cierto, no debería ser un concepto aplicable a este nivel de abstracción.

3.8.2 NUEVA IMPLEMENTACIÓN

Se contemplaron todas las sugerencias, restando aproximadamente 30 líneas de código sobre la porción correspondiente a la implementación original, y, aun así, con un comportamiento más completo.

En un principio no fue tan claro comprender que una pregunta también almacenaba una respuesta. Como el usuario también las hace, pero la información de la respuesta no es necesaria en ese caso, puede generar confusión. Se aplicó herencia, de moto tal que el FrontEnd utiliza Query para desplegar la lista de preguntas, mientras que el árbol almacena Choice, o sea, la pregunta con su respuesta.

Un ejemplo claro de la importancia de este concepto se observa en el método StartGame() en la clase Machine, la cual emplea un Hashset para almacenar las diferentes preguntas que van a llenar la lista de opciones brindada al usuario. Por dar un ejemplo, si se pregunta: ¿Tiene pelo negro?, habría dos respuestas posibles, y, por consiguiente, dos instancias distintas de Pregunta. La única forma de que el Hashset agregue una sola de ellas, que es lo que necesita el usuario, sería que tanto Equals como GetHashCode retornen lo mismo en ambos casos. Luego alguien querría utilizar alguno de estos métodos para la información en los nodos del árbol y obtendría un comportamiento erróneo. Al disponer de la nueva jerarquía, la solución fue implementarlos en la clase Query. Si bien podrían haberse sobrescrito en Choice, no fue necesario, pero, con la decisión tomada, se deja preparado el código en caso de necesitar refinarse el comportamiento.

3.8.3 QUERI

La frase, o sea la pregunta concreta, ahora se calcula en el constructor y se almacena en un atributo. La complejidad espacial es prácticamente la misma, ya que la implementación original, de todos modos, guardaba el formato, ósea la pregunta sin el atributo (negro, Francia, etc). El beneficio se da respecto a la complejidad temporal, ya que el método ToString() simplemente retorna el valor de un atributo.

3.8.4 CHOICE

El método coincide(), renombrado como Answer(), ahora le pertenece a Choice. Mediante un simple cambio, se logró un resultado significativo, no más preguntas del estilo “¿No tiene ojos

azules? Ahora las preguntas son más claras y por tanto sus respuestas son intuitivas, ya sea un “no”, o un “si”.

3.8.5 RESULT

Como se mencionó anteriormente, DecisionData se reorganizó. Result es la clase que representa el contenido de un nodo hoja, o sea, el personaje que la máquina estipula que el usuario puede tener.

Aprovechando el paradigma orientado a objetos, se le dio el comportamiento a Result de poder elegir al azar uno de sus elementos. Esto, anteriormente, se realizaba en la clase Backend.

Como se explicó anteriormente, en la sección “Probabilidad vs Ganancia”, ahora tenemos al clon de Alex. Se crearon las imágenes correspondientes y se observa que comparten una hoja, cada uno con el 50% de probabilidad.

La implementación original no formateaba adecuadamente múltiples valores a la hora de imprimir el contenido de una hoja. Eso se soluciono, y, además, al igual que con la clase Query, se modificó el código para que el cálculo se realice una única vez, ganando complejidad temporal en desmedro de complejidad espacial. Respecto al formato, se imprimen entre paréntesis con la correspondiente probabilidad, o sea, la compartida por todos los personajes en ella.

Respecto al algoritmo que calcula la probabilidad de cada personaje, se lo adapto al uso particular de juego. Anteriormente se lo calculaba para cada uno, sin embargo, como no se repiten los personajes, es innecesario, ya que la probabilidad es equitativa. Además, ésta se calcula únicamente en el constructor y se almacena en un atributo para ser consultado en O(1) las veces que sea necesario.

Al no haber personajes repetidos, no es necesario almacenar la ocurrencia de cada uno, y por lo tanto se ganó complejidad espacial al reemplazar el diccionario por un HashSet.

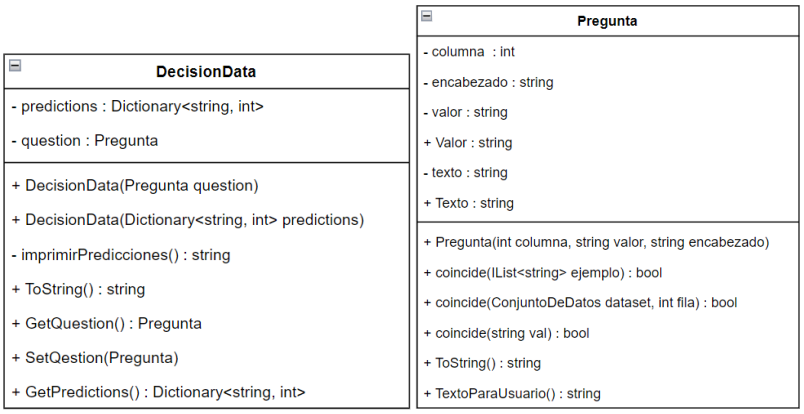


Tabla 1. Diuagramas UML de la clase DecisionData y Pregunta

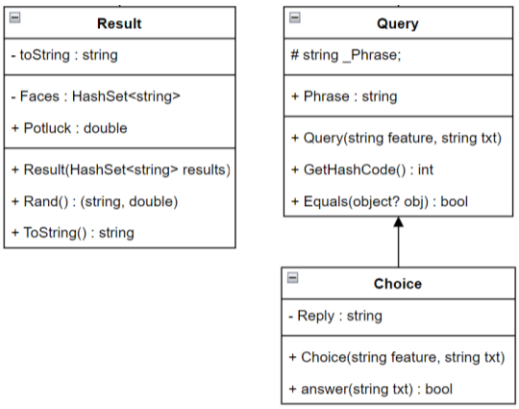


Ilustración 7. Diagramas UML de las clases, viejas y nuevas, vinculadas a la información almacenada por cada nodo del árbol.

3.9 PARSER

Como su nombre lo indica, esta clase provee métodos para recuperar la información de un archivo, en particular preguntas.csv.

La primera línea contiene los formatos de las preguntas, y las otras, los datos de los personajes. De este modo, la información se guarda en listas de string, una por cada renglón, quedando las preguntas en el atributo encabezado y las demás en una lista de listas.

3.9.1 SUGERENCIAS

ParseHeadAndTail() solo se usa en el constructor de *ConjuntoDeDatos*. Este método retorna un iterador, y luego, en dicho constructor se lo emplea dentro de un foreach para copiar la información. Esto es muy poco eficiente.

Si bien, aborda la cuestión de los espacios en blanco, no se ocupa de los que se encuentran entre el final del valor y el separador.

ParseHeadAndTail() es demasiado extenso (63 líneas), todo dentro de un solo método lleno de condicionales, lo cual, además es menos eficiente que un switch.

