

# Clase 6 - Excepciones - Tratamiento de errores y control de recursos

**Autor:** Pablo D. Roca <[pablodroca@gmail.com](mailto:pablodroca@gmail.com)>  
**Revisión:**2

<b>Clase 6 - Excepciones - Tratamiento de errores y control de recursos. 1</b>	
1. Introducción .....	1
2. Métodos de tratamiento de errores.....	1
Enfoque conservador.....	2
Enfoque optimista.....	3
3. Lanzamiento de excepciones .....	4
4. Captura de excepciones .....	4
5. Jerarquías de excepciones.....	7
Especificación del error.....	7
Captura especializada y generalizada.....	8
6. Control de recursos.....	9
RAII (Resource Acquisition is Initialization).....	10
Buen uso del Stack .....	10
Buen uso de la composición de objetos .....	11
Excepciones en constructores y destructores .....	13
7. Buenas prácticas .....	14
8. Referencias.....	14

## 1. Introducción

Uno de las características más buscadas en cualquier sistema informático es la robustez y la estabilidad del mismo frente a cualquier escenario de ejecución. Es imposible pensar en cumplimentar esa característica sin realizar un chequeo más o menos periódico del estado actual del sistema, que nos asegure que su estado aún es válido y que se puede continuar con el procesamiento.

Existen diversas técnicas para realizar el control por el estado del sistema. Algunas de ellas son más intrusivas que otras. Algunas demandan mayor complejidad y cuidado.

Intentaremos abordar el tratamiento de errores y el manejo de excepciones a fin de fijar un conjunto de herramientas que permitan asegurar cierto grado de robustez en un sistema de mediana escala.

## 2. Métodos de tratamiento de errores

Uno de los problemas más frecuentes en la ejecución de un sistema surge con las diversas situaciones de error producto de variables que no pueden ser controladas por él mismo. El entorno de ejecución en el cual se desarrolla un programa cambia constantemente

provocando una cantidad infinitamente grande de situaciones no contempladas por del desarrollador a la hora de resolver el problema que lo aboca. Frente a esta realidad es indispensable tomar precauciones para que cualquier error emergente sea tratado de la forma que le corresponda. A tal fin se suelen tomar al menos dos esquemas de tratamiento de errores:

## Enfoque conservador

Básicamente supone un escenario desfavorable frente a cada paso que realiza el sistema. Esto implica preguntar si una operatoria se puede realizar antes de intentar utilizarla o bien un chequeo posterior del estado del aplicativo para determinar si la operación anterior causó algún fallo que dejara al sistema en estado inválido.

De esta forma, no es extraño escribir funciones que retornen códigos de resultados que deben ser chequeadas. Asimismo, los invocadores, al detectar un código de resultado no esperado deberán informar de lo ocurrido retornando ellos mismos un código de error. El siguiente ejemplo ilustra este enfoque:

```
int operacion()
{
    if ( ... )
    {
        //operaciones en estado válido
        return 0;
    }
    else
        return -1; //informo error
}

int operacion2()
{
    if (operacion())
        return -1; //operacion finalizó con cód. de error
    else
    {
        //operación finalizó exitosamente
        ...
        return 0;
    }
}
```

Otro ejemplo similar donde se usa una variable global para informar sobre el estado:

```
void operacion()
{
    errno = 0;
    if ( ... )
    {
        //operaciones en estado válido
        ...
    }
    else
        errno = EJEC_OPERACION_INVALIDA;
```

```
    }  
  
void operacion2()  
{  
    operacion();  
    if (! errno)  
    {  
        //operación finalizó exitosamente. Continuamos con la ejecución  
        ...  
    }  
}
```

## Enfoque optimista

En este caso se opta por suponer que toda operación se ejecuta felizmente, en el estado ideal para el que fueron pensados los distintos pasos del algoritmo. En un enfoque optimista se evita chequear paso a paso por las distintas operaciones realizadas. Por el contrario, se espera que el sistema avise de alguna manera su condición de error para que pueda ser tratada.

