Práctica 3: Megabot 3000 (parte 3)

Programación 2

Curso 2019-2020

Esta práctica consiste en implementar la Práctica 1 y alguna funcionalidad de la Práctica 2, siguiendo el paradigma de programación orientada a objetos, incluyendo algunos cambios en el funcionamiento que se detallan en este enunciado. Los conceptos necesarios para desarrollar esta práctica se trabajan en todos los temas de teoría, aunque especialmente en el *Tema 5*.

Condiciones de entrega

- La fecha límite de entrega para esta práctica es el **viernes 22 de mayo**, hasta las **23:59**
- La práctica consta de varios ficheros: `Example.cc`, `Example.h`, `Intent.cc`, `Intent.h`, `Chatbot.cc`, `Chatbot.h`, `Util.cc` y `Util.h`. Todos ellos se deberán comprimir en un único fichero llamado `prog2p3.tgz` que se entregará a través del servidor de prácticas de la forma habitual. Para crear el fichero comprimido debes hacerlo de la siguiente manera:

Terminal

```
$ tar cvfz prog2p3.tgz Example.cc Example.h Intent.cc Intent.h Chatbot.cc
Chatbot.h Util.cc Util.h
```

Código de honor

Si se detecta copia (total o parcial) en tu práctica, tendrás un **0** en la entrega y se informará a la dirección de la Escuela Politécnica Superior para que adopte medidas disciplinarias

Está bien discutir con tus compañeros posibles soluciones a las prácticas
Está bien apuntarte a una academia si sirve para obligarte a estudiar y hacer las prácticas

Está mal copiar código de otros compañeros para resolver tus problemas
Está mal apuntarte a una academia para que te hagan las prácticas

Si necesitas ayuda acude a tu profesor/a
No copies

Normas generales

- Debes entregar la práctica exclusivamente a través del servidor de prácticas del Departamento de Lenguajes y Sistemas Informáticos (DLSI). Se puede acceder a él de dos maneras:

    - Página principal del DLSI (`https://www.dlsi.ua.es`), opción “ENTREGA DE PRÁCTICAS”
    - Directamente en la dirección `https://pracdlsi.dlsi.ua.es`
- Cuestiones que debes tener en cuenta al hacer la entrega:
    - El usuario y la contraseña para entregar prácticas son los mismos que utilizas en UACloud
    - Puedes entregar la práctica varias veces, pero sólo se corregirá la última entrega
    - No se admitirán entregas por otros medios, como el correo electrónico o UACloud
    - No se admitirán entregas fuera de plazo
- Tu práctica debe poder ser compilada sin errores con el compilador de C++ existente en la distribución de Linux de los laboratorios de prácticas
- Si tu práctica no se puede compilar su calificación será 0
- La corrección de la práctica se hará de forma automática, por lo que es imprescindible que respetes estrictamente los textos y los formatos de salida que se indican en este enunciado
- Al comienzo de todos los ficheros fuente entregados debes incluir un comentario con tu NIF (o equivalente) y tu nombre. Por ejemplo:

Chatbot.h

```
// DNI 12345678X GARCIA GARCIA, JUAN MANUEL
...
```

- El cálculo de la nota de la práctica y su relevancia en la nota final de la asignatura se detallan en las transparencias de presentación de la asignatura (*Tema 0*)

1. Descripción de la práctica

En esta práctica se implementará de nuevo la Práctica 1, y algunas funcionalidades de la Práctica 2, utilizando en esta ocasión el paradigma de programación orientada a objetos. A las funcionalidades que ofrecía la primera práctica se añadirá la opción `Configure` y la distancia basada en n-gramas de la Práctica 2, pero se descarta toda la parte correspondiente al manejo de ficheros y argumentos del programa.

Gran parte del código que desarrollaste para la Práctica 1 y Práctica 2 puede ser reutilizado en esta ocasión, por lo que no debes empezar el trabajo desde cero sino tratar de adaptar el código existente y añadir el que consideres necesario para implementar el diseño propuesto.

