Un ejemplo práctico

Jordi Brínquez Jiménez

PID_00161679

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Este módulo pretende dotar a los estudiantes de un ejemplo completo del proceso de desarrollo de una aplicación partiendo de cero, para que puedan asimilar mejor los contenidos de los módulos anteriores, y también para que se familiaricen con los procedimientos para resolver problemas.

Este módulo está organizado en tres grandes bloques:

- Una primera parte en la que planteamos un enunciado que nos servirá para el resto del módulo.

- A continuación, crearemos el diagrama UML resultante del problema planteado.

- Finalmente, resolveremos paso a paso los problemas de codificación que puedan surgir.

Objetivos

Los objetivos de este módulo son:

1. Facilitar la comprensión de los conceptos explicados durante los módulos anteriores, para que los estudiantes seáis capaces de resolver correctamente la práctica de la asignatura.

2. Dar al estudiante de un modelo de resolución de problemas mediante la programación orientada a objetos.

1. Enunciado

La UOC quiere añadir una aplicación nueva al conjunto de utilidades que hay en el campus para facilitar el estudio al estudiante.

La aplicación en concreto es un sistema gestor de preguntas y pruebas que permitirá realizar y corregir en línea tests de cualquier temática. De este modo, los usuarios (los estudiantes) pueden evaluar sus conocimientos sobre cualquier tema que sea de su interés, y así poder prepararse para exámenes reales.

Para optar a este servicio, hay que estar registrado como estudiante dentro del sistema de la UOC.

El equipo docente podrá crear y añadir preguntas en el sistema, a partir de las cuales se podrán generar los tests. De este modo, se pueden aprovechar preguntas para más de un test.

Es necesario que los tests sean dados de alta en el sistema para que los estudiantes los puedan resolver. El proceso que hay que seguir para elaborar un test consiste en distinguir qué preguntas hay que añadirle de entre todas las preguntas dadas de alta en el sistema. Una característica que se quiere que tenga el sistema es que permita crear los tests en diferentes instantes de tiempo, es decir, que permita empezar a crear un test, dejarlo a medias y continuarlo más adelante. Hasta que un test no esté acabado, no puede estar disponible para los estudiantes.

De momento, se han definido una serie de posibles tipos de preguntas, aunque no se descarta que se puedan ampliar en el futuro. Los diferentes tipos de preguntas son:

- Preguntas textuales. Este tipo de preguntas esperan una respuesta textual, en forma de cadena de texto. Hay que tener en cuenta que las respuestas deben ser cortas para que se puedan corregir fácilmente de manera automatizada y, por lo tanto, nunca se tendría que pedir que se desarrollara un tema.

  Preguntas que entrarían en este formato:

  - ¿Cuál es la capital de Dinamarca?
  - ¿Por qué nombre se conoce también *La Gioconda*?

- Preguntas de opción múltiple. Éstas son las preguntas de test típicas, en las que se dan diferentes opciones y hay una única respuesta correcta. Una pregunta de opción múltiple debe tener, como mínimo, dos respuestas posibles.

Sobre el funcionamiento respecto del estudiante, se desea poder seleccionar de la lista de los tests disponibles dentro del sistema el test que quiere resolver. Cuando el estudiante selecciona un test para resolverlo, éste pasa a estar disponible para el estudiante. Un estudiante puede tener como máximo un test disponible para resolverlo en cada momento. El estudiante no debe tener ninguna limitación de tiempo y tiene que poder contestar las preguntas en el orden que crea necesario. Cuando el estudiante indique al sistema que ya ha acabado el test, se considerará cerrado y no se podrá modificar ninguna respuesta. En este instante, se procederá a la corrección automática del test. Todas las preguntas no contestadas se considerarán erróneas y, en las otras, se deberán comprobar las respuestas dadas con las respuestas almacenadas. Un estudiante sólo puede realizar un test una única vez.

Se considera de gran utilidad que se pueda acceder a la historia de todas las pruebas realizadas por un estudiante. Se quiere que figuren cuáles son las pruebas que ha realizado, la nota de la prueba y, en caso de que tenga algún test a medias, es necesario que también se refleje.

