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 10. Polimorfismo (III)

Objetivos

- Conocer cómo se puede **identificar en tiempo de ejecución el tipo de un objeto polimórfico**.
- Saber crear y utilizar **plantillas de clase** en C++.

Índice

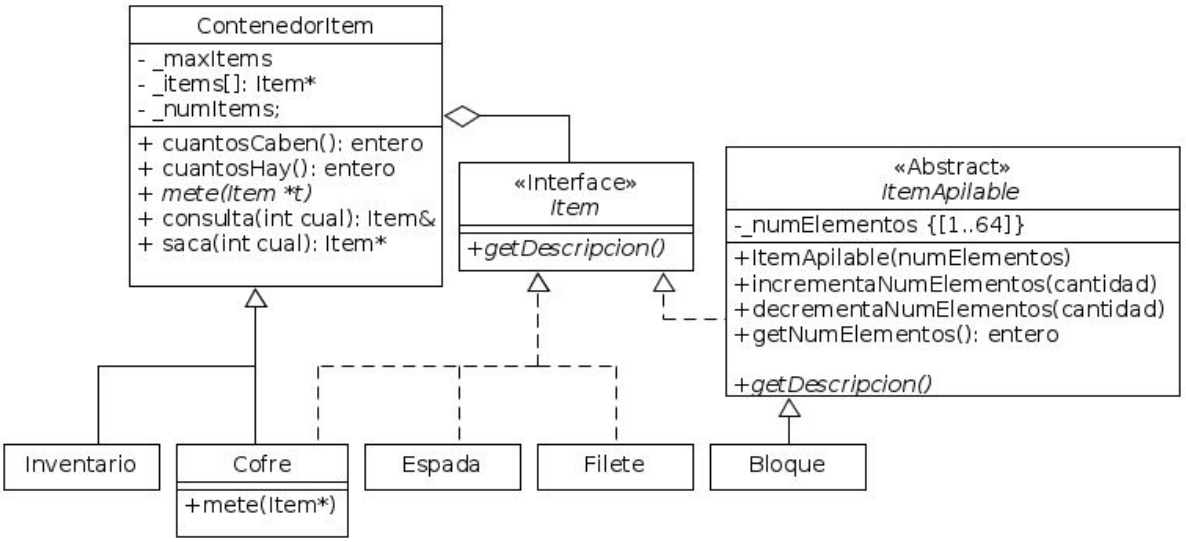
Contexto de la práctica

Continuando con el diseño de la nueva versión de Minecraft en tu plan por cambiar el futuro, decides ahora pasar a refactorizar tu diseño para que pueda gestionarse el inventario del personaje.