2. Detalles de implementación

En el Moodle de la asignatura se publicarán varios ficheros que necesitarás para la correcta realización de la práctica:

- `Util.h` y `Util.cc`. El fichero `Util.h` contiene la definición de la clase `Util` y el tipo enumerado `Error`. Esta clase contiene un método `error` (implementado en `Util.cc`) que permite mostrar por pantalla los errores pasados como parámetro. Se han añadido nuevos errores a los que ya se daban en la Práctica 1 y 2. Estos ficheros tendrás que complementarlos con las nuevas funciones que se describen en la Sección 3.1
- `prac3.cc`. Fichero que contiene el `main` de la práctica, junto con el código para mostrar los menús de opciones del programa. Este fichero se encarga de crear los objetos de las clases implicadas en el problema, mostrar los menús e ir realizando las llamadas a los métodos correspondientes para replicar el funcionamiento de las prácticas anteriores utilizando en esta ocasión programación orientada a objetos. Este fichero no debe modificarse ni incluirse en la entrega

- `converter.o`. Fichero en código objeto (compilado para máquinas Linux de 64 bits) que contiene una función `cleanString` que recibe como parámetro una cadena de texto y devuelve esa misma cadena sin acentos, ni diéresis, ni eñes (consulta la Práctica 1, Sección 6.1.3, para más información)
- `prac3`. Fichero ejecutable de la práctica (compilado para máquinas Linux de 64 bits) desarrollado por el profesorado de la asignatura, para que puedas probarlo con las entradas que quieras y ver la salida correcta esperada
- `makefile`. Fichero que permite compilar de manera óptima todos los ficheros fuente de la práctica y generar un único ejecutable
- `autocorrector-prac3.tgz`. Contiene los ficheros del autocorrector para probar la práctica con algunas pruebas de entrada. Este autocorrector incluye todas las pruebas del corrector de la Práctica 1. La corrección automática de la práctica se realizará con un corrector similar, con estas pruebas y otras más definidas por el profesorado de la asignatura

En esta práctica cada una de las clases se implementará en un módulo diferente, de manera que tendremos dos ficheros para cada una de ellas: `Example.h` y `Example.cc` para los ejemplos, `Intent.h` e `Intent.cc` para las intenciones, `Chatbot.h` y `Chatbot.cc` para los *chatbot*, y `Util.h` y `Util.cc` para las funciones auxiliares. Estos ficheros se deben compilar conjuntamente para generar un único ejecutable. Una forma de hacer esto es de la siguiente manera:

Terminal

```
$ g++ -Wall Example.cc Intent.cc Chatbot.cc Util.cc prac3.cc converter.o -o prac3
```

Esta solución no es óptima, ya que compila de nuevo todos los fuentes cuando puede que solo alguno de ellos haya sido modificado. Una forma más eficiente de realizar la compilación de código distribuido en múltiples ficheros fuente es mediante la herramienta `make`. Debes copiar el fichero `makefile` proporcionado en Moodle dentro del directorio donde estén los ficheros fuente y hacer la siguiente llamada:

Terminal

```
$ make
```

Puedes consultar las transparencias 60 en adelante del *Tema 5* si necesitas más información sobre su funcionamiento.

Algunos de los métodos que vas a crear en esta práctica deben lanzar excepciones. Para ello deberás utilizar `throw` seguido del tipo de excepción, que será uno de los valores posibles del tipo enumerado `Error`. Esta excepción se deberá capturar mediante `try/catch` donde quiera que se invoque a un método que pueda lanzar una excepción. A continuación se muestra un ejemplo de cómo se lanzaría una excepción desde un método y cómo se capturaría en otro:

```
// Método donde se produce la excepción
Example::Example(string text)
{
  ...
  // Si "text" es una cadena vacía lanzamos la excepción con "throw"
  if(...){
    throw ERR_EMPTY;
  }
  ...
}
...
// Método donde se captura la excepción
void Chatbot::addExample(string name)
{
```

```
  ...
  try{
    ...
    Example example(text); // Tratamos de crear un nuevo ejemplo
    ...
  }
  // Si se produce la excepción, lanzamos el error correspondiente
  catch(Error e){
    Util::error(e);
  }
}
```