3.9.2 NUEVA IMPLEMENTACIÓN

Se modificó completamente el código. El método ParseHeadAndTail(), renombrado a Parse(), se redujo en 40 líneas menos, agregando el comportamiento que permite evitar los espacios en blanco entre el separador y el siguiente dato. Se reemplazaron los condicionales por un switch y un método muy corto.

Se decidió seguir permitiendo los comentarios dentro del archivo a parsear (entre comillas) pero se sacrificó la posibilidad de parsear con separadores elegidos dinámicamente. Esto es porque, primero no es necesario ese comportamiento, y segundo, de este modo se puede emplear un switch. Para que sea simple de modificar el delimitador se lo colocó como un atributo constante.

El nuevo método retorna las estructuras que se esperan en RawData, evitando el iterador antes descrito, por lo tanto, es más eficiente.

Además, se eliminaron el resto de los métodos que no brindaban utilidad significativa, de hecho, bastó con dos simples líneas, muy básicas, en RawData para reemplazarlos.

El comportamiento establecido para el caso de que falte un valor, o sea, dos comas (delimitador) seguidas, fue agregar un string vacío.

Con algunos pequeños cambios se puede obtener un vector para el encabezado y listas para las columnas. Esto es cómodo, ya que las comparaciones se realizan por columnas. Sin embargo, se optó por mantener las listas por filas ya que Spliter trabaja separando filas. Y, con una sintaxis un poquito más larga (matricial), se puede trabajar en todo el proyecto.

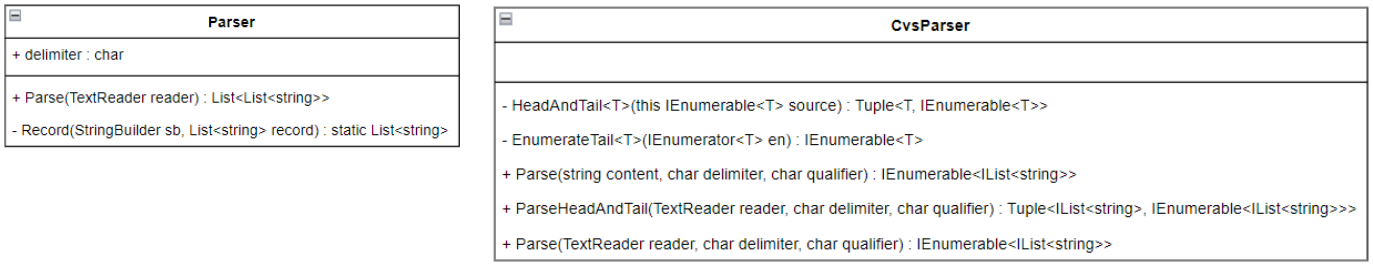


Ilustración 8. Diagramas UML de las clases encargadas de parsera el archivo preguntas.csv

3.10 MACHINE

La clase Backend es el juego en sí mismo, almacena el árbol, y en función de cómo sean llamados sus métodos, el juego se va desarrollando. Además es quien realmente da inicio al juego, eligiendo al personaje de la computadora, gestionando la creación del árbol y preparando la lista de preguntas que el usuario puede realizar.

3.10.1 SUGERENCIAS

Es una clase que únicamente provee métodos estáticos, no es que esté mal, solo que, nuevamente, no se saca provecho del paradigma orientado a objetos. Por ese motivo es que conceptualmente es demasiado amplia y al mismo tiempo no es lo suficientemente basta. En otras palabras, por ejemplo, brinda comportamiento que no concierne a la abstracción jugador (PC) porque es demasiado amplia. Pero el frontend se ocupa de una parte de la lógica del juego, de hecho, es quien toma realmente las decisiones que dan rumbo a la partida, justamente, porque la clase Backend no es lo suficientemente basta. La interfaz de usuario, o sea, cada formulario (ventana), debería ser tan simple como fuera posible, solamente informar al usuario de lo que hace la máquina y viceversa, en conjunto con alguna animación o efecto netamente visual.

Se usan atributos de clase solamente en el método initialize(), por ejemplo, pregunta_seleccionada, deberían ser variables locales.

3.10.2 NUEVA IMPLEMENTACIÓN

La clase se renombró para reflejar su nuevo comportamiento, es ahora la máquina, o sea, un jugador. Su estado interno tiene el árbol actual con las preguntas que hará, una lista con las respuestas que debe darle al usuario ante cada una de sus interrogantes y el nombre del personaje

elegido. En función a esto, sabe preguntar, responder, arriesgar, así como, elegir y revelar a su personaje.

Buscando la eficiencia, se optó por evitar crear nuevamente la clase, por ellos se almacena la información parseada, o sea, el objeto RawData sin particionar. También se mantiene una referencia a la raíz del árbol. Como es una clase concreta, sin métodos estáticos, podría haber implementado el patrón singleton, pero no se consideró que fuera necesario, sobre todo gracias a la claridad y funcionalidad de la clase Wiw que se explicará más adelante.

Como las consultas 1,2 y 3 del enunciado, no son más que preguntas a la "conciencia" del jugador, o sea, sería como preguntarle que piensa hacer, se decidió que la clase brinde métodos de instancia para acceder a los del árbol en cuestión. Esto permite encapsular la estructura interna, o sea, nadie accede realmente al árbol. Refinando el planteo, incluso, si bien los métodos tienen nombre que dan a entender que se utiliza un árbol, de todos modos, la naturaleza de las consultas por sí mismas ya lo sugieren.

Por cuestiones de complejidad temporal, se aprovecha el algoritmo que calcula las respuestas, para determinar, en simultáneo, todas las posibles preguntas del usuario que las originan. Esta última información no se almacena, si no que se retorna en el método StartGame(), que además elige un personaje al azar. Dicha decisión de diseño se puso en duda, y no fue hasta que se decidió como implementar el frontend y su relación con el backend, que se vió justificada esta metodología. El problema radica en el hecho de que, desde que se crea la instancia hasta que se inicia el juego (invocando a StartGame()), hay un lapso en el cual es posible llamar a métodos cuya funcionalidad no aplica, lo cual obliga a colocar condicionales y prever excepciones. Esto se solucionaría si se iniciara el juego al instanciar el objeto. Sin embargo, no podría regresar a la lista de preguntas. Es cierto que una opción sería almacenarlas en un atributo de instancia, pero conceptualmente existía la necesidad de encontrar un modo diferente de resolverlo. Con más escrutinio, podemos ver que este inconveniente es inherente a la clase misma, o sea, por ejemplo, no tiene sentido pedir que revele su personaje si todavía no terminó el juego, así que tampoco es realmente algo que salga de lo esperado.