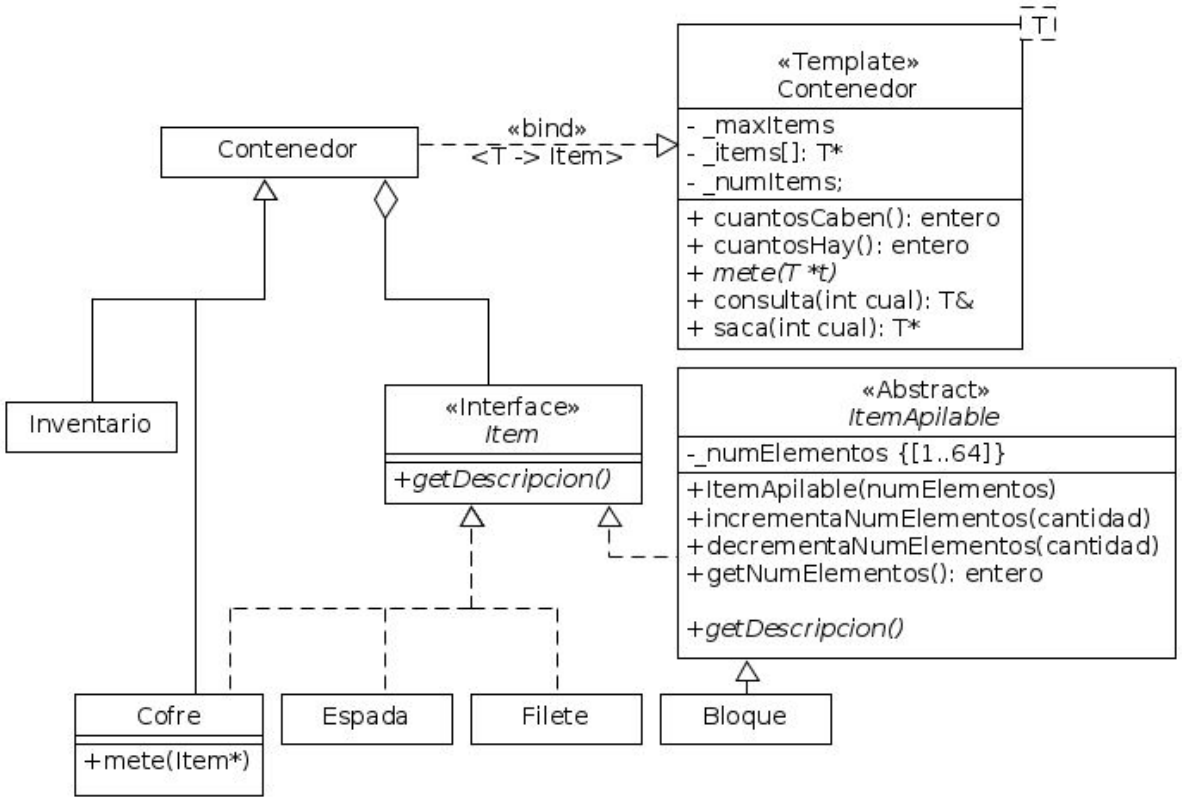
El inventario del personaje es similar a lo que ya teníamos implementado en la clase Cofre, aunque dispone de algunos ítems adicionales que tienen un uso específico: ítems de armadura equipados, escudo utilizado, ítems de acceso rápido, etc. Sin embargo, para la práctica actual nos centraremos sólo en su funcionalidad básica como contenedor de Items. Considerando sólo este aspecto del Inventario, recuerdas que

los profesores de POO te comentaron que cuando dos clases tenían un comportamiento idéntico quizás había que plantearse generalizar dicho código en una clase ancestro común, por lo que decides trasladar ese comportamiento a la clase *ContenedorItem*.

Sin embargo, en la dinámica del juego no es posible meter un cofre dentro de otro, aunque estos sí pueden formar parte del inventario. Para solventar este punto, dedices que los Cofres también implementen la interfaz de *Ítem* para que puedan añadirse a los Inventarios. Por otro lado, es necesario redefinir el método *Cofre::mete* para evitar que un cofre no se pueda meter en otro. Estos cambios quedan reflejados en el siguiente diseño.



También, gracias a ese poder de "programación defensiva" que estás desarrollando últimamente, te das cuenta que la clase *ContenedorItem* podrías reutilizarla en muchas más partes del juego y no sólo para almacenar Items: Personajes, Armaduras, Monstruos, Strings, u objetos de cualquier otra clase que pudiera crearse posteriormente. Por tal motivo, cuando todo esté funcionando, decides hacer una ¿última? refactorización del diseño para transformar esta clase en una plantilla de clase:



Punto de partida

El punto de partida de esta práctica es el programa de prueba con la solución de la práctica anterior. El proyecto NetBeans está disponible en el siguiente enlace: http://bit.ly/poouja1617pr10

Ejercicios

1. Transformar la clase Cofre en una **clase ContenedorItems** para poder reutilizar su comportamiento de contenedor de Items en otros tipos de contenedores. El método *mete* será de tipo virtual para que pueda redefinirse en clases derivadas. Volver a crear la clase Cofre, esta vez como derivada de la nueva clase *ContenedorItems*.
2. Definir la nueva **clase Inventario** como derivada de la nueva clase ContenedorItems. Hacer que la **clase Cofre implemente la Interfaz Item** para que pueda añadirse un cofre al inventario del personaje. Redefinir el método *getDescripcion()* de Cofre para que además muestre el número de elementos que contiene. Comprobar en el programa principal que podemos crear un objeto de tipo inventario y añadirle el cofre que ya teníamos.
3. **Modificar el programa principal sustituyendo el vector de punteros a Items** por un objeto de la nueva clase *ContenedorItems*. Modificar las funciones: *inicializaItems* y *liberaItems*, para inicializar y liberar respectivamente el contenedor con los objetos iniciales. Adaptar el resto del programa para obtener los objetos necesarios desde este contenedor en vez hacerlo desde el anterior vector.