Se quiere mantener un registro de la respuesta que el estudiante ha dado a cada pregunta de cada test, para poder dar una realimentación o *feedback* al estudiante que indique los temas que tiene que repasar. Esta realimentación es un texto asociado con cada pregunta independiente de las respuestas del estudiante.

Para intentar aclarar un poco qué datos queremos que se almacenen, os ofrecemos una lista de atributos de algunas entidades. Por otra parte, se tienen que almacenar aquellos datos que surjan de la lectura del enunciado.

- La entidad *Estudiante* tiene los atributos siguientes:
  - Un identificador único dentro del sistema
  - Nombre
  - Apellidos
  - Dirección
  - Teléfono
  - Correo electrónico

- La entidad *Test* tiene los atributos siguientes:
  - Un identificador único dentro del sistema
  - Título (nombre que se da a la prueba)
  - Descripción (descripción corta, genérica, que hace referencia a toda la prueba)

- La entidad *Pregunta* tiene los atributos siguientes:
  - Un identificador único dentro del sistema
  - Enunciado de la pregunta
  - Realimentación (en caso de respuesta errónea)
  - Respuesta correcta
  - Puntuación de la pregunta

A partir de la descripción del problema que queremos resolver, os mostramos las operaciones que queremos que nuestro sistema permita realizar:

- La *Gestión de preguntas* ha de permitir:
  - Añadir una pregunta
  - Consultar una pregunta
  - Hacer una lista de todas las preguntas del sistema

- La *Gestión de tests* ha de permitir:
  - Añadir un test
  - Consultar un test
  - Añadir una pregunta a un test
  - Eliminar una pregunta de un test
  - Hacer una lista de las preguntas de un test
  - Hacer una lista de todos los tests del sistema
  - Finalizar un test (hacerlo disponible para que los alumnos lo resuelvan)

- La *Gestión de estudiantes* ha de permitir:
  - Añadir a un estudiante
  - Modificar los datos de un estudiante
  - Consultar los datos de un estudiante
  - Hacer una lista de todos los estudiantes del sistema

- La *Resolución de tests* ha de permitir:
  - Iniciar la resolución de un test por un estudiante
  - Contestar una pregunta del test
  - Dar un test por finalizado

- La *Evaluación* ha de permitir:
  - Consultar la nota de un estudiante en una prueba
  - Consultar las notas de un estudiante

2. Resolución

Para resolver este problema, en primer lugar tenemos que generar un diagrama UML con las clases que nos servirán para modelar la aplicación y, posteriormente, debemos codificar la solución en el lenguaje de programación que utilicemos. Esta separación es necesaria, porque el lenguaje de programación no tiene que condicionar en ningún momento nuestro diseño. Vistas las características de un lenguaje determinado, en algunos casos podríamos incurrir en errores de diseño.

2.1. Identificación de las clases

Inicialmente, como hemos comentado, habrá que identificar las clases que formarán parte de nuestro modelo, y también sus atributos.

Para hacer la lista de clases necesarias subrayaremos todos los sustantivos y las frases nominales del texto.

Ved el método para identificar clases explicado en el módulo "Abstracción y clasificación".

La UOC quiere añadir una aplicación nueva al conjunto de utilidades que hay en el campus para facilitar el estudio al estudiante.

La aplicación en concreto es un sistema gestor de preguntas y pruebas que permitirá realizar y corregir en línea tests de cualquier temática. De esta manera, los usuarios (los estudiantes) pueden evaluar sus conocimientos sobre cualquier tema que sea de su interés, para poder prepararse para exámenes reales.

Para optar a este servicio, hay que estar registrado como estudiante dentro del sistema de la UOC.

El equipo docente podrá crear y añadir preguntas en el sistema, a partir de las cuales se podrán generar los tests. De esta manera, se pueden aprovechar preguntas para más de un test.

Es necesario que los tests sean dados de alta en el sistema para que los estudiantes los puedan resolver. El proceso que hay que seguir para elaborar un test consiste en distinguir qué preguntas hay que añadirle de entre todas las preguntas dadas de alta en el sistema. Una característica que se quiere que tenga el sistema es que permita crear los tests en diferentes instantes de tiempo, es decir, que permita empezar a crear un test, dejarlo a medias y continuarlo más adelante. Hasta que un test no esté acabado, no puede estar disponible para los estudiantes.