Para facilitar la posterior corrección por parte del profesorado de la práctica mediante pruebas unitarias (pruebas que evalúan el funcionamiento de cada una de las clases de manera aislada), deberás declarar los atributos y métodos privados de las clases como `protected` en lugar de como `private`. Un atributo o método `protected` es similar a uno privado. La diferencia es que los atributos o miembros `protected` son inaccesibles fuera de la clase (como los privados), pero pueden ser accedidos por una subclase (clase derivada) que herede de ella. Por ejemplo, la clase `Chatbot` se declararía de esta manera:

```
class Chatbot{
  // Ponemos "protected" en lugar de "private"
  protected:
    float threshold;
    ...
  public:
    Chatbot();
    ...
};
```

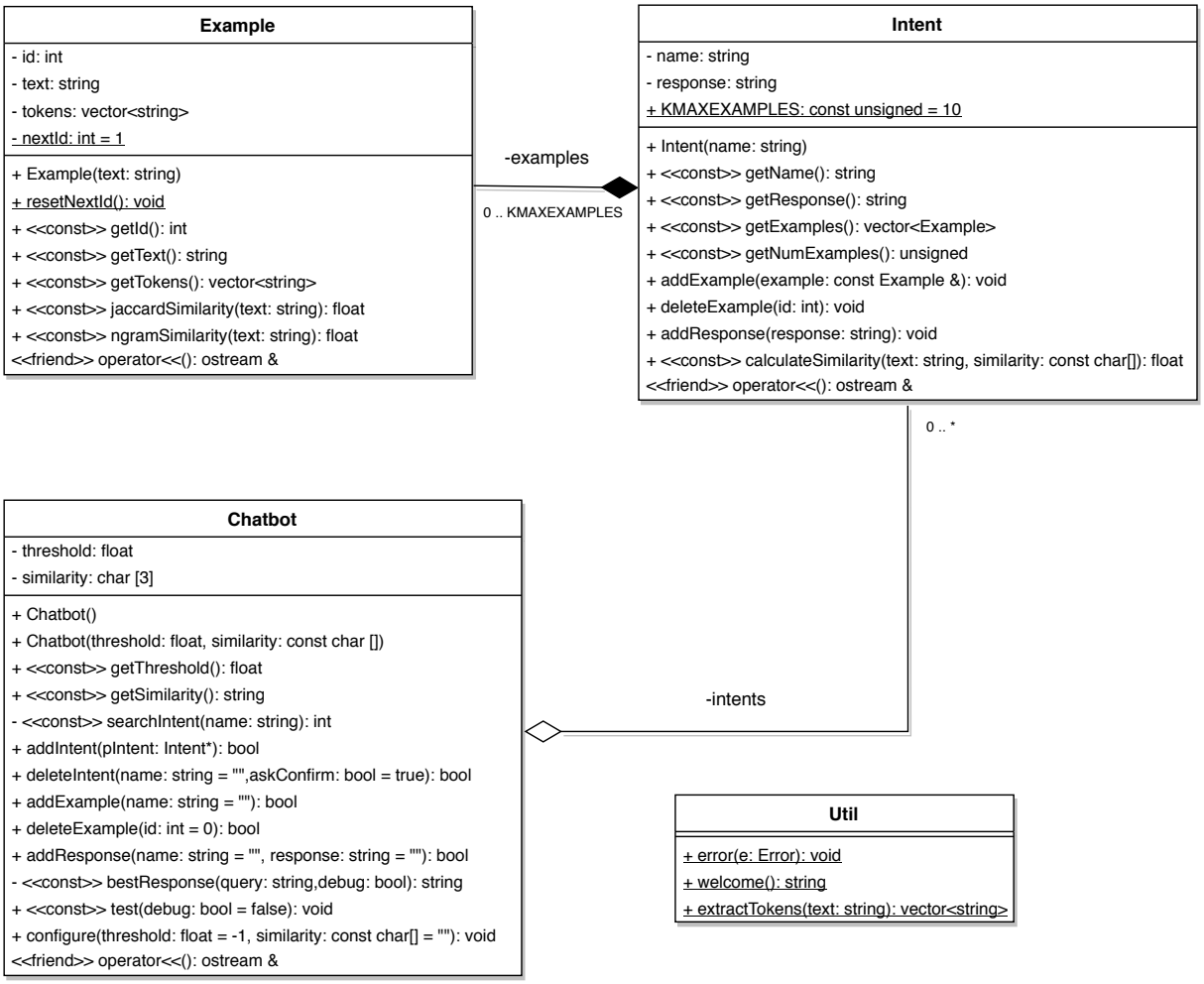
A lo largo de este enunciado, siempre que hablemos de métodos o atributos “privados” nos estaremos refiriendo a aquellos que irán en la parte `protected` de las clases que vas a definir.