4. **Modificar la función visualiza(Cofre &c)** para que acepte objetos de tipo *ContenedorItem* en general. Utilizarla en el programa principal para mostrar el objeto inventario creado.
5. **Redefinir el método mete de la clase cofre** para que no permita que un cofre se pueda meter dentro de otro. Para esto, usar RTTI con el operador *dynamic_cast<T>* para comprobar si el objeto a meter es de tipo Cofre, y lanzar una excepción del tipo *std::invalid_argument* si el ítem que se intenta añadir al cofre es otro Cofre.
6. En el programa principal, **recuperar el cofre del inventario del personaje** usando el método *consulta(cual)*. Comprobar que efectivamente es un cofre y, en su caso, mostrar el contenido por pantalla usando la función *visualiza*. Capturar adecuadamente las excepciones que pudieran originarse en su caso.
7. Transformar la clase *ContenedorItem* en la **plantilla de clase Contenedor<T>**. Modificar las clases *Inventario* y *Cofre* para que hereden de *Contenedor<Item>*. Modificar las funciones *inicializaItems*, *liberaItems* y *visualiza* para que acepten objetos de la nueva clase: *visualiza(Contenedor<Item>&),* y modificar el tipo del contenedor de Items en el programa principal por una instancia de la nueva plantilla.

Contenidos

Identificación de tipos en tiempo de ejecución

La utilización de métodos virtuales y clases abstractas se ha mostrado como una gran solución a la hora de incluir en un mismo vector de punteros a una superclase objetos de muy distinta índole, todos ellos derivados de dicha superclase. Posteriormente, dichos vectores pueden ser pasados a funciones, métodos, etc. de modo que se pueden tratar todos los objetos a los que apunta de forma homogénea ya que todos comparten las mismas funciones virtuales (definidas precisamente en la superclase).

No obstante, el hecho de que los distintos objetos puedan ser incluidos en un mismo vector conlleva una pequeña desventaja: perdemos la posibilidad de acceder a los atributos y métodos propios de cada uno de dichos objetos, de modo que sólo podemos pasarles los mensajes que han heredado de la superclase. Este inconveniente puede ser salvado gracias a la técnica de identificación de tipos en tiempo de ejecución.

Antes de explicar en qué consiste, es importante destacar que esta técnica (que actualmente implementan los compiladores de C++) debe ser usada como último recurso, siempre de forma justificada. Un uso excesivo de esta técnica podría ser un indicio de que hemos realizado un mal diseño de la jerarquía de clases y debería llevarnos a contemplar la posibilidad de volver a realizar el diseño y cambiar, consecuentemente, nuestro código.

Una de las formas de utilizar la identificación de tipos en tiempo de ejecución es utilizar el operador **dynamic_cast<T>()**. Este operador intenta transformar un puntero o referencia a un objeto polimórfico en alguna de sus posibles clases derivadas, representada por T. Si la conversión tiene éxito, devuelve respectivamente el puntero o referencia al objeto de la nueva clase. Dependiendo si tratamos de convertir un puntero o una referencia el comportamiento difiere ligeramente:

- Si **T es de tipo puntero**, devuelve el puntero al objeto original o *nullptr* si el objeto no era de tipo T
- Si **T es una referencia**, devuelve una referencia al objeto original o lanza excepción std::bad_cast

El siguiente ejemplo muestra una función a la que se le pasa un *SerVivo* y muestra sus propiedades; no obstante, un *SerVivo* puede ser animal o vegetal siendo las propiedades de cada uno distintas. Vemos dos versiones de uso de dynamic_cast<T>() usando punteros y referencias para comprender sus diferencias:

a) Identificación de tipos pasando a dynamic_cast un puntero a objeto de la clase base:

```
void visualiza ( SerVivo &sv ) {
	std::cout << sv.getFamilia()<<endl;
	std::cout << sv.getEspecie() << endl;
	Vegetal *pVegetal=dynamic_cast<Vegetal*>(&sv);//pasamos dirección de memoria
	if( pVegetal ) {		//!=nullptr si el objeto es de la clase Vegetal
		if( pVegetal->getReproduccionPorSemillas() )
			std::cout << "Se reproduce por semillas"<<endl;
		if( pVegetal->getReproduccionPorPropagacionVegetativa() )
			std::cout << "Se reproduce por propagacion vegetativa"<<endl;
	}
}
```