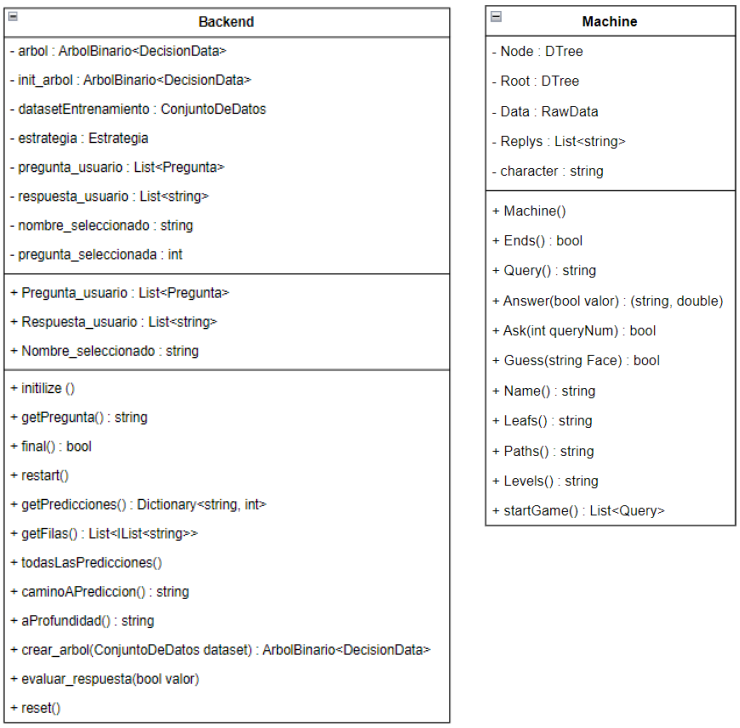


Tabla 2. Diagramas UML de las clases Backend y Machine

Si se observan los métodos se verá que la funcionalidad es muy similar. Por eso anteriormente se utilizó la frase “conceptualmente basta”, o sea, las abstracciones son diferentes, porque su razón de ser es distinta. Este concepto permite un diseño más limpio, con posibilidad de modularizar y crecer .

3.11 DTree

La clase ArbolBinario en sí misma no tenía mayor comportamiento implementado, ni tampoco era necesario. El trabajo con árboles, núcleo del proyecto, estaba estipulado, en el código original, estuviese dentro una clase llamada Estrategias. En la misma se creaba al árbol y atendía las tres consultas del enunciado en métodos estáticos.

3.11.1 Nueva implementación

El método de creación del árbol se convirtió en el constructor del mismo, que además dispone de otro privado. De este modo, instanciar un árbol, para los usos del proyecto, significa instanciar el árbol de decisión completo.

Las mejoras realizadas sobre la implementación original, permitieron obtener un resultado mucho más limpio en general, tal es el caso de la implementación del mencionado constructor. La lógica en sí es la misma, pero la sintaxis es más clara.

Aprovechando el hecho de que ningún nodo padre tiene un único hijo, se cambió el típico método EsHoja() por ChildLess().

Las tres consultas ya no reciben un árbol, porque justamente, el objeto en si mismo, o sea this, es el árbol.

Se aprovechó que los métodos debían devolver strings, obteniendo un gran resultado. Después de varias mejoras, se logró simplificarlos al punto tal de estar constituidos tan solo por unas pocas líneas de código. Y, aun así, brindar tanta funcionalidad como antes, formateando la salida en un modo amigable para el usuario.

3.11.2 Leafs

La consulta 1 fue renombrada Leafs() ya que es un nombre corto que sintetiza su funcionalidad. Devuelve un string con todos los nodos hojas realizando un recorrido preorden. Básicamente, retorna el dato hoja, o, de lo contrario, hace el llamado recursivo con su hijo izquierdo y luego con el derecho.

Debido al modo en que se implementó el método ToString() en la clase Choice, solo se imprime la probabilidad de acierto para los nodos con más de un candidato potencial, o sea, para Alex y su clon Alexander, presentados anteriormente.

Como se mencionó anteriormente, sacando provecho de lo mencionado, se redujo la cantidad de líneas de código empleando ChildLess() y la sintaxis propia del manejo de strings que provee el lenguaje de programación empleado.

```
public string Leafs()
{
    if (ChildLess())
        return Data.ToString()!;
    return Left!.Leafs() + "\n" + Right!.Leafs();
}
```

Ilustración 9. Implementación de la consulta 1

3.11.3 LEVELS

La consulta 3 pasó a llamarse Levels(). Para poder distinguir el comienzo y fin de cada uno, se analizó una variante del recorrido entre niveles.

Debido al agregado estético de caracteres, el método se volvió demasiado extenso, probablemente más de 30 líneas de código. Por ese motivo se perfeccionó el algoritmo en una versión más escueta, que a su vez permite modularizarlo en distintos métodos.

3.11.3.1 CARACTERÍSTICAS

La primera diferencia, es que este método puede implementarse tanto de un modo recursivo, como iterativo, en una cantidad de líneas de código semejante.

La segunda, es que utiliza dos colas, aunque no por eso tiene peor complejidad espacial. Para el caso recursivo, que por definición implica más recursos, al emplearse con referencias y el método Dequeue(), se evita tener elementos repetidos.

Luego, tenemos el hecho de que para este nuevo modo de resolver el problema, no es necesario utilizar un separador, o sea, no hace falta encolar un null para diferenciar en niveles.

3.11.3.2 ITERADOR

El truco está en darse cuenta de que el último nodo de un nivel a mostrar es también el último en encolar sus hijos. Dicho en otras palabras, cuando se termina de mostrar un nivel, también se finaliza de encolar el siguiente. Entonces, se creó un nuevo método Next(), que si bien tiene algunas pocas líneas, es de mucha utilidad. El mismo desencola, agrega hijos si es necesario y retorna un dato. Al abstraer el comportamiento de este modo, se puede recorrer la cola como si se lo estuviera haciendo con un iterador. Como el iterador sería de cola, pero debe trabajar con árboles binarios, se decidió que sea un método privado del árbol, auxiliar de Levels().

En conclusión, se imprime un nivel completo en cada iteración o llamado recursivo (por ello se agrega el contador de nivel como parámetro). Es gracias a esto, que no es necesario hacer uso de separadores.

3.11.3.3 MANEJO DE STRINGS

Cabe mencionar, que se aprovecha el atributo Count, de la cola, para formatear la salida de un modo más amigable para el usuario. Analizando el hecho de que todo padre tiene dos hijos, se puede observar que cuando el contador es par, estamos en presencia de un hijo izquierdo y viceversa. Así, se llega a la conclusión de que los elementos se pueden separar con comas y pipes alternadamente.

Como siempre se termina en cantidad par, se puede comenzar la secuencia con un “ \n | “ , distinguiendo así la etiqueta de nivel, englobando entre pipes la secuencia de valores y desligando

el salto de línea de la sentencia return. Es un modo barato y estético de resolver el último carácter de cada línea.