3. Clases y métodos

La figura que aparece en la página siguiente muestra un diagrama UML con las clases que hay que implementar, junto con los atributos, métodos y relaciones que tienen lugar en el escenario descrito.

Si necesitas incorporar más métodos y atributos a las clases descritas en el diagrama, puedes hacerlo, pero siempre incluyéndolos en la parte privada (`protected`) de las clases. Recuerda también que las relaciones de *agregación* y *composición* dan lugar a nuevos atributos cuando se traducen del diagrama UML a código. Consulta las transparencias 57 y 58 del *Tema 5* si tienes dudas sobre cómo traducir las relaciones de *agregación* y *composición* a código.

- **¡Ojo!** En esta práctica no está permitido añadir ningún atributo o método público (`public`) a las clases definidas en el diagrama. Sin embargo, como se ha indicado, sí que puedes añadir atributos y métodos privados (`protected`)
- El atributo `nextId` de la clase `Example` almacena el identificador del siguiente ejemplo que se vaya a crear. Es decir, hace el papel del atributo `nextId` de la estructura `Chatbot` que se usaba en la Práctica 1 y 2 para almacenar este siguiente identificador
- El atributo `KMAXEXAMPLES` de `Intent` representan el número máximo de ejemplos que puede tener un Intent
- El resto de atributos de las clases almacenan la misma información que los registros correspondientes de las prácticas anteriores. Por ejemplo, el atributo `float threshold` de la clase `Chatbot` almacena la misma información que el campo `float threshold` que tenía el registro `Chatbot` de la Práctica 1 y 2



A continuación se describen los métodos de cada clase. Algunos de estos métodos no es necesario utilizarlos en la práctica, pero se utilizarán en las pruebas unitarias durante la corrección. Se recomienda implementar las clases en el orden en que aparecen en este enunciado.

3.1. Util

En el Moodle de la asignatura se proporcionará una versión inicial de esta clase, incluyendo el tipo enumerado `Error` con todos los posibles errores que se pueden dar en la práctica, además del método `error` para emitir los correspondientes mensajes de error por pantalla.

Por ejemplo, para mostrar el mensaje de error `ERR_INTENT`, deberás invocar al método `error` pasándole el correspondiente parámetro de la siguiente manera:

```
Util::error(ERR_INTENT);
```

Ten en cuenta que `error` es un método de clase (`static`) y por eso debe invocarse de esta manera. Consulta la transparencia 44 del *Tema 5* si tienes dudas a este respecto.