Implementar este tipo de manejos no es tarea sencilla sin un nuevo concepto. El siguiente ejemplo es un intento de enfoque optimista:

```
void manejarError()  
{  
    perror("Ha ocurrido un error grave");  
    exit(EXIT_FAILURE);  
}  
  
void operacion()  
{  
    if (...)  
        manejarError(); //en caso de error, no se retorna el control del sistema  
    //aquí podemos continuar con nuestras tareas bajo enfoque optimista  
    ...  
}  
  
void operacion2()  
{  
    operacion();  
    ...  
    //seguimos ejecutando tareas bajo el enfoque optimista.  
}
```

El código del ejemplo anterior es mucho más claro que el código de enfoques conservadores. Los pasos del algoritmo no se ven afectados por el chequeo constante del estado del sistema. Pero fue necesario establecer un punto común de manejo de errores forzando un ABORT del sistema. Esto no es aceptable al programar bibliotecas de funciones y clases.

### 3. Lanzamiento de excepciones

Las excepciones facilitan el uso de enfoques optimistas de control de errores. Una excepción es un objeto que informa sobre una condición anómala, no frecuente o de error durante la ejecución de un programa.

La aparición de una excepción no es necesariamente un acontecimiento malo; el sistema puede estar aún en un estado válido pero cierto algoritmo detectó una condición que no fue planeada dentro de su 'camino optimista'. El caso excepcional comúnmente rompe con la armonía del algoritmo necesitando un manejo que éste no le puede dar. En estas circunstancias lo más prudente es informar a capas superiores sobre el problema para que éstas decidan que hacer.

Una excepción entonces nos otorga un mecanismo para lanzar hacia arriba el problema encontrado, desapilando la pila de llamadas hasta que alguno de las funciones invocadores sepa como tratar el caso excepcional.

En C++, una excepción es un objeto común y corriente, no requiere ningún tipo particular ni formalidad alguna en su construcción. Un **std::string** puede ser una excepción así como un **const char\***, un **int**, una **std::exception** o cualquier otra instancia. Las excepciones se lanzan mediante la palabra reservada **throw**. Los siguientes son ejemplos de disparo de excepciones:

```
throw "Un error ha ocurrido";
```

```
throw std::string("error");
```

```
throw COD_EJEC_INVALIDA;
```

```
throw new MiClase();
```

```
throw MiClase("error");
```

Como está por verse, este mecanismo posee un poder enorme a la hora de controlar el flujo de un sistema. Al lanzarse una excepción se corta el flujo actual y se eleva la misma hasta el punto donde alguien la captura y maneja el caso de error. Se supone que si alguien captura un tipo determinado de excepción es porque conoce el significado y sabe cómo manejar ese tipo de errores.

### 4. Captura de excepciones

Una vez lanzada, la excepción se elevará hasta encontrar un mecanismo de guarda que indique que se desea realizar el manejo del caso anómalo. Si bien es posible no controlar las excepciones, nuestro código TIENE que hacerlo en algún punto para evitar que la pila de llamadas se rebobine por completo y el sistema aborte de forma no controlada.

Las excepciones se capturan mediante la sentencia **catch**. Los **catch** hacen referencia al tipo de excepción que capturan y permiten recibir el objeto excepción tal y como si se tratara de un parámetro en una función. De esta forma, su declaración cumple con las reglas usuales para declarar las firmas de funciones.

Veamos los **catch** que deberían realizarse según los **throw** utilizados en el ejemplo anterior:

```
catch (const char* e) //para throw "Un error ha ocurrido";
```

```
catch (std::string e) //para throw std::string("error");  
catch (std::string& e) //otra forma para throw std::string("error");  
catch (const std::string& e) //otra forma para throw std::string("error");  
catch (int e) //para throw COD_EJEC_INVALIDA;  
catch (MiClase* e) //para throw new MiClase();  
catch (const MiClase& e) //para throw MiClase("error");
```

Pero con los catch no alcanza para definir desde donde se acepta capturar excepciones. La sintaxis completa del manejo de excepciones incluye la palabra reservada **try** para indicar que se intenta ejecutar un código y uno o varios catch para capturar distintos errores posibles dentro del código a intentar ejecutar. El siguiente código muestra un ejemplo de uso:

```
void operacion()  
{  
    if (...)  
        throw std::exception();  
    else if (...)  
        throw MiExcepcion("No se esperaba el estado actual");  
    ...  
}  
  
...  
try  
{  
    operacion();  
    ...  
}  
catch(const std::exception&)  
{  
    //no hacer nada, esta excepción podría ocurrir sin presentar necesariamente un error para el sistema  
}
```

Para el ejemplo anterior, el disparo de la excepción `std::exception` es un caso contemplado, su captura simplemente evita que el error se propague hacia arriba indicando que el catch sabe como tratar el problema. Notar que nadie está capturando la excepción `MiExcepcion` en el código anterior. Si esa excepción llegara a ocurrir el sistema posiblemente abortaría.