Por último, aprovechando la simpleza del llamado recursivo, se coloca un operador condicional " ? " de un modo intuitivo: Nivel + Enter [+ PróximoNivel]

```
private string _Levels(Queue<DTree> q, int n)
{
    Queue<DTree> c = new();
    string s = "Nivel: " + n + "\n  | ";
    while (q.Count > 0)
        s += Next(q, c) + (q.Count % 2 == 1 ? " , " : " | ");
    return s + "\n" + (c.Count > 0 ? _Levels(c, ++n) : "");
}
```

Ilustración 10. Algoritmo recursivo para el recorrido por niveles

```
public string Levels()
{
    Queue<DTree> a = new();
    a.Enqueue(this);
    return _Levels(a, 0);
}
```

```
private IData Next(Queue<DTree> a, Queue<DTree> b)
{
    DTree t = a.Dequeue();
    if (!t.ChildLess())
    {
        b.Enqueue(t.Left!);
        b.Enqueue(t.Right!);
    }
    return t.Data;
}
```

Ilustración 11. Métodos complementarios de _Levels()

3.11.4 PATHS

La consulta 2 se renombró, por los mismos motivos que las anteriores, su nombre es Paths(). Retorna un string con todos los caminos desde la raíz hasta las hojas.

En este caso también es un recorrido preorden, con la diferencia de que se van anexando al resultado final, tanto las preguntas de los nodos padres, como sus respuestas. Esto último, si bien no era necesario, permite un resultado final mucho más fácil de interpretar, es muy útil a la hora de analizar cada camino.

Este método se simplificó mucho desde su implementación original, manteniendo el mismo comportamiento. La primera implementación requería de dos métodos, que ya no son necesarios. El resultado final es muy simple e intuitivo, pero su creación obligó a romper algunas estructuras, modos de ver las cosas, que se habían vuelto una traba para avanzar.

Además, es más eficiente, porque trabajar con listas, como originalmente fue pensado, implica armarlas y luego recorrerlas para reconstruir el string. Lo mismo sucede respecto a la complejidad espacial, es más barato guardar un string, que almacenar nodos que los contienen.

```
public string Paths()
{
    return _Paths("");
}
```

```
public string _Paths(string c)
{
    string d = c + Data;
    if (ChildLess())
        return d + "\n";
    return Left!._Paths(d + " | si | ") +"\n"+ Right!._Paths(d + " | no | ")
}
```

Ilustración 12. Implementación de la consulta 2.

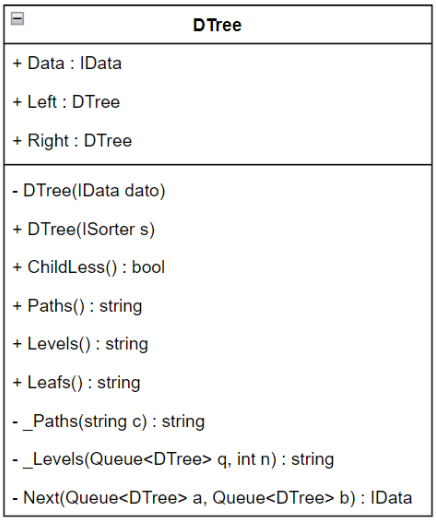


Ilustración 13. Diagrama UML de la clase que modela el árbol de decisión.

3.12 FILEMGR

La clase Utils, brinda un mecanismo para centralizar los path y nombres de archivos de todo el proyecto. Además guarda alguna que otra información de metadata.

3.12.1 SUGERENCIAS

La clase solo brindaba métodos static por lo tanto no se saca provecho del paradigma orientado a objetos. Los usuarios invocan al método init_patron() porque no saben si la clase está inicializada. Por sobre esto, el método no ofrece nada, porque el atributo de clase se inicializa por defecto.

El path default para la carpeta dataset es erróneo, obligando al usuario a cargar a mano la ubicación al comenzar el juego indefectiblemente, lo cual es bastante tedioso.

Los atributos w y h guardan un entero que representa la dimensión de las imágenes. Si bien la idea está bien, en la práctica solo se usa en un método, que a su vez tiene una versión muy similar en otra clase que no emplea Utils porque necesita una escala diferente. No parece ser la mejor solución al problema.

El método get_path_preguntas() se calcula cada vez que es invocado, no porque realmente sea necesario, si no que se asocia a la cuestión comentada para init_patron().

3.12.2 NUEVA IMPLEMENTACIÓN

Se convirtió a la clase en un Singleton, de este modo es accesible por todos, no se repite información, y se aprovecha las ventajas del paradigma orientado a objetos.

La clase se posiciona automáticamente en la ubicación del dataset como referencia del directorio actual.

Los métodos no retornan path, si no, directamente archivos o imágenes, o sea, instancias de Image y FileStream.

Ya no se pide una carpeta, porque la nueva ventana no permite ver los archivos que hay en ella. Se pide directamente el archivo que falta y se busca los demás en ese directorio.

Provee un mecanismo seguro para cargar los archivos, en caso de no encontrarlos brinda al usuario la oportunidad de elegir otra ubicación para recuperarse del problema. Tiene tantos intentos como desee, si decide cancelar la operación se lanza una excepción. Para no repetir código, se emplean funciones lambda.

Las imágenes se encuentran a un nivel de profundidad mayor en la jerarquía de carpetas respecto a preguntas.csv. Por lo tanto, en caso de que no se encuentre el archivo, es necesario editar el path original, para que se posicione en la carpeta elegida por el usuario, sin avanzar en profundidad. Si bien se podría colocar todo en el mismo directorio, por motivos didácticos, se aceptó el desafío que esto planteaba.

Dicho en otras palabras, el usuario no tiene porque conocer la jerarquía de carpetas, le debería bastar con indicar la ubicación exacta de la imagen. Para resolver este inconveniente, la función lambda emplea un contador de errores, de modo que al incurrir en al menos un problema se limpia el path, computando exactamente el ingresado por el usuario.

Como el contador es una variable local del método que define la función lambda, se reinicia con cada invocación al método, pero no con cada llamado a la función lambda dentro del bucle de intentos por encontrar el archivo.

Sin embargo, el path es un atributo de instancia, entonces este mecanismo permite: por un lado, memorizar que el path ya no tiene porque cumplir con la jerarquía original de carpetas en las posteriores invocaciones al primer método, y, además, que, si todas las imágenes están en la misma carpeta, no sea necesario pedir de nuevo el dato al usuario.

Lo que no hace, es recordar más de un path. De modo que, si las imágenes están repartidas en varias carpetas, dependerá del orden en que sean buscadas, si es que se le solicitará, o no, al usuario indicar el antiguo path.

Como se contempla el caso en que las imágenes están muy dispersas, se emplea un formulario guardado en el atributo bringToFront que permite que los nuevos mensajes se muestren siempre en primer plano.