Además, deberás añadir los siguientes métodos a la clase:

- `string welcome()`. El fichero `Util.cc` incluye el vector `greetings` de las prácticas anteriores con todos los saludos iniciales del *chatbot*. Este método `static` devolverá un `string` conteniendo uno de los saludos iniciales de manera aleatoria, de la misma manera que se hacía en la práctica anterior. No se debe fijar la semilla inicial con `srand`. Esto se hace ya en el fichero `prac3.cc`
- `vector<string>extractTokens(string text)`. Dada una cadena de texto pasada como parámetro, este método `static` devolverá un vector con los tokens que la componen ya normalizados. Es

decir, de la cadena pasada como parámetro deberá eliminar acentos, signos de puntuación, pasarla a minúsculas, etc.

3.2. Example

Esta clase permite almacenar los datos de los ejemplos. Los métodos de esta clase son los siguientes:

- `Example(string text)`. Constructor para crear un ejemplo a partir de un texto pasado como parámetro. Este método deberá almacenar el texto pasado en el atributo `text` de la clase, guardar los tokens correspondientes en el atributo `tokens`, asignar al atributo `id` el valor que haya en ese momento en `nextId` (que en esta práctica ya no está en `Chatbot` sino que es un atributo de `Example`) y, finalmente, incrementar el valor de `nextId`. Si la cadena pasada como parámetro fuera vacía, o al limpiar los tokens todos quedaran vacíos, se deberá lanzar una excepción de tipo `ERR_EMPTY` (este tipo de error no se controlaba en las prácticas anteriores). En ese caso, al no crearse el ejemplo, no se debe incrementar el valor de `nextId`
- `void resetNextId()`. Este método permite reiniciar el atributo `nextId`, que almacena el identificador del siguiente ejemplo, volviéndolo a poner a `1`
- `int getId() const`. Devuelve el identificador del ejemplo
- `string getText() const`. Devuelve el valor del campo `text` del ejemplo
- `vector<string>getTokens() const`. Devuelve el vector `tokens` del ejemplo
- `float jaccardSimilarity(string text) const`. Devuelve la similitud de Jaccard entre el texto pasado como parámetro y el texto almacenado en el ejemplo. El cálculo de la similitud se realiza de la misma manera que en la Práctica 1
- `float ngramSimilarity(string text) const`. Devuelve la similitud, calculada mediante la distancia basada en n-gramas, entre el texto pasado como parámetro y el texto almacenado en el ejemplo. El cálculo de la similitud se realiza de la misma manera que en la Práctica 2
- `ostream& operator<<(ostream &os,const Example &example)`. Operador de salida que muestra por pantalla los datos del ejemplo con el mismo formato que la función `report` de la Práctica 1. Por ejemplo:

Terminal

```
Example 1: ¿Cuándo empiezan las vacaciones de Navidad?
Tokens 1: <cuando> <empiezan> <la> <vacacione> <de> <navidad>
```

Recuerda que el número que aparece después de `Example` y `Tokens` se corresponde al valor `id` del ejemplo. Detrás del último token mostrado (`<navidad>` en el caso anterior) no tiene que aparecer un salto de línea

3.3. Intent

Esta clase permite almacenar la información de las intenciones. Como se puede ver en el diagrama, la clase `Intent` tiene una relación de composición con la clase `Example`, lo que implica que en la clase `Intent` aparecerá un nuevo atributo de tipo `vector` para poder almacenar los ejemplos de la intención. Para más información sobre cómo se transforman a código las relaciones de composición entre clases, puedes consultar la transparencia 57 del *Tema 5*. Los métodos de esta clase son los siguientes:

- `Intent(string name)`. Constructor para crear una intención. Asignará la cadena pasada como parámetro al atributo `name` del `Intent`. Si el parámetro contiene una cadena vacía, el constructor lanzará la excepción `ERR_EMPTY`. El constructor no debe comprobar si hay otros `Intent` con el mismo nombre. Eso se hará en la función `addIntent` de `Chatbot`, como se explica en la Sección 3.4