Es importante destacar que toda instrucción colocada dentro del bloque `try` que se encuentre por debajo del punto de lanzamiento de una excepción, **NO SERÁ EJECUTADA** si la excepción se dispara.

Supongamos el caso en que no se sabe si el origen de la excepción causa o no algún estado inválido para el sistema. En estas circunstancias puede ser necesario informar del error en el momento en que ocurrió pero dejar que este continúe elevándose. Para ello se puede reutilizar la excepción capturada e invocar un nuevo `throw`:

```
try
{
    operacion();
    ...
}
catch(const std::exception& e)
{
    //loguear y relanzar la excepción
    std::cerr << "Ha ocurrido un error." << e.what() << std::endl;
    throw;
}
```

Notar que no es necesario utilizar la variable de la excepción para realizar el re-throw. El sistema conoce la excepción que está siendo lanzada y la utiliza para el throw.

Si se pretendiera capturar MiExcepcion lo único necesario es colocar una cláusula catch para ella:

```
try
{
    operacion();
    ...
}
catch(const std::exception& e)
{
    //loguear y relanzar la excepción
    std::cerr << "Ha ocurrido un error." << e.what() << std::endl;
    throw MiExcepcion(e.what());
}
catch(const MiExcepcion& e)
{
    //loguear y relanzar la excepción
    std::cerr << "Ha ocurrido un error." << e.what() << std::endl;
    throw;
}
```

En el ejemplo anterior, los distintos catches aseguran que las excepciones del tipo `std::exception` sean 'convertidas' al formato `MiExcepcion`. Funciones invocantes sólo deberán tomar en cuenta `MiExcepcion` como tipo de excepción que este código puede lanzar, quedando más restringido el tipo de anomalías que pueden aparecer. Notar que el primer catch lanza un objeto `MiExcepcion` que no es capturado por el segundo catch. Esto es así porque los **catch** SÓLAMENTE actúan sobre el bloque **try**. Cualquier instrucción colocada en los bloques **catch** no será manejada por los sucesivos, al menos que se aniden bloques try/catch.

Pero en el caso anterior no nos aseguramos que nuestro código tire solamente excepciones del tipo `MiExcepcion`. Para que esto fuera así deberíamos asegurarnos que 'operacion()' no arroja ningún otro tipo de excepción o bien capturar todos los otros tipos posibles. En el siguiente ejemplo se capturan cualquier otro tipo de excepción utilizando el operador elipsis (de la misma forma que al utilizarlo en las firmas de las funciones).

```
try
{
    operacion();
    ...
}
catch(const std::exception& e)
{
    //loguear y relanzar la excepción
    std::cerr << "Ha ocurrido un error." << e.what() << std::endl;
    throw MiExcepcion(e.what());
}
catch(const MiExcepcion& e)
{
    //loguear y relanzar la excepción
    std::cerr << "Ha ocurrido un error." << e.what() << std::endl;
    throw;
}
catch(...)
{
    throw MiExcepcion("Error no reconocido.");
}
```

## 5. Jerarquías de excepciones

### Especificación del error

Hasta ahora nos hemos limitado a arrojar excepciones dispares con significados dispares, esto no suele ser de gran utilidad a la hora de realizar un sistema de mediana envergadura. Si se realiza un relevamiento de cualquier programa, seguramente se note que el grueso de los errores que se manejan y se informan al usuario puede ser agrupado o clasificado en unos pocos tipos. Estas clasificaciones seguramente respondan a los distintos conceptos debajo de cada tipo de anomalía que soporta el sistema. Como alternativa para agrupar cada especialización de un caso excepcional se acuerda realizar jerarquías de clases por tipo de anomalía.

De esta forma, se pueden definir varios tipos de errores bajo un mismo concepto utilizando la herencia para especializar:

```
class ComunicacionException: public std::exception
{
};
class ConexionCanceladaException: public ComunicacionException
{
};
class EnvioInvalidoException: public ComunicacionException
{
};
class RecepcionInvalidaException: public ComunicacionException
{
};
```

En este caso se realizó una jerarquía de los posibles errores de comunicación. Todo error específico hereda de `ComunicacionException`. Esta clase a su vez hereda de `std::exception` que es la excepción base provista por STL. Este paso es opcional pero recomendable por cuestiones que veremos a continuación.