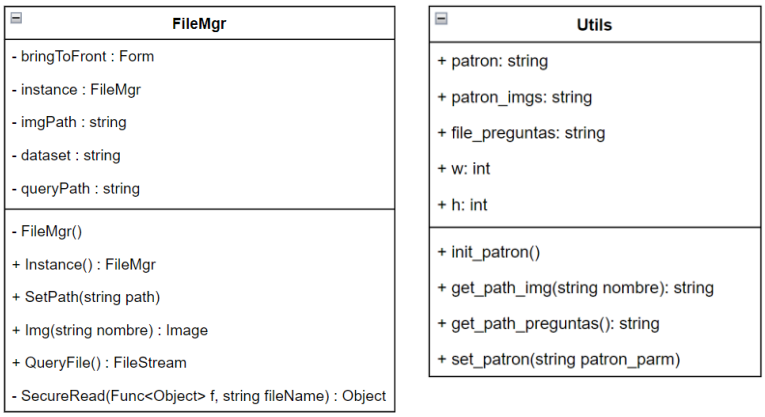


Ilustración 14. Diagrama UML de Utils , la clase original que retornaba paths, y de la clase nueva FileMgr , que retorna archivos.

4. FRONT-END

Esta es la parte del proyecto que, si bien dejó muchísimo a nivel personal y profesional, más se aleja de los objetivos del presente. Por ese motivo se dará un panorama completo, pero sin especificar con tanto detalle los cambios implementados.

Requirió de investigación, ya que no se habían cursado materias que trataran estos temas. Partiendo de la base adquirida en Metodologías de Programación 1, se comenzó la búsqueda de algún patrón que nos permita una base sólida sobre la cual empezar a construir. La investigación se vió recompensada al encontrar lo que se denomina “Patrones de Arquitectura”. En particular, se optó por utilizar el “Patrón MVP”, haciendo uso del “Principio de inversión de dependencia”.

Como el cambio fue a nivel global, más bien un cambio de paradigma por así llamarlo, tampoco se consideró pertinente dar las sugerencias particulares, como se realizó anteriormente, para cada clase.

En línea general se optó por dar al usuario el modo más práctico en cada caso. Por ejemplo, no se pregunta al usuario si está seguro al elegir un personaje, si no que se colocó un botón “Nuevo juego” , que además le brinda una funcionalidad extra. Lo mismo sucede a la hora de elegir los personajes, seleccionar un archivo, o incluso el botón “Arriesgar”, que está activo todo el tiempo.

4.1 PATRÓN DE ARQUITECTURA MVP

A grandes rasgos contamos con tres abstracciones principales, cuyas siglas componen el nombre del patrón: Modelo, Vista y Presentador. En la bibliografía consultada el modelo era una base de datos y la vista la interfaz que permitía al usuario trabajar con la información. Por su parte el presentador se ocupaba de intermediar entre las partes.

Este concepto se adaptó al proyecto, la vista, o sea, los form, únicamente se encargan de mostrar lo que el presentador le pide y dar aviso de las acciones o decisiones que toma el usuario. Para implementar la comunicación entre ambos se empleó el patrón Observer y el principio de inversión de dependencias. Esto significa que, el presentador (observador) se suscribe a los eventos de la vista (observable). Y, además, el presentador no conoce a la vista directamente, si no que trabaja con una interfaz. Por consiguiente, los formularios Board, FacesForm, GameOver y Output, implementan, cada uno, su correspondiente interfaz.

La vista, por su parte, no es observadora del presentador, si no que, en este caso, la interfaz brinda métodos cuyo objetivo es indicar a la vista que muestre algo.

El hecho de utilizar interfaces permite desacoplar completamente el frontend del backend. De hecho, se puede implementar una interfaz gráfica completamente distinta y emplear el mismo backend.

Respecto al modelo, en nuestro caso, tenemos dos, o al menos se tomó la decisión de diseño de conceptualizarlo de esa forma. El primero está representado en RawData, o sea, la información del archivo parseada y estructurada convenientemente. El segundo, no tan obvio, es la clase Machine, cuya abstracción, al final de cuentas, no es más que otro modo de organizar y gestionar la misma información.

Es cierto que podría haberse pensado que Machine es quien controla y decide como se muestran los datos, mientras que el árbol es la estructura de datos que almacena la información. Lo cual es cierto, sin embargo, si se analiza cuidadosamente, se estaría incurriendo en el mismo

problema que al principio, aquel que presentaba la clase Backend. Este concepto se describe en la siguiente sección, ya que toma forma con una nueva abstracción, oculta entre líneas dentro de la implementación original.

```
public interface BoardView : FormView
{
    2 references
    public int SelectedQuery { get; }

    public event EventHandler BtnYes;
    public event EventHandler BtnNo;
    public event EventHandler BtnPaths;
    public event EventHandler BtnLevels;
    public event EventHandler BtnLeafs;
    public event EventHandler BtnGuess;
    public event EventHandler BtnAsk;
    public event EventHandler BtnClose;
    public event EventHandler BtnNewGame;

    2 references
    public void UserTurn(string pcQuery);
    3 references
    public void PcTurn();
    2 references
    public void PcReply(bool reply);
}
```

Ilustración 15. Ejemplo de interfaz de vista. Dispone de getters (y setters en otros casos) , eventos (patrón Observer) y métodos con el propósito exclusivo de dar la indicación a la vista de que muestre algo.

4.2 WiW, otra clase del Backend

Uno de los inconvenientes más claros de la implementación original era el modo en que se creaban, cerraban y ocultaban los form. Tal es el hecho, que había un bug relacionado. Si el usuario intentaba adivinar el personaje, pero se arrepentía, tenía dos opciones: la primera era cerrar la ventana que daba a elegir una de las caras, y, la segunda, hacer una elección y arrepentirse, o sea, responder que no al mensaje emergente. En el primer caso, se cierra abruptamente la aplicación. Si bien podría argumentarse que era el comportamiento deseado, teniendo en cuenta la cantidad de mensajes que se generaban al principio del juego, no guarda mucha correlación. La segunda, es peor, desaparecen las ventanas, pero en realidad la aplicación sigue corriendo. La línea 165 de FormUser (comentada) da entender que el problema ya era conocido por el desarrollador

Más allá del bug, era muy difícil de seguir el código, ya que, indirectamente, el orden en que se administran estas operaciones, guarda estrecha relación con la lógica del juego. De algún modo, la información de cada estado de la partida está relacionada con las mismas.

La segunda mención, es que el frontend llamaba a los métodos de Backend para tomar las decisiones de la máquina, pero, las consecuencias de las mismas, y, por consiguiente, la dinámica del juego, quedaban a manos de los form; disimuladas por eventuales condicionales.

