Universidad de Jaén
Departamento de Informática

Programación Orientada a Objetos

http://bit.ly/poouja

Curso 2019/2020

José Ramón Balsas Almagro
Víctor Manuel Rivas Santos
Ángel Luis García Fernández
Juan Ruiz de Miras

Práctica 9. Polimorfismo (II)

Objetivos

- Conocer qué son las **clases abstractas**, cómo se implementan y usan.
- Conocer el concepto de **Interfaz** y saber utilizarlo en C++.
- Saber aprovechar el concepto de polimorfismo a la hora de capturar excepciones que formen parte de la **jerarquía de excepciones estándar de C++**.

Índice

Contexto de la práctica

Tu nuevo programa para gestionar los poderes de los equipos mutantes parece que ha sido bien recibido por la patrulla X y les ha permitido tomar la iniciativa en sus acciones frente a los villanos de *Magneto*. Esto ha hecho que te ganes la confianza de *Charles*, el Profesor X, y te haya permitido acceder a los recursos más secretos de la organización. En concreto, has podido tener acceso a la máquina del tiempo y has aprovechado para viajar 10 años en el futuro para ver tu situación… y no podía ser peor. Terminaste la carrera (sí, también aprobastes POO con muy buena nota gracias a tu experiencia adquirida con las prácticas de la asignatura) y entraste a formar parte del equipo de desarrollo de S.H.I.E.L.D. alcanzando el puesto de jefe máximo de desarrollo (bueno, quizás haya tenido algo que ver que el resto del personal en plantilla fue despedido en sucesivos EREs y tú eres el único que queda con tu beca en prácticas de empresa no remuneradas...). También has descubierto que Segis finalmente no triunfó en el mundo de la música (como no podía ser de otra forma) pero se hizo

millonario revendiendo el código fuente de tu aplicación a una gran multinacional que además lo premió con un puesto de asesor musical en la delegación de Chafarinas donde pudiera explotar adecuadamente su potencial (sin molestar mucho, eso sí). Pero el caso es que no consideras nada justo cómo te va a tratar el futuro y decides darle un giro a la situación como hicieron tus antiguos profesores de POO que, dos años después de finalizar la asignatura, y utilizando su capacidad polimórfica (¿serían en realidad mutantes?), se convirtieron en guionistas de éxito de series de ciencia ficción en Bollywood. Tu plan de venganza es muy simple: aprovechando lo útiles que fueron los conocimientos que adquiriste en POO, decides utilizar la máquina del tiempo para volver al pasado y publicar una versión del célebre juego Minecraft, pero 10 años antes de su aparición real. De esta forma te convertirás en multimillonario, con tus influencias conseguirás que Segis sea el primer hombre en viajar a Marte en viaje sin retorno y tomarás el control de los mutantes, que estarán a tus órdenes en una gran consultora que controlará todo el desarrollo software a nivel mundial, pasando totalmente desapercibidos trabajando como personas norm… como informáticos. Y quién sabe, igual podrías comprar los derechos de Star Wars y crear una nueva línea argumental con la saga de la familia Gramo.

Realmente, la idea de desarrollar Minecraft en el pasado no fue totalmente tuya, sino que te la contó el último empleado del equipo de desarrollo de S.H.I.E.L.D. Él te proporcionó la versión preliminar en la que estaba trabajando pero nunca pudo terminarla porque no llegó a leerse los últimos capítulos del libro "Aprenda a programar en 7 días". Por lo tanto, tu primer paso consiste en mejorar el diseño de tu predecesor para poder gestionar correctamente los objetos del juego. Para empezar, sabes que en el juego, los objetos que utiliza el personaje pueden almacenarse en Cofres y decides comenzar por ahí.

Un Cofre (Figura 1) permite guardar hasta 27 Ítems de distintos tipos, que pueden ser utilizados por un jugador para diferentes fines: construir edificaciones, luchar, alimentarse, fabricar otros objetos, herramientas para facilitar ciertas tareas (talar árboles, picar piedra, pescar...), etc.