- `void addExample(const Example &example)`. Añade un nuevo ejemplo al vector `examples` del `Intent`. El número máximo de ejemplos que puede tener una intención viene definido en el atributo `KMAXEXAMPLES` de la clase. Si el `Intent` ya contiene este número máximo en el momento de invocar al método, no se podrá insertar el nuevo ejemplo y se deberá lanzar la excepción `ERR_MAXEXAMPLES`. Esta condición no se controlaba en las prácticas anteriores
- `void deleteExample(int id)`. Elimina un ejemplo del vector `examples` del `Intent` cuyo identificador coincida con el valor pasado como parámetro a este método. Si no existe un ejemplo con ese identificador en el `Intent`, se lanzará la excepción `ERR_EXAMPLE`
- `void addResponse(string response)`. Añade la cadena pasada como parámetro al campo `response` del `Intent`
- `string getName() const`. Devuelve el nombre del `Intent`
- `string getResponse() const`. Devuelve la respuesta almacenada en el `Intent`
- `vector<Example>getExamples() const`. Devuelve el vector de ejemplos del `Intent`
- `unsigned getNumExamples() const`. Devuelve el número de ejemplos almacenados en el `Intent`
- `float calculateSimilarity(string text,const char similarity[]) const`. Dado un texto (primer parámetro) y un algoritmo de similitud (segundo parámetro), calcula la similitud del texto con todos los ejemplos que contenga el `Intent` y devuelve el máximo valor. Si no hubiera ningún ejemplo devolvería `0`. Los valores posibles del parámetro `similarity` son `jc` y `ng`. Si se proporciona un valor diferente a estos se lanzará la excepción `ERR_SIMILARITY`
- `ostream& operator<<(ostream &os,const Intent &intent)`. Operador de salida que muestra por pantalla los datos de la intención con el mismo formato que la función `report` de la Práctica 1. Si la intención tiene ejemplos almacenados, deberá utilizar el operador de salida de la clase `Example` para mostrarlos, tal y como se describía en la sección anterior. Éste sería un ejemplo de salida para un `Intent` que tiene un único ejemplo asociado:

Terminal

```
Intent: vacaciones navidad
Response: El 24 de diciembre
Example 1: ¿Cuándo empiezan las vacaciones de Navidad?
Tokens 1: <cuando> <empiezan> <la> <vacacione> <de> <navidad>
```

  Al igual que para `Example`, detrás del último token mostrado (`<navidad>` en el ejemplo) no tiene que aparecer un salto de línea

3.4. Chatbot

En esta clase se implementan los métodos relacionados con el *chatbot*. Como se puede observar en el diagrama, existe una relación de agregación entre `Chatbot` e `Intent`. Esto hará que aparezca otro atributo de tipo `vector` en `Chatbot` que almacenará punteros a `Intent`. Para más información sobre cómo se transforman a código las relaciones de agregación entre clases, puedes consultar la transparencia 58 del *Tema 5*. Los métodos de esta clase son los siguientes:

- `Chatbot()`. Crea un *chatbot* con los parámetros por defecto: umbral `0.25` y algoritmo `jc`
- `Chatbot(float threshold,const char similarity[])`. Crea un *chatbot* a partir de los parámetros pasados, que indican el umbral y el algoritmo de similitud. En primer lugar se comprobará si el umbral pasado es correcto (debe estar entre los valores `0` y `1`, inclusive). Si no lo es, se lanzará la excepción `ERR_THRESHOLD`. A continuación se comprobará si el algoritmo es válido (`jc` o `ng`). En caso contrario se lanzará la excepción `ERR_SIMILARITY`