4.2.1 Nueva implementación

La clase Wiw es el presentador. Su función es controlar y comunicar la vista con el modelo, o sea, es el juego en sí mismo. Dicho de otro modo, centraliza el orden en que deben suceder los acontecimientos. Determina las consecuencias e informa a cada jugador lo que corresponda.

Esta clase instancia cada uno de los objetos, y, por eficiencia, los almacena en atributos. De este modo solo se necesita instanciarlos por única vez (a excepción de Board) todos están pensados para corresponder con dicho comportamiento, incluso ante una nueva partida.

Además, es observadora de todas las vistas. Es mediante la administración de los eventos que es posible hacer que una misma ventana se comporte de diferentes maneras dependiendo de la situación. En esta implementación, por ejemplo, si se decide arriesgar por un personaje, al cerrar la ventana de selección se retorna al juego, mientras que ese mismo botón en un contexto distinto, como a la hora de comenzar una nueva partida, cierra la aplicación si el usuario decide no elegir ningún personaje para jugar.

Al estar toda esta lógica en un mismo lugar, es mucho más simple evitar bugs y agregar nuevas modificaciones. Un ejemplo es el botón “Nuevo juego” que se agregó. Bastó con poner el evento en la interfaz y agregar una línea en la vista (muy similar a las demás, pero con el nuevo nombre). Luego se suscribe una función lambda que simplemente llama al método correspondiente, que en este caso ya estaba implementado. Del mismo modo se podría cambiar el comportamiento de los demás botones del proyecto.

Esto es especialmente útil, ya que las ventanas que se ocultan, se pierden. Es fácil ocultarla en un método, y luego instanciarla en otro. El resultado es que cada vez se ocupa más memoria con ventanas que nunca liberan sus recursos.

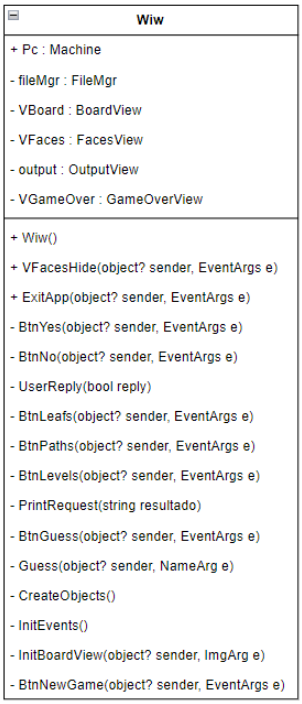


Ilustración 16. UML de la clase Wiw

5. Conclusión

Los árboles son una herramienta muy útil y al mismo tiempo sencilla. Lograr que la máquina tome decisiones por sí misma es algo realmente fascinante, y la simpleza con la que se resuelve el problema lo es aún más.

Los algoritmos vistos en la materia, así como la práctica que se adquirió al realizar los ejercicios, me brindaron un punto de vista y capacidad analítica que antes no tenía. Sigo

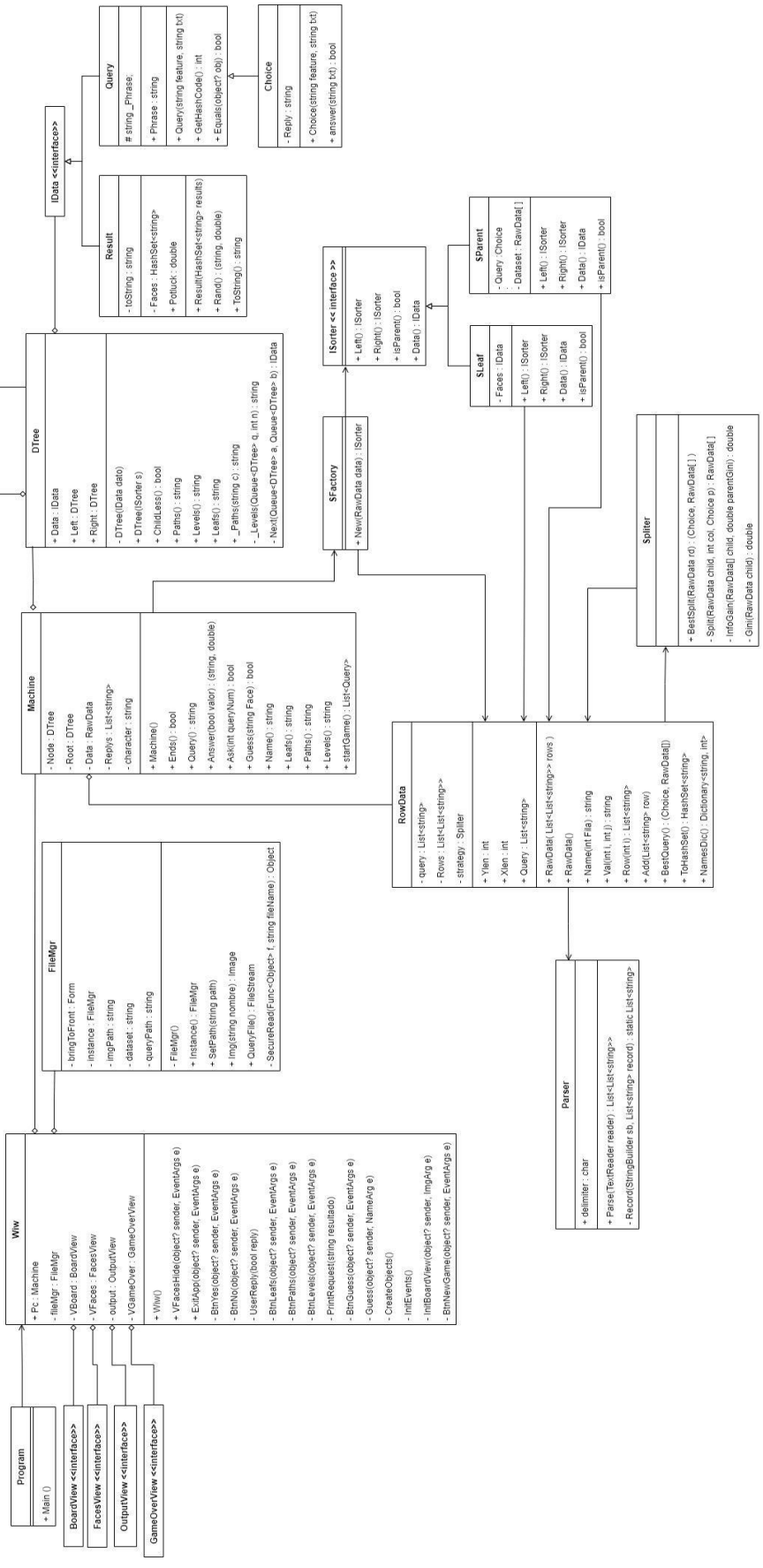
impresionado con el modo en que se pudo resolver, sobre todo la consulta 3, que en un principio era demasiado larga. Analizar, comprender y mejorar algoritmos como el que parseaba el archivo, me permitieron notar el avance, que, sin darme cuenta, había logrado gracias a lo que estudiamos en la materia.