Figura 1: Cofre y su contenido

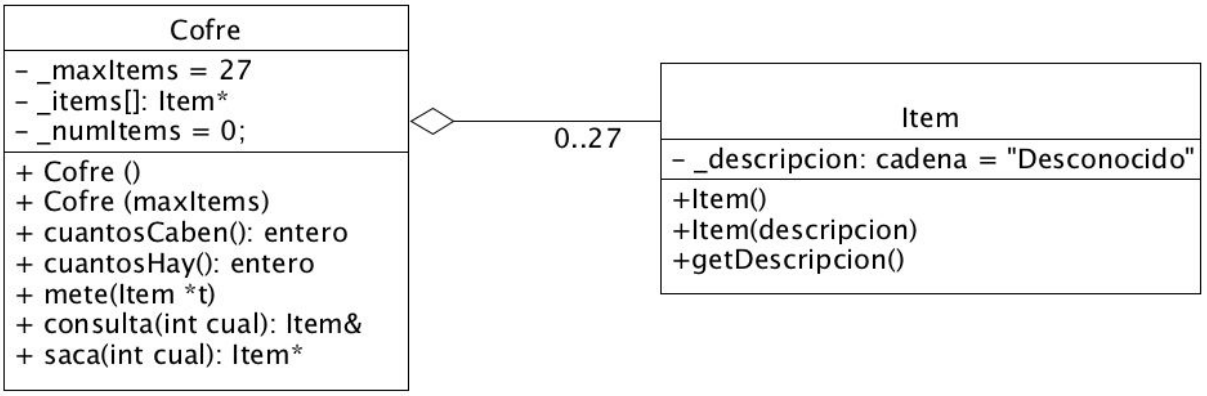

Figura 2: Diseño de clases original

Sin embargo, observando el diseño original (Figura 2) te das cuenta de que no se ha considerado que la descripción de los Ítems no es fija y puede variar dependiendo del tipo de Ítem y de sus características específicas (p.e. número de unidades que representa, tipo de material, estado en que se encuentra, etc.). Además, algunos de los Ítems (no todos) pueden apilarse para facilitar su manipulación por el jugador, por lo que habría que contemplar el número de unidades para estos Ítems.

En esta primera sesión haremos los cambios necesarios al diseño original para permitir que en un Cofre se puedan insertar hasta 27 ítems de diferentes tipos que se irán diseñando posteriormente. Por ahora solo serán necesarios bloques de construcción, espadas y filetes. Como se observa en la Figura 1, algunos de los objetos son apilables, como por ejemplo los bloques, por lo que para ellos será necesario conocer el número de unidades que representan (valor entre 1 y 64). Para cualquier ítem existente o que se añada en el futuro debe conocerse su descripción. En el caso de ítems que sean apilables, su descripción contendrá además el número de unidades que representan.

El nuevo diseño que se propone (Figura 3) permitirá en primer lugar, gracias a la interfaz *Ítem,* que todos los nuevos tipos de ítem que se creen en el futuro se puedan añadir a un Cofre y que también se pueda conocer su descripción. En segundo lugar, la clase Abstracta *ItemApilable* permite a los nuevos ítems que sean apilables, como los bloques de construcción, disponer de la funcionalidad necesaria para gestionar su número de unidades.