- `float getThreshold() const`. Devuelve el umbral de similitud almacenado en el atributo `threshold`
- `string getSimilarity() const`. Devuelve el algoritmo de similitud almacenado en el campo `similarity`, pero en lugar de un tipo `char[]` devuelve un `string`
- `int searchIntent(string name) const`. Devuelve la posición, dentro del vector de punteros a intenciones, de aquel puntero que apunta a una intención cuyo nombre coincide con el que se ha pasado como parámetro. Si no lo encuentra devuelve `-1`
- `bool addIntent(Intent* pIntent)`. A este método se le pasa un puntero que apunta a un `Intent` ya creado y debe incorporar ese puntero al vector de punteros `intents` que tiene el *chatbot*. Antes de incorporarlo, deberá comprobar que no existe ya un puntero a una intención que tenga el mismo nombre. En caso de existir ya, se mostraría por pantalla el error `ERR_INTENT`. El método devuelve `true` si se ha podido añadir la intención y `false` en caso contrario
- `bool deleteIntent(string name="",bool askConfirm=true)`. Este método elimina del *chatbot* el puntero al `Intent` cuyo nombre coincida con el nombre pasado como parámetro. Si no existe una intención con ese nombre, deberá mostrar el error `ERR_INTENT`. Si la intención se ha podido eliminar devolverá `true` y si no devolverá `false`. Este método debe actuar igual que la opción `Delete intent` de la Práctica 1 y, dependiendo del valor de los parámetros de entrada, pedir el nombre del `Intent` y pedir confirmación al usuario antes del borrado con los mismos mensajes que se usaron en su momento. A continuación se muestran las posibles llamadas al método que se podrían hacer:

  ```
  deleteIntent(); // Pedirá el nombre del Intent y confirmación para borrarlo
  deleteIntent("cordial"); // Pedirá solo confirmación para borrar "cordial"
  deleteIntent("cordial",false); // No pedirá nada y borrará el Intent "cordial"
  ```

  Si se pasara como parámetro una cadena vacía en `name`, se mostrará también el mensaje solicitando el nombre del `Intent`

  ¡Ojo! El `Intent` no se destruye en realidad, ya que se trata de una relación de agregación. Lo que estamos haciendo es eliminar del *chatbot* el puntero que apunta a él para desvincularlo

- `bool addExample(string name="")`. Añade ejemplos al `Intent` cuyo nombre coincida con la cadena pasada como parámetro. Este método se comportará como la opción `Add example` de la Práctica 1. Si no se le pasa el parámetro al método o el valor `name` es igual a la cadena vacía, se le pedirá al usuario que introduzca el nombre del `Intent` con el mensaje habitual. Si no existiera, se mostraría el error `ERR_INTENT`. A continuación se pedirá que vaya introduciendo ejemplos hasta que introduzca `q`, en cuyo caso terminará. Si al crear o añadir un nuevo ejemplo se lanzara una excepción, deberá capturarse y mostrar el correspondiente error, para luego mostrar de nuevo el mensaje de solicitud de nuevo ejemplo. El método devolverá `true` si se inserta al menos un nuevo ejemplo y `false` en caso contrario.
- `bool deleteExample(int id=0)`. Elimina un ejemplo a partir de su identificador. Si no se le pasa parámetro, o el parámetro tiene el valor `0`, se mostrará el mismo mensaje que se mostraba en la opción `Delete example` de la Práctica 1 para pedir el identificador al usuario. Si no existe un ejemplo con ese `id` en todos los `Intent` que tiene el *chatbot*, se mostrará el error `ERR_EXAMPLE`. El método devolverá `true` si ha podido borrar el ejemplo y `false` en caso contrario
- `bool addResponse(string name="",string response="")`. Este método añade una respuesta a una intención. Puede recibir dos parámetros, el nombre de la intención y la respuesta que se quiere almacenar. Si no se pasa el primer parámetro, o su valor es la cadena vacía, se mostrará un mensaje solicitando al usuario el nombre del `Intent` igual que se hacía en la opción `Add response` de la Práctica 1. Del mismo modo, si no se pasa la respuesta, o su valor es la cadena vacía, se pedirá al usuario que la introduzca por teclado. Si no existe un `Intent` con ese nombre en el *chatbot* se mostrará el error `ERR_INTENT`. El método devuelve `true` si se ha podido guardar la respuesta y `false` en caso contrario