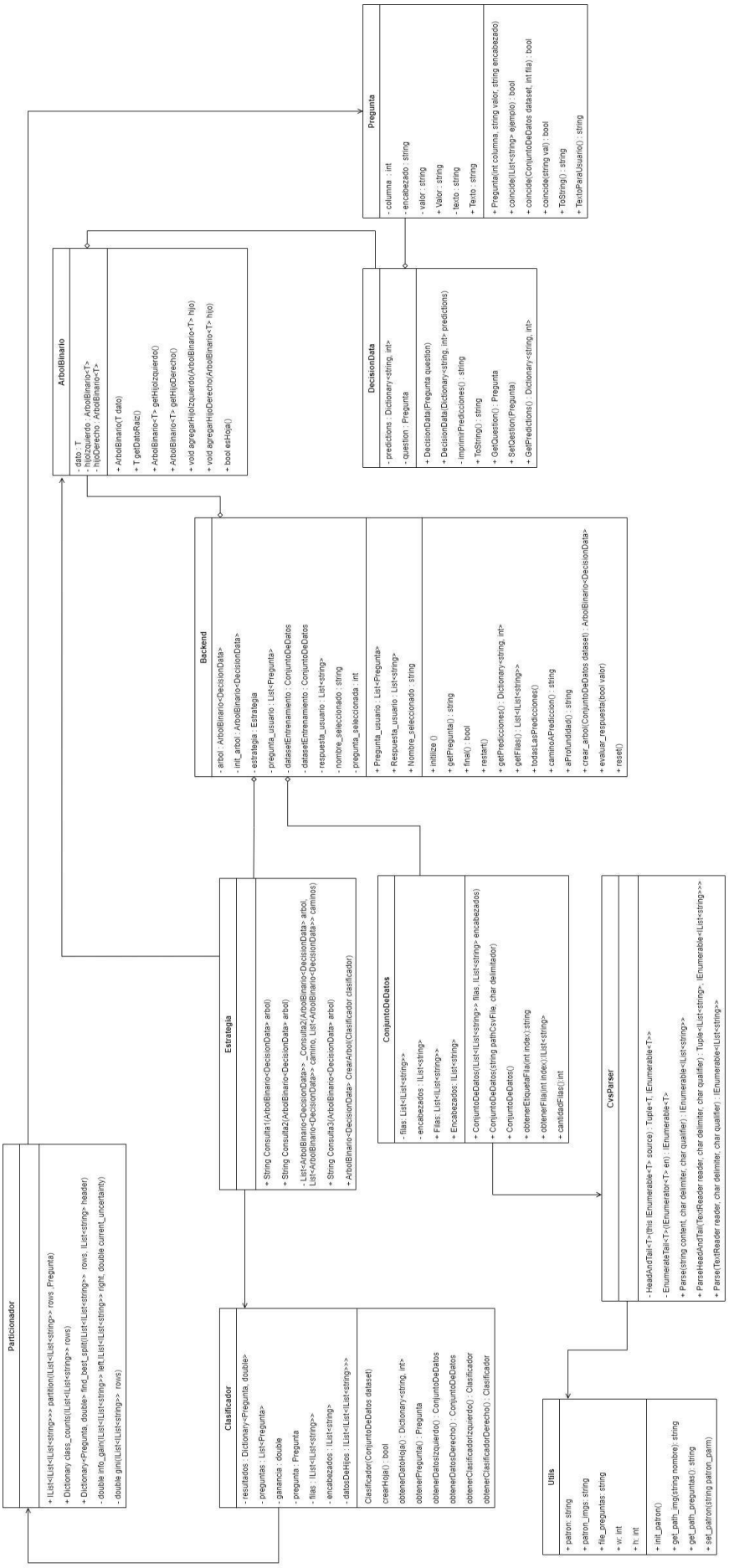
Los patrones, ya sean de diseño, los algoritmos o incluso los de arquitectura, también son una herramienta muy útil. Comprender o asignar un significado conceptual a cada parte de un proyecto es fundamental a la hora de buscar un resultado limpio y consistente con estos patrones.

Una conclusión que me fue muy útil, es que todos estos modos de resolver problemas, no vienen a limitar la creatividad, por el contrario, la potencian, ya que son los cimientos sólidos sobre los cuales empezar a construir, ya sea, algoritmos, clases o proyectos.

6. ANEXO

Diagramas UML completos en la siguiente página.


Program
+ Main ()
BoardView <<interface>>
FacesView <<interface>>
OutputView <<interface>>
GameOverView <<interface>>
Wiw
+ Pc : Machine
- fileMgr : FileMgr
- VBoard : BoardView
- VFaces : FacesView
- output : OutputView
- VGameOver : GameOverView
+ Wiw()
+ VFacesHide(object? sender, EventArgs e)
+ ExitApp(object? sender, EventArgs e)
- BtnYes(object? sender, EventArgs e)
- BtnNo(object? sender, EventArgs e)
- UserReply(bool reply)
- BtnLeafs(object? sender, EventArgs e)
- BtnPaths(object? sender, EventArgs e)
- BtnLevels(object? sender, EventArgs e)
- PrintRequest(string resultado)
- BtnGuess(object? sender, EventArgs e)
- Guess(object? sender, NameArg e)
- CreateObjects()
- InitEvents()
- InitBoardView(object? sender, ImgArg e)
- BtnNewGame(object? sender, EventArgs e)
FileMgr
- bringToFront : Form
- instance : FileMgr
- imgPath : string
- dataset : string
- queryPath : string
- FileMgr()
+ Instance() : FileMgr
+ SetPath(string path)
+ Img(string nombre) : Image
+ QueryFile() : FileStream
- SecureRead(Func<Object> f, string fileName) : Object
Machine
- Node : DTree
- Root : DTree
- Data : RawData
- Replys : List<string>
- character : string
+ Machine()
+ Ends() : bool
+ Query() : string
+ Answer(bool valor) : (string, double)
+ Ask(int queryNum) : bool
+ Guess(string Face) : bool
+ Name() : string
+ Leafs() : string
+ Paths() : string
+ Levels() : string
+ startGame() : List<Query>
DTree
+ Data : IData
+ Left : DTree
+ Right : DTree
- DTree(IData dato)
+ DTree(ISorter s)
+ ChildLess() : bool
+ Paths() : string
+ Levels() : string
+ Leafs() : string
+ _Paths(string c) : string
- _Levels(Queue<DTree> q, int n) : string
- Next(Queue<DTree> a, Queue<DTree> b) : IData
SFactory
+ New(RawData data) : ISorter
ISorter << interface >>
+ Left() : ISorter
+ Right() : ISorter
+ isParent() : bool
+ Data() : IData
SLeaf
- Faces : IData
+ Left() : ISorter
+ Right() : ISorter
+ Data() : IData
+ isParent() : bool
SParent
- Query : Choice
- Dataset : RawData[ ]
+ Left() : ISorter
+ Right() : ISorter
+ Data() : IData
+ isParent() : bool
Splitter
+ BestSplit(RawData rd) : (Choice, RawData[ ])
- Split(RawData child, int col, Choice p) : RawData[ ]
- InfoGain(RawData[][] child, double parentGini) : double
- Gini(RawData child) : double
RowData
- query : List<string>
- Rows : List<List<string>>
- strategy : Spliter
+ Ylen : int
+ Xlen : int
+ Query : List<string>
+ RawData(List<List<string>> rows)
+ RawData()
+ Name(int Fila) : string
+ Val(int i, int j) : string
+ Row(int i) : List<string>
+ Add(List<string> row)
+ BestQuery() : (Choice, RawData[])
+ ToHashSet() : HashSet<string>
+ NamesDic() : Dictionary<string, int>
Parser
+ delimiter : char
+ Parse(TextReader reader) : List<List<string>>
- Record(StringBuilder sb, List<string> record) : static List<string>
IData <<interface>>
Query
string _Phrase;
+ Phrase : string
+ Query(string feature, string txt)
+ GetHashCode() : int
+ Equals(object? obj) : bool
Result
- toString : string
- Faces : HashSet<string>
+ Potluck : double
+ Result(HashSet<string> results)
+ Rand() : (string, double)
+ ToString() : string
Choice
- Reply : string
+ Choice(string feature, string txt)
+ answer(string txt) : bool