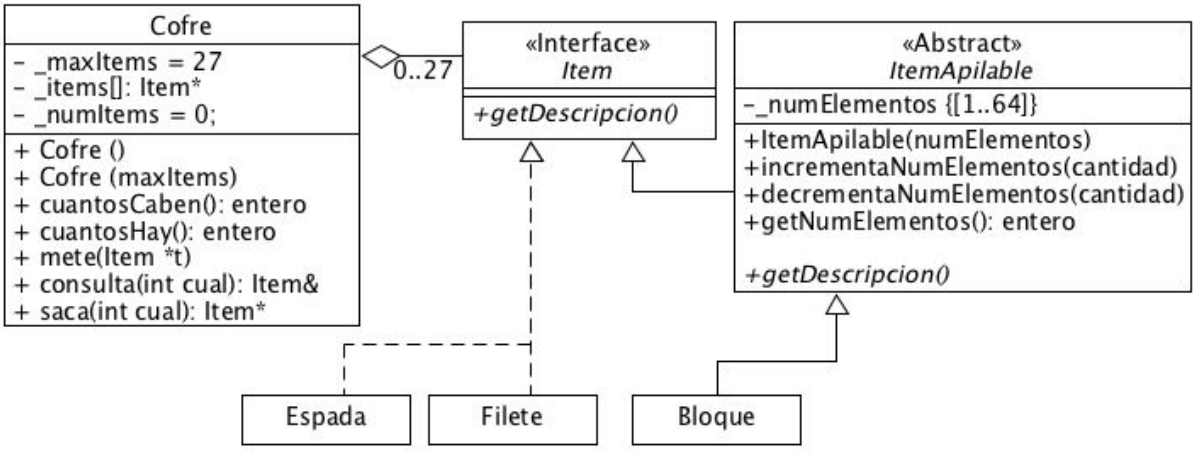

Figura 3: Diseño de clases propuesto

Punto de partida

El punto de partida de esta práctica es el programa de prueba del último empleado fijo del equipo de programadores de S.H.I.E.L.D antes de ser despedido. El proyecto NetBeans está disponible en el siguiente enlace: http://bit.ly/poouja1617pr9

Ejercicios

1. Transformar la clase Ítem en una **Interfaz** con un único método abstracto *getDescripción()* que deberán implementar sus derivadas.

2. Definir las clases **Espada**, **Filete** y **Bloque** que implementen la interfaz Ítem. Redefinir el método *getDescripcion()* de dichas clases para que devuelva respectivamente una cadena identificando al objeto en cuestión. p.e. "Una Espada", "Un Filete", "Un bloque de tierra". Adaptar programa principal para que inicialice el vector de punteros a Ítems utilizando objetos de las nuevas clases

3. Algunos tipos de Ítems permiten apilar varios objetos iguales en uno sólo para simplificar su gestión, conservando en este caso únicamente el número de unidades del objeto que representa dicho ítem. Defina una **clase abstracta *ItemApilable*,** que implemente (herede de) la interfaz *Ítem* y que disponga de la funcionalidad común para todos estos ítems: Construir un *ItemApilable* con un número determinado de unidades ([1,64], por defecto 1), consultar éste número, incrementarlo y decrementarlo en un número de unidades determinado.

4. Modificar la clase ***Bloque* para que herede de la clase *ItemApilable***. Redefinir el método *getDescripcion()* de la clase para que muestre además el número de unidades que representa cada Bloque, p.e. "Bloque de tierra (5)". Inicializar los objetos de este tipo de forma adecuada en el programa principal según el ejemplo propuesto: añadir al cofre respectivamente un Bloque de 5 unidades, otro de 8 y uno con una sola unidad.

5. Modifique las siguientes clases del diseño y el programa principal para que se puedan

detectar las siguientes situaciones anómalas y lancen las **excepciones adecuadas de la jerarquía estándar de C++** en su caso:

a. *ItemApilable*
    i. Lanzar *std::out_of_range* si se intenta añadir elementos fuera del rango admitido por la clase: constructor y métodos incrementa/decrementa
b. *Cofre*
    i. Lanzar una excepción *std::invalid_argument* si se intenta meter un puntero *nullptr* en el Cofre.
    ii. **Crear una nueva excepción *EmptyContainer*** que herede de *std::domain_error* y que represente intentos no válidos de operaciones sobre un cofre vacío
    iii. Lanzar excepciones de tipo *EmptyContainer* al tratar de consultar o sacar algún elemento de un contenedor que esté vacío
    iv. Puesto que Cofre sólo lanza excepciones de la jerarquía estándar de excepciones de C++, modificar el programa principal para capturar las excepciones que puedan lanzar sus objetos utilizando el tipo base std::exception: *catch (std::exception &e)*
c. Programa principal
    i. Capturar cualquier excepción que puedan generarse de la jerarquía estándar de C++ y que no haya sido capturada de forma expresa para informar al usuario del error que se ha producido. Utilizar polimorfismo de objetos de tipo excepción capturando una referencia a un objeto de tipo *std::exception*.

Contenidos

Clases abstractas

Una clase abstracta es aquella que incluye la declaración de uno o más métodos virtuales puros, esto es, métodos para los cuales se indica su cabecera completa pero no su implementación. Al no haber implementación para estos métodos, no se pueden declarar objetos del tipo de esta clase, sino simplemente punteros y referencias. Esto permitirá que posteriormente, en tiempo de ejecución y gracias a la ligadura dinámica se pueda llamar a dicho método que será implementado por cada una de las subclases.

Las clases abstractas sirven como superclases de las cuales derivaremos nuevas clases. Toda clase derivada de una clase abstracta hereda sus atributos y métodos. Si una clase derivada redefine todos y cada uno de los métodos virtuales puros de sus superclases, dejará de ser una clase abstracta; en caso contrario, si deja aunque sólo sea uno de los métodos virtuales puros por redefinir, seguirá siendo una clase abstracta de la cual no se podrán declarar objetos, sólo punteros y referencias.