b) Identificación de tipos pasando a dynamic_cast una referencia a objeto de la clase base:

```
void visualiza ( Ser_Vivo &sv ) {
	cout << sv.getFamilia()<<endl;
	cout << sv.getEspecie() << endl;
	try {
		Vegetal &v = dynamic_cast<Vegetal&>(sv);  //pasamos referencia
		if( v.getReproduccionPorSemillas() )
			cout << "Se reproduce por semillas"<<endl;
		if( v.getReproduccionPorPropagacionVegetativa() )
			cout << "Se reproduce por propagacion vegetativa"<<endl;
	} catch (std::bad_cast &e) {
		//No es vegetal
	}
}
```

A la función anterior podríamos llamar pasándole distintos tipos de variables, por ejemplo:

```
	visualiza( perro );   // perro es de tipo Animal
	visualiza( geranio ); // geranio es de tipo Vegetal
	visualiza( *pRosal ); // pRosal es de tipo Vegetal*
```

Por supuesto, también podemos implementar una función que reciba un vector con punteros a Seres Vivos, e ir mostrándolos todos:

```
void visualiza ( SerVivo* sv[], int tama ) {
	int i;
	for (i = 0; i < tama; ++i) {
		if ( dynamic_cast<Vegetal*>( sv[i] ) ) {
			cout << "Es un vegetal " << endl;
		}else if ( dynamic_cast<Animal*>( sv[i] ) ) {
		}else if (...) {
		}else {
			cout << "Es una especie desconocida";
		}
	}
}
```

Plantillas de clase

La declaración de **plantillas de clase** sigue la misma sintaxis que en el caso de las plantillas de función, precediendo a su definición de la palabra reservada *template* donde, nuevamente, pueden ir declarados uno o varios tipos parametrizados así como parámetros con tipo fijo.

El objetivo de las plantillas de clase es crear una colección de posibles instanciaciones de clases que comparten la misma interfaz(métodos) y comportamiento (implementaciones), aunque difieren entre sí en la representación, puesto que algún/os atributos tendrán el tipo parametrizado.

Existe una diferencia con las plantillas de función a la hora de realizar la instanciación de una plantilla de clase a un tipo concreto. Esta diferencia consiste en que es necesario indicar expresamente, entre < >, cuales serán los tipos/valores específicos para los que se realizará la instanciación de la plantilla de clase. Sin embargo, al igual que ocurría con las plantillas de función, aunque esta instanciación sintáctica pudiera realizarse a priori para cualquier conjunto de tipos/constantes, hay que tener en cuenta que dependiendo de los detalles concretos de la definición de la plantilla, es posible que la instanciación no sea posible debido a incompatibilidades de los tipos indicados con la implementación de la propia plantilla. P.e. por ser necesaria la implementación de cierto método para el tipo instanciado.

Una plantilla de clase frecuentemente utilizada suele ser la clase `Pareja<T1,T2>`. Esta clase `Pareja`, convenientemente instanciada, permite crear objetos formados por la composición de dos objetos que pueden ser de la misma clase o de clases diferentes. Esto puede ser muy interesante en ciertas situaciones en las cuales necesitamos establecer una asociación entre ambos objetos.

```
/*---
Ejemplo de definición de la plantilla de clase Pareja
--- */
template<typename T1,typename T2>
class Pareja {
	T1 primero;
	T2 segundo;
```

```
public:
	Pareja(T1 p=T1(), T2 q=T2() ): primero(p),segundo(q) {}
	T1& getPrimero() {return primero;}
	T2& getSegundo() {return segundo;}
	void setPrimero( T1 &val) {primero=val;}
	void setSegundo(T2 &val) {segundo=val;}
};
```

Con esta plantilla de clase podríamos realizar las siguientes instanciaciones:

```
Pareja<double, double>     punto1(0,0), punto2(2.5,4.5);
Pareja<Persona,Persona>    proyectoPOO( Persona("María"),Persona("Carlos") );
Pareja<Persona*,Persona*> matrimonio(  new Persona("Ana"),
                                       new Persona("Jaime"));
Pareja<string,double> precios[]={      Pareja<string,double>("patatas",1.15),
                                       Pareja<string,double>("calabacín",0.99)};
```