Particionador
+ IList<IList<IList<string>>> partition(IList<IList<string>> rows ,Pregunta)
+ Dictionary class_counts(IList<IList<string>> rows)
+ Dictionary<Pregunta, double> find_best_split(IList<IList<string>> rows, IList<string> header)
- double info_gain(IList<IList<string>> left,IList<IList<string>> right, double current_uncertainty)
- double gini(IList<IList<string>> rows)
Clasificador
- resultados : Dictionary<Pregunta, double>
- preguntas : List<Pregunta>
- ganancia : double
- pregunta : Pregunta
- filas : IList<IList<string>>
- encabezados : IList<string>
- datosDeHijos : IList<IList<IList<string>>>
Clasificador(ConjuntoDeDatos dataset)
creaHoja() : bool
obtenerDatosHoja() : Dictionary<string, int>
obtenerPregunta() : Pregunta
obtenerDatosIzquierdo() : ConjuntoDeDatos
obtenerDatosDerecho() : ConjuntoDeDatos
obtenerClasificadorIzquierdo() : Clasificador
obtenerClasificadorDerecho() : Clasificador
Estrategia
+ String Consulta (ArbolBinario<DecisionData> arbol)
+ String Consulta2(ArbolBinario<DecisionData> arbol)
- List<ArbolBinario<DecisionData>> _Consulta2(ArbolBinario<DecisionData> arbol, List<ArbolBinario<DecisionData>> camino, List<ArbolBinario<DecisionData>> caminos)
+ String Consulta3(ArbolBinario<DecisionData> arbol)
+ ArbolBinario<DecisionData> CrearArbol(Clasificador clasificador)
ConjuntoDeDatos
- filas: List<IList<string>>
- encabezados : IList<string>
+ Filas: List<IList<string>>
+ Encabezados: IList<string>
+ ConjuntoDeDatos(IList<IList<string>> filas, IList<string> encabezados)
+ ConjuntoDeDatos(string pathCsvFile, char delimitador)
+ ConjuntoDeDatos()
+ obtenerElIquetaFila(int index):string
+ obtenerFila(int index):IList<string>
+ cantidadFilas():int
CvsParser
- HeadAndTail<T>(this IEnumerable<T> source) : Tuple<T, IEnumerable<T>>
- EnumerableTail<T>(IEnumerator<T> en) : IEnumerable<T>
+ Parse(string content, char delimiter, char qualifier) : IEnumerable<IList<string>>
+ ParseHeadAndTail(TextReader reader, char delimiter, char qualifier) : Tuple<IList<string>, IEnumerable<IList<string>>>
+ Parse(TextReader reader, char delimiter, char qualifier) : IEnumerable<IList<string>>
Utils
+ patron: string
+ patron_imgs: string
+ file_preguntas: string
+ w: int
+ h: int
+ init_patron()
+ get_path_img(string nombre): string
+ get_path_preguntas(): string
+ set_patron(string patron_parm)
ArbolBinario
- dato : T
- hijoIzquierdo : ArbolBinario<T>
- hijoDerecho : ArbolBinario<T>
+ ArbolBinario(T dato)
+ T getDatoRaiz()
+ ArbolBinario<T> getHijoIzquierdo()
+ ArbolBinario<T> getHijoDerecho()
+ void agregarHijoIzquierdo(ArbolBinario<T> hijo)
+ void agregarHijoDerecho(ArbolBinario<T> hijo)
+ bool esHoja()
Backend
- arbol : ArbolBinario<DecisionData>
- int_arbol : ArbolBinario<DecisionData>
- estrategia : Estrategia
- pregunta_usuario : List<Pregunta>
- datasetEntrenamiento : ConjuntoDeDatos
- datasetEntrenamiento : ConjuntoDeDatos
- respuesta_usuario : List<string>
- nombre_seleccionado : string
- pregunta_seleccionada :int
+ Pregunta_usuario : List<Pregunta>
+ Respuesta_usuario : List<string>
+ Nombre_seleccionado : string
+ init()ice ()
+ getPregunta() : string
+ final() : bool
+ restart()
+ getPredicciones() : Dictionary<string, int>
+ getFilas() : List<IList<string>>
+ todasLasPredicciones()
+ caminoAPrediccion() : string
+ aProfundidad() : string
+ crear_arbol(ConjuntoDeDatos dataset) : ArbolBinario<DecisionData>
+ evaluar_respuesta(bool valor)
+ reset()
DecisionData
- predictions : Dictionary<string, int>
- question : Pregunta
+ DecisionData(Pregunta question)
+ DecisionData(Dictionary<string, int> predictions)
- imprimirPredicciones() : string
+ ToString() : string
+ GetQuestion() : Pregunta
+ SetQuestion(Pregunta)
+ GetPredictions() : Dictionary<string, int>
Pregunta
- columna : int
- encabezado :string
- valor : string
+ Valor : string
- texto : string
+ Texto : string
+ Pregunta(int columna, string valor, string encabezado)
+ coincide(IList<string> ejemplo) : bool
+ coincide(ConjuntoDeDatos dataset, int fila) : bool
+ coincide(string val) : bool
+ ToString() : string
+ TextoParaUsuario() : string