Para incluir un método virtual puro en una clase indicaremos lo siguiente:

```
virtual tipo nombreMetodo( parámetros ) = 0;
```

Como se puede observar, lo declaramos **virtual** (para poder utilizar ligadura dinámica posteriormente) e incluimos una cabecera de método como cualquier otra: tipo devuelto, nombre del método y posibles parámetros que puede llevar (con su tipo, nombre, etc). La diferencia estriba en que usaremos al finalizar una asignación: **= 0**; indicando de esta forma que es virtual pura.

Interfaces

Otro uso de las clases abstractas puras en C++ es para la implementación de ***Interfaces***. Una *interfaz* sirve para definir un comportamiento esperado de un conjunto de objetos independientemente de las clases a las que estos pertenezcan. De esta forma, es posible implementar funciones que, haciendo uso del polimorfismo que aporta el uso de punteros a objetos de la Interfaz, permitan manipular objetos de clases diferentes como si fueran del mismo tipo; es decir, que a efectos de la funcionalidad que comparten por heredar de una misma interfaz, es posible tratarlos de la misma forma.

En algunos lenguajes, como es el caso de Java, C# o PHP, las interfaces utilizan una sintaxis específica. En C++ se utiliza la misma sintaxis de una clase abstracta pura donde, por definición de interfaz, no existen atributos y todos los métodos son virtuales puros. De esta forma, y haciendo uso de la herencia múltiple, objetos de clases con ancestros no relacionados, pueden tratarse de la misma forma si implementan (heredan en C++) adicionalmente una o varias Interfaces.

Un ejemplo habitual de uso de interfaces es la implementación de métodos o funciones que se encargan de almacenar o transferir objetos de diferentes clases a dispositivos. Por ejemplo, imaginemos que en una aplicación de compraventa de vehículos coexisten las dos jerarquías anteriores de objetos: *Vehículo*, *Furgoneta*, *Caravana*,etc. y *Usuario*, *Comprador*, etc... En dicha aplicación se ve oportuno implementar funciones de utilidad para almacenar o recuperar objetos de cada una de las clases anteriores en ficheros en formato CSV. En este caso, todas las clases deberían implementar métodos análogos a los ya vistos (*toCSV*/*fromCSV*) para pasar sus atributos a una cadena de caracteres en formato CSV o para recuperarlos a partir de ella posteriormente. Sin embargo, las funciones para almacenar vectores de punteros a objetos en un fichero deben ser diferentes puesto que los objetos son de clases distintas y no relacionadas. O al menos, haciendo polimorfismo a partir de punteros a las clases principales de las diferentes jerarquías (*Vehiculo* y *Usuario*), necesitaríamos, al menos, dos procedimientos de almacenamiento y dos de recuperación:

```
void Util::AlmacenaVehiculos ( Vehiculo* v[], int numVehiculos, string& nomAr );
int  Util::RecuperaVehiculos ( Vehiculo* v[], int tamV, string& nomArchivo );
void Util::AlmacenaUsuarios  ( Usuario*  v[], int numUsuarios, string& nomArchivo );
int  Util::RecuperaUsuarios  ( Usuario*  v[], int tamV, string& nomArchivo );
```

A cada uno de estos métodos podríamos pasar indistintamente (gracias al polimorfismo) vectores de punteros a objetos de la clase base que apuntarán a cualesquiera de los objetos de las clases derivadas.

Sin embargo, las implementaciones son idénticas salvo por el tipo de los parámetros. No obstante, si observamos lo que estas implementaciones necesitan saber de los objetos, nos