## Captura especializada y generalizada

El principal beneficio de realizar una jerarquía de excepciones es la capacidad de capturar errores de manera general o específica dependiendo del caso. Suponiendo como existente la jerarquía antes indicada, puede resultar útil la posibilidad de capturar todo tipo de excepción de comunicación en un punto en lugar de hacerlo tipo por tipo. De esta forma se soporta automáticamente el agregado de nuevos tipos de excepciones de comunicación, el código de la captura no tiene necesidad de cambiar.

```
try
{
    operacion();
}
catch(const std::ComunicacionException& ex)
{
    //cualquier tipo de error en la comunicacion es manejado en este punto.
    //el objeto 'ex' puede ser de cualq. clase heredera de
    std::ComunicacionException
}
```

Notemos que en todos los ejemplos indicados hasta aquí se hace uso de la palabra reservada **const** y de operador **&** para indicar que la excepción capturada es una referencia constante de la original. En este momento, esta afirmación cobra especial importancia: 'ex' puede ser de cualquier tipo heredero de `std::ComunicacionException` por lo tanto, se deben tomar las mismas precauciones que en un pasaje de parámetros de objetos polimórficos. Esto es:

```
try
{
    operacion();
}
catch(std::ComunicacionException ex)
{
    //cualquier tipo de error en la comunicacion es manejado en este punto.
    //el objeto 'ex' puede ser de cualq. clase heredera de
    std::ComunicacionException
}
```

Actúa de la misma forma que el ejemplo anterior, pero en este caso la variable `ex` es indefectiblemente una copia del objeto originalmente lanzado. Esto implica que de tratarse de un clase heredada, el *slicing* elimina la posibilidad de polimorfismo.

Planteemos un caso más complejo donde se pretende realizar un envío de datos con una cantidad dada de reintentos en caso de encontrarnos con una excepción de envío inválido. reintentos :

```
bool enviado = false;
int intentosEnvio = 0;
while(!enviado)
{
    try
    {
```

```
        enviarDatos();
        enviado = true;
    }
    catch(const std::EnvioInvalidoException&)
    {
        //volver a intentar el envio
        intentosEnvio++;
        if (intentosEnvio == MAX_INTENTOS_ACEPTADOS)
            throw;
    }
    catch(const std::ComunicacionException&)
    {
        //cualquier otro tipo de error en la comunicacion debe abortar el proceso
        std::cerr << "Ha ocurrido un error durante la comunicación. No se pudo
        finalizar el envío" << std::endl;
        throw;
    }
}
```

El código del ejemplo anterior ejecuta la función `enviarDatos`, saliendo luego del bucle si la ejecución fue exitosa. Bajo una ejecución fallida de `enviarDatos`, en caso de ocurrir una excepción del tipo `EnvioInvalidoException` se incrementa el contador y se reintenta el envío. Si por otro lado, la excepción es del tipo `ComunicacionException` (o cualquier heredera salvo `EnvioInvalidoException`), se aborta la ejecución.

## 6. Control de recursos

La utilización de los distintos recursos que provee el sistema durante la realización de un programa conlleva una responsabilidad tácita que no todos tienen presente: el ciclo de vida del recurso y su administración suelen quedar en manos del programador.

A fin de cumplir con este compromiso es que se desarrollan técnicas, metodologías y reglas prácticas con el único fin de organizar la estructura del programa y con ello simplificar el control de los recursos necesarios. Recomendaciones tales como "adquirir y liberar los recursos de forma simétrica" o "no pedir un recurso a menos que sea necesario" son ejemplos de la intención de ordenar la programación inherente logrando que el control de recursos sea simple, previsible y fácil de comprender.

Aunque en muchos lenguajes y sistemas operativos se intente minimizar la necesidad de control de recursos automatizándola o haciendo transparente su uso; es importante entender que el alcance del término **recurso** abarca tanto a los componentes ofrecidos por el sistema operativo (como ser archivos, hilos, espacio de memoria, etc.) como a componentes externos a él tanto físicos como lógicos (dispositivos periféricos, dispositivos en máquinas remotas, componentes en un sistema remoto, etc.).