- `string bestResponse(string query,bool debug) const`. Este método devuelve una cadena con la mejor respuesta a una consulta (`query`) pasada como parámetro. Para ello, calculará la similitud de la consulta con todos los ejemplos almacenados en el *chatbot* y aplicará el algoritmo de similitud que éste tenga configurado en el atributo `similarity` (Jaccard o N-gramas). Si el máximo valor de similitud está por debajo del umbral (`threshold`) el método lanzará una excepción `ERR_RESPONSE`. El método cuenta con un segundo parámetro `debug`. Cuando este valor esté a `true`, se deberá incluir antes de la respuesta devuelta, y entre paréntesis, el valor máximo de similitud obtenido. Un ejemplo de cadena devuelta con la opción a `true` sería `"(0.666667) El 24 de diciembre"`. Con la opción a `false` la cadena devuelta sería simplemente `"El 24 de diciembre"`

- `void test(bool debug=false) const`. Este método replicará el funcionamiento de la opción `Test` de la Práctica 1, mostrando un saludo inicial y quedando a la espera de que el usuario escriba su consulta. Este método tiene que hacer uso del método `bestResponse` descrito anteriormente para obtener la mejor respuesta para una consulta del usuario, pasándole para ello el parámetro `debug` recibido. Si `bestResponse` lanzara una excepción por no superar el umbral, deberá de capturarse y mostrar por pantalla el error `ERR_RESPONSE`

- `void configure(float threshold=-1,const char similarity[]="")`. Este método realizará la misma funcionalidad que la opción `Configure` de la Práctica 2. Si no se proporciona el parámetro `threshold` o su valor es `-1`, se mostrará un mensaje al usuario solicitando que introduzca el valor de umbral, usando el mismo mensaje que en la Práctica 2. Si no se proporciona el parámetro `similarity` o éste contiene la cadena vacía, se mostrará un mensaje al usuario solicitando que introduzca el nombre del algoritmo usando el mismo mensaje que en la Práctica 2. Si el umbral introducido no está comprendido entre `0` y `1` se mostrará el error `ERR_THRESHOLD`. Si el algoritmo no es `jc` o `ng` se mostrará el error `ERR_SIMILARITY`

- `ostream& operator<<(ostream &os,const Chatbot &chatbot)`. Operador de salida que muestra por pantalla los datos del *chatbot* con el mismo formato que la opción `Report` de la Práctica 1. Si el *chatbot* tiene intenciones asignadas, se deberá utilizar el operador de salida de `Intent` para mostrar su información por pantalla. Al igual que en `Exaple` e `Intent`, después de la última línea mostrada (en el ejemplo de la Práctica 1 era `Examples per intent: 2.5`) no debe aparecer un salto de línea

¡Ojo! Algunos métodos de `Chatbot` lanzan excepciones, mientras que otros muestran mensajes de error. En la descripción de los métodos se especifica claramente en qué casos se lanzan excepciones y en qué casos se muestran errores. Concretamente, los constructores de la clase y el método `bestResponse` lanzan excepciones con `throw`, mientras que el resto de métodos muestran errores por pantalla invocando al método `error` de la clase `Util`

4. Programa principal

El programa principal estará en el fichero `prac3.cc` que se publicará en el Moodle de la asignatura. Este fichero contendrá la función que muestra los menús de opciones y la función `main` que se encarga de gestionar dichos menús. Este fichero utiliza los principales métodos que ofrecen las clases implementadas, aunque no todos. Te puede servir para entender mejor cómo se utilizan los módulos que has creado para cada una de las clases. Este fichero no debe entregarse con la práctica.