fijamos que es únicamente si tienen métodos *toCSV* o *fromCSV*. Es decir, la implementación de las cuatro funciones de almacenamiento/recuperación se podrían unificar en sólo dos que trabajasen únicamente con objetos que se "comportasen" de una forma similar. Por lo tanto, una opción factible, según se ha tratado anteriormente, es crear una interfaz, *itemCSV*, que refleje este comportamiento (métodos *toCSV* y *fromCSV*) y que las clases *Usuario* y *Vehículo* "implementen" (hereden en C++) dicha Interfaz:

```
/**@brief Interfaz objetos convertibles a/desde CSV*/
class ItemCSV
{  public:
       virtual std::string toCSV () = 0;
       virtual bool fromCSV ( string& cadenaCSV ) = 0;
};
```

```
class Vehiculo: public ItemCSV
{ public:
  // Esta clase o cualquier derivada deben tener implementados
  // los métodos de la interfaz
  virtual std::string toCSV ()
  {  // Código del método
  }
  virtual bool fromCSV ( string& cadenaCSV )
  {  // Código del método
  }
};
```

```
class Usuario: public ItemCSV
{ public:
  // Esta clase o cualquier derivada deben tener implementados
  // los métodos de la interfaz
  virtual std::string toCSV ()
  {  // Código del método
  }
  virtual bool fromCSV ( string& cadenaCSV )
  {  // Código del método
  }
};
```

De esta forma, los métodos necesarios para volcar a disco/recuperar cualquier tipo de objetos que se puedan transformar en formato CSV quedarían unificados en los dos siguientes:

```
void Util::AlmacenaObjetosCSV ( ItemCSV* v[], int numObjs, string& nomArchivo );
int  Util::RecuperaObjetosCSV ( ItemCSV* v[], int tamV, string& nomArchivo );
```

Incluso se podrían reutilizar en el futuro para otros objetos de nuevas clases que se integrarán en la aplicación siempre que implementen la interfaz *itemCSV*.

Uso de la jerarquía estándar de excepciones en C++

En prácticas anteriores hemos visto una ventaja importante que supone utilizar objetos de tipo excepción especializados frente a utilizar tipos simples (enteros) o incluso objetos std::string : podemos distinguir el tipo de excepción concreto a la hora de capturarlas, preguntando en el *catch* por cada tipo particular de excepción que deseemos atender.

Sin embargo, las principales ventajas se obtienen a la hora de crear jerarquías de clases de tipo excepción:

- Se puede reutilizar comportamiento (p.e. métodos de obtención de información del error a partir de la excepción, como *what()*).
- Se pueden definir especializaciones concretas de excepciones que identifican a problemas similares: argumento inválido (*invalid_argument*), intento de acceso a un valor fuera de un rango (*out_of_range*), etc. que pueden capturarse expresamente en apartados *catch()* para el tipo determinado.
- Se puede utilizar el polimorfismo de los objetos en las relaciones de herencia para poder capturar excepciones de diferentes tipos de una jerarquía de forma unificada utilizando referencias a objetos de clases base.

Debido a estas ventajas a la hora de gestionar las excepciones de un programa, existe ya en C++ una jerarquía de clases de tipo excepción de uso generalizado para estas tareas. Dicha clase deriva de la clase base ***std::exception***. Podemos estar prácticamente seguros de que cualquier función o método de una biblioteca que estemos utilizando (y que esté correctamente programada) lanzará excepciones de esta clase o de alguna clase derivada de la misma. Por tanto, si usamos la clase *std::exception* o sus derivadas permitiremos a otros programadores utilizar nuestras bibliotecas y a nosotros simplificar la captura de todas las excepciones que ocurran en nuestro programa.

La Figura 4 muestra la jerarquía de clases derivadas de la clase exception y que ya vienen definidas en el módulo <*stdexcept*>.