Frente a la problemática habitual del tratamiento de errores es que el control de recursos adquiere una importancia mayor: el esquema de tratamiento de errores debe ser compatible con las reglas de control de recursos para que éstas sigan siendo útiles. De los enfoques antes desarrollados, es el pesimista el que resulta más flexible para realizar control de recursos en todas sus variantes, sin embargo, este enfoque está en desuso para lenguajes con soporte a excepciones. Por otro lado, utilizar un enfoque optimista (con o sin excepciones) agrega una gran complejidad al control de recursos debido a que el flujo del programa no es secuencial y distintas subrutinas pueden abortar espontáneamente.

## RAII (Resource Acquisition is Initialization)

RAII (traducido frecuentemente como 'adquisición de un recurso es su inicialización') es una de las técnicas de programación para el control de recursos más conocidas. Fue inventada por Bjarne Stroustrup en respuesta a los conocidos problemas de liberación de memoria y archivos que son comunes en C++.

Tal y como lo indica su nombre, esta técnica plantea que el momento de adquisición de un recurso es la inicialización (en este caso de un objeto asociado) y, por simetría, su liberación será la destrucción (del objeto asociado). De esta forma se busca vincular el ciclo de vida de un objeto con el ciclo de uso de un recurso y suponer que si uno de estos es correctamente controlado, también lo será el otro.

Para ilustrar este concepto, veamos un ejemplo de encapsulación de la estructura FILE\* en C++ mediante RAII:

```
class Archivo
{
    private:
        FILE* archivo;
    public:
        Archivo(const char* ruta, const char* modo)
        {
            this->archivo = fopen(ruta, modo);
            if (! this->archivo)
                throw std::exception("El archivo no fue abierto
                    correctamente.");
        }

        ~Archivo()
        {
            fclose(this->archivo);
        }
        ...
        //operaciones de escritura y lectura
        ...
};
```

Se comprende fácilmente que construir un objeto de tipo Archivo supone adquirir el recurso de un archivo y destruirlo supone liberar ese recurso. Esta simetría cumple con todas las premisas que indicamos para un control de recursos: es simple, previsible y fácil de comprender.

### Buen uso del Stack

Ahora sólo debemos garantizar que los distintos objetos Archivo, siempre sean destruidos. Veamos un ejemplo de un mal uso de esta clase:

```
...
Archivo* archivo = new Archivo("archivo.txt", "r");
operarSobreArchivo(*archivo);
delete archivo;
...
```

Si bien se respeta la simetría en la construcción y destrucción del objeto, y este encapsula correctamente el concepto de RAII, el código anterior no contempla la ocurrencia de una excepción dentro de la función `operarSobreArchivo`. Una versión mejorada pero aún errónea sería:

```
...
Archivo* archivo = new Archivo("archivo.txt", "r");
try
{
    operarSobreArchivo(*archivo);
}
catch(...)
{
    delete archivo;
    throw;
}
delete archivo;
...
```

En este caso, el objeto `Archivo` es destruido ya sea si la función retornó exitosamente o si esta arrojó una excepción, sin importar su tipo. El uso correcto de la clase `archivo` sería:

```
...
Archivo archivo("archivo.txt", "r");
operarSobreArchivo(archivo);
...
```

Al utilizar una variable automática para el objeto de tipo `Archivo`, nos su destrucción en cuanto el programa salga del *scope* en que fue declarada. En este caso, una excepción arrojada por la función **`operarSobreArchivo`** es elevada hacia afuera del *scope* actual ya que no se especificó su captura. Al ocurrir esto, la variable **`archivo`** es destruida y eliminada del *stack*, forzando la invocación del destructor que libera los recursos adquiridos. Notemos que esta metodología funciona tanto para cuando contemplamos la posibilidad de una excepción como para cuando deseamos utilizar instrucciones de **`return`** en distintos puntos de código. Observemos además que al no utilizar el *heap* innecesariamente, ganamos en velocidad y no corremos peligro de perder memoria por un mal manejo del recurso.

Por último, debemos indicar que la cancelación del programa mediante una llamada a la función **`exit`** (o similar) no supone el rebobinado de la pila de ejecución (liberación de variables automáticas) con lo que no podemos confiar en la invocación del destructor para liberar los recursos adquiridos. Distinto es el caso de variables declaradas en contexto estático, que siempre son liberadas al finalizar el programa.