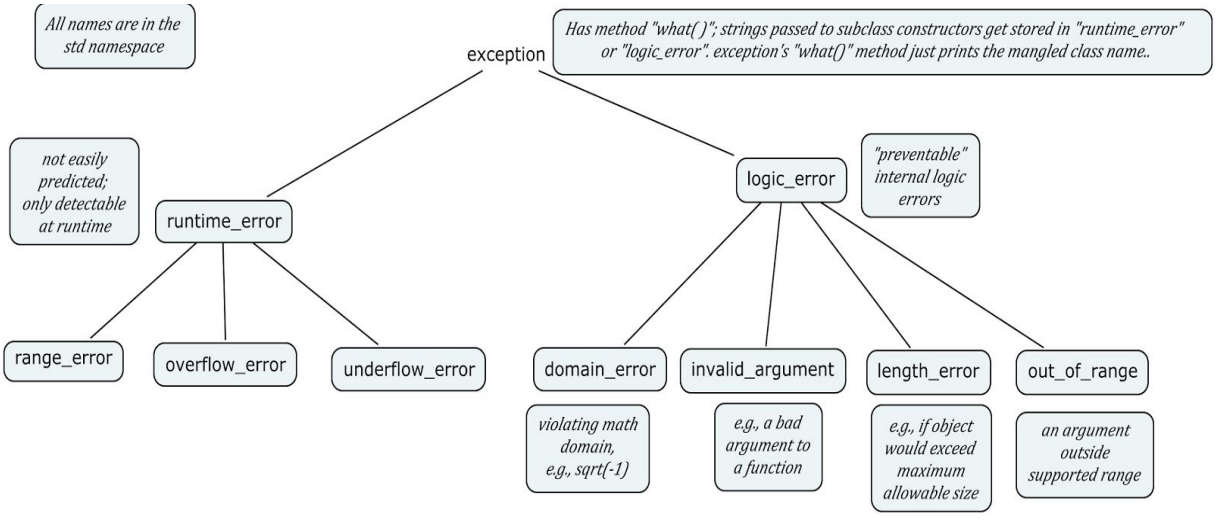

Figura 4. Jerarquía de clases de excepción en C++ (*Fuente: James R. Miller,*

https://people.eecs.ku.edu/~jrmiller)

Ya hemos visto en prácticas cómo capturar de forma independiente algunas de estas excepciones de la jerarquía estándar de C++. Veamos ahora algunos ejemplos a la hora de trabajar sobre esta jerarquía de excepciones.

En primer lugar, puede ser útil usar polimorfismo de objetos para capturar excepciones de diferentes clases de la jerarquía mediante una referencia a un objeto de la clase base. Supongamos un método que puede lanzar diferentes tipos de excepciones derivadas de la clase *std::logic_error*:

```
/**Añade un nuevo propietario a un vehículo
   @throw invalid_argument si el propietario es un puntero nulo, nullptr
   @throw out_of_range si se supera el número máximo de propietarios
   @throw domain_error si el propietario es menor de edad
   @throw bad_alloc si no hay memoria para añadir el nuevo elemento*/
void Vehiculo::nuevoPropietario ( Propietario& p )
{...}
```

Si deseamos capturar todas las posibles excepciones que puede lanzar el método podríamos hacerlo de forma genérica:

```
Vehiculo v ( "9999AAA" );
Propietario p ( "Juan" );

try
{  v.nuevoPropietario ( p );
}
catch ( std::exception& e )
{  //Uso de polimorfismo de objetos
   std::cerr << "No se puede añadir el propietario. Error: "
             << e.what() << std::endl;
}
```

o de forma selectiva:

```
try
{  v.nuevoPropietario ( p );
}
catch ( std::bad_alloc& e )
{  std::cerr  << "Error no hay memoria.  " << e.what() << std::endl;
   throw e; //Excepción no manejable. No puede continuar la ejecución
}
catch ( std::logic_error& e )
{  std::cerr << "No se puede añadir el propietario. Error: "
             << e.what() << std::endl;
    //Excepción manejable. Podemos continuar la ejecución
}
```

Sin embargo, en el ejemplo anterior, la excepción *std::domain_error* parece que no identifica

de forma específica al tipo de error que queremos notificar (el propietario no es mayor de edad), por lo que podríamos decidir crear un nuevo tipo de excepción derivado para detectar esta situación específica:

```
class MenorEdad: public std::domain_error
{  public:
      MenorEdad ( std::string error );
      MenorEdad ( const MenorEdad& orig );
      virtual ~MenorEdad() noexcept;
};
```

```
MenorEdad::MenorEdad ( std::string error ): std::domain_error ( error )
{ }
MenorEdad::MenorEdad ( const MenorEdad& orig ): std::domain_error ( orig )
{ }
MenorEdad::~MenorEdad () noexcept
{ }
```

y ahora capturarla de forma específica en la secuencia anterior:

```
try
{  v.nuevoPropietario ( p );
}
catch ( std::bad_alloc& e )
{  std::cerr << "Error no hay memoria.  "  << e.what() << std::endl;
   throw e; //Excepción no manejable. No puede continuar la ejecución
}
catch ( MenorEdad& e )
{  std::cout << "No se admiten propietarios menores de edad " << std::endl;
   //Excepción manejable. Podemos continuar la ejecución
}
catch ( std::logic_error& e )
{  std::cerr << "No se puede añadir el propietario. Error: "
             << e.what() << std::endl;
   //Excepción manejable. Podemos continuar la ejecución
}
```