## Buen uso de la composición de objetos

El esquema de RAII y el uso de variables automáticas se completa con un correcto diseño del modelo de objetos y con un buen uso de la composición. Básicamente se busca establecer las relaciones de composición de forma correcta para que la destrucción de los recursos sea en cascada: la destrucción de un objeto compuesto presupone la destrucción de todos sus componentes. Veamos un ejemplo:

```
class CintaTransportadora
{
    ...
    ~CintaTransportadora()
    {
        if (this->enMovimiento())
            this->detener();
        this->cortarSuministroEnergia();
    }
    ...
};

class Robot
{
    ...
    virtual ~Robot()
    {
        this->cortarSuministroEnergia();
    }
    ...
};

class BrazoSoldador: public Robot
{
};

class BrazoMecanico: public Robot
{
    ...
    ~BrazoMecanico()
    {
        this->liberarPinzas();
    }
    ...
};
```

Las clases anteriores usan el patrón RAII para controlar recursos externos de suma importancia. Acciones como detener una cinta, liberar las pinzas de una máquina o cortar el suministro de energía eléctrica pueden ser vitales en un contexto de funcionamiento normal y también bajo la ocurrencia de un error.

Supongamos ahora una clase desde la cual se importen las órdenes para estos elementos:

```
...
ControlFabrica fabrica;
fabrica.inicializarRobots();
fabrica.encenderCinta();
fabrica.operar();
...
```

Una correcta implementación de la fabrica podría ser:

```
class ControlFabrica
{
private:
    CintaTransportadora cinta;
    std::vector<BrazoMecanico> pinzas;
    std::vector<BrazoSoldador> soldados;
    ...
};
```

donde la utilización de variables miembro, la correcta composición y el uso del *stack* nos evitan el uso de complejas estructuras para el control de recursos tan importantes. Otra posibilidad de implementación, podría ser:

```
class ControlFabrica
{
private:
    CintaTransportadora cinta;
    std::vector<Robot*> robots;
public:
    void inicializarRobots()
    {
        ...
        //realizar la contruccion de cada Robot en el Heap y luego colocarlos en el
        vector.
        ...
    }

    ~ControlFabrica()
    {
        std::vector<Robot*>::const_iterator itRobots;
        for(itRobots = robots.begin(); itRobots != robots.end(); ++itRobots)
        {
            delete *itRobots;
        }
    }
    ...
};
```

En este caso, al optar por el heap para almacenar los elementos compuestos, debemos realizar una programación extra para garantizar la premisa básica de la composición y liberar a los objetos contenidos.

## Excepciones en constructores y destructores

Practicamente todos los lenguajes de programación poseen reglas distintas para lanzamiento de excepciones durante el momento de la construcción o destrucción de objetos. Esto hace que decisiones de diseño sobre estos aspectos queden ligadas fuertemente al lenguaje en que se realice la implementación.

En particular, en C++ se establece que el lanzar una excepción durante la construcción de un objeto se aborta su creación con lo que no es necesario llamar al destructor asociado. Esto puede traer complicaciones si el objeto logró reservar recursos antes de lanzar la excepción. Se debe tomar la precaución de liberar estos recursos dentro del constructor y

antes de arrojar la excepción.

En el caso de excepciones dentro de destructores, en el caso en que éste haya sido invocado de forma implícita por la destrucción de una variable automática, no puede plantear un manejo elegante del error y el programa aborta de forma abrupta. Por este motivo se recomienda siempre evitar las excepciones en destructores.

## 7. Buenas prácticas

- Utilizar una cantidad controlada de excepciones por paquete o capa del sistema. Atrapar las restantes y arrojar sólo las que nuestra capa dice conocer.
- Utilizar jerarquías de excepciones para modelizar los distintos casos encontrados y así poder clasificarlos.
- Emplear siempre el modificador **const** y referencias en las capturas de excepciones. De esa forma se evita recortar las excepciones arrojadas por utilizar constructores de copia en los catch.
- Utilizar RAII siempre que se deba proteger un recurso y que se deba asegurar que éste no se perderá en caso de una situación anómala.
- Confiar en la composición de objetos y en la destrucción en cadena.
- Utilizar el stack para objetos de poco peso siempre que sea posible.

## 8. Referencias