De momento, han definido una serie de posibles tipos de preguntas, aunque no descartan que en un futuro se puedan ampliar. Los diferentes tipos de preguntas son:

Preguntas textuales. Este tipo de preguntas esperan una respuesta textual, en forma de cadena de texto. Hay que tener en cuenta que las respuestas deben ser cortas para que se puedan corregir fácilmente de manera automatizada y, por lo tanto, nunca se tendría que pedir que se desarrollara un tema.

Preguntas de opción múltiple. Éstas son las típicas preguntas de test, en las que se dan diferentes opciones y hay una única respuesta correcta. Una pregunta de opción múltiple debe tener, como mínimo, dos respuestas posibles.

> Sobre el funcionamiento respecto del estudiante, quieren que pueda seleccionar de la lista de los tests disponibles dentro del sistema el test que quiere resolver. Cuando el estudiante selecciona un test para resolverlo, éste pasa a estar disponible para el estudiante. Un estudiante puede tener como máximo un test disponible para resolverlo en cada momento. El estudiante no debe tener ninguna limitación de tiempo y tiene que poder contestar las preguntas en el orden que crea necesario. Cuando el estudiante indique al sistema que ya ha acabado el test, se considerará cerrado y no se podrá modificar ninguna respuesta. En este instante, se procederá a la corrección automática del test. Todas las preguntas no contestadas se considerarán erróneas y, en las otras, habrá que comprobar las respuestas dadas con las respuestas almacenadas. Un estudiante sólo puede realizar un test una única vez.
>
> Se considera de gran utilidad que se pueda acceder a la historia de las pruebas realizadas por un estudiante. Se quiere que figuren cuáles son las pruebas que ha realizado, la nota de la prueba y, en caso de que tenga algún test a medias, es necesario que también se refleje.
>
> Se quiere que se mantenga un registro de la respuesta que el estudiante ha dado a cada pregunta de cada test, para poder dar una realimentación o *feedback* al estudiante que indique los temas que tiene que repasar. Esta realimentación es un texto asociado con cada pregunta independiente de las respuestas del estudiante.

Después de leerlo otra vez y subrayar los sustantivos y las frases nominales, obtenemos la lista siguiente:

- La UOC
- Aplicación
- Campus
- Estudiante
- Sistema gestor de preguntas y pruebas
- Tests
- Temática
- Usuarios
- Exámenes reales
- El equipo docente
- Preguntas
- Preguntas textuales
- Respuesta textual
- Respuestas
- Pregunta de opción múltiple
- Respuesta correcta
- Preguntas no contestadas
- Respuestas almacenadas
- Historia de todas las pruebas realizadas por un estudiante
- Nota de la prueba
- Test a medias
- Registro de la respuesta que el estudiante ha dado a cada pregunta de cada test
- Realimentación

Una vez tenemos esta lista, hay que eliminar las clases incorrectas. Si seguimos la metodología explicada en el módulo "Abstracción y clasificación", llegaremos a las conclusiones siguientes:

a) Las clases siguientes son redundantes

– Sistema gestor de preguntas y pruebas: es un sinónimo de aplicación.

– Usuarios: representa el mismo concepto que estudiantes.

b) Las clases siguientes son irrelevantes

- Campus: es un concepto que no utilizaremos para resolver el problema, pero que podría ser necesario tenerlo en cuenta en otros problemas.
- Exámenes reales: sólo denota que los tests almacenados en el sistema simulan exámenes reales, pero esto no es de ninguna utilidad para nuestro sistema.
- Respuesta textual: es un concepto que quiere añadir información al concepto de `respuesta`, pero que no aporta nada nuevo.

c) Las clases siguientes son vagas

- La UOC: es una clase sobre la cual no se nos pide que almacenemos ningún dato directamente.
- Temática: es un concepto que se menciona, pero no se utiliza en ningún sitio ni tiene una finalidad bien definida.

d) Los conceptos siguientes resultan ser atributos

- Nota de la prueba: este concepto no es una clase en sí misma, sino una propiedad de una relación entre un estudiante y un test.
- Realimentación: es un concepto inherente a la pregunta; por lo tanto, tiene que ser un atributo suyo.
- Respuesta correcta: deberemos tener cierto cuidado a la hora de tratar este concepto, porque hay distintos tipos de preguntas.

e) Hallamos los siguientes roles:

- Estudiante: es un rol de `Persona`, como lo es el concepto de equipo docente, pero en este caso, el equipo docente será el encargado de insertar los datos en el sistema.

f) Sobre las estructuras o conceptos de la implementación

- Respuestas almacenadas: no será una clase directamente, pero sí un concepto que tenemos que almacenar para cada test resuelto de cada estudiante.
- Preguntas no contestadas: si en vez de almacenar el resultado de todas las preguntas, almacenamos las preguntas contestadas con la respuesta dada (tal y como pide el enunciado), podemos obtener las preguntas no contestadas sin que sea necesario tener una clase que las represente.
- Historia de todas las pruebas realizadas por un estudiante: este concepto es una lista de las pruebas realizadas, no una clase, pero habrá que tenerlo en cuenta durante el diseño, ya que tenemos que garantizar su obtención.
- Registro de la respuesta que el estudiante ha dado a cada pregunta de cada test: este concepto no es directamente una clase, pero hay que tenerlo en cuenta a la hora de realizar el diseño.

– Test a medias: es un concepto que no es directamente una clase, pero hay que tenerlo presente durante el diseño.

Veamos qué clases son las que nos han quedado del proceso de filtrado anterior:

- Aplicación
- Estudiante
- Tests
- Preguntas
- Preguntas textuales
- Pregunta de opción múltiple
- Respuestas

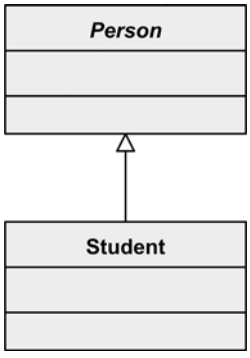
2.2. Creación de diagramas parciales de clases

Con las clases de la lista anterior y el enunciado, construiremos el diagrama de clases de nuestra aplicación.

Más que construir un diagrama de clases entero, lo que tenemos que hacer es crear diagramas pequeños que más tarde juntaremos para crear el diagrama definitivo.

En primer lugar, nos fijamos en los usuarios del sistema. Tal y como hemos comentado anteriormente, la clase `Student` (`Estudiante`) es una especialización de `Person` (`Persona`). Puesto que en principio no tendremos ninguna instancia de la clase `Person` (o como mínimo el enunciado no nos dice nada al respecto), la podemos definir como una clase abstracta y posteriormente, en caso de futuras ampliaciones del sistema, podemos heredar de ésta para crear los nuevos roles.

De esta manera, obtenemos este trozo del diagrama final:

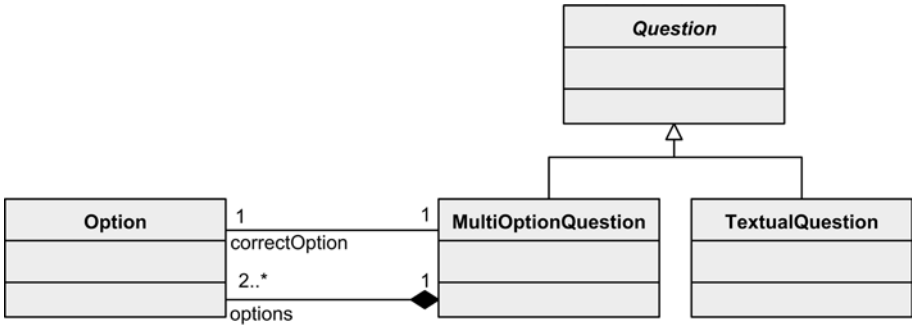


A continuación, nos fijaremos en la estructura de las preguntas del test. Repasemos el enunciado para tener claras las relaciones que hay:

> El equipo docente podrá crear y añadir preguntas en el sistema, a partir de las cuales se podrán generar los tests. De esta manera, se pueden aprovechar preguntas para más de un test.
>
> De momento, han definido una serie de posibles tipos de preguntas, aunque no descartan que, en un futuro, se puedan ampliar. Los diferentes tipos de preguntas son:
>
> - Preguntas textuales. Este tipo de preguntas esperan una respuesta textual, en forma de cadena de texto. Hay que tener en cuenta que las respuestas deben ser cortas para que se puedan corregir fácilmente de manera automatizada y, por tanto, nunca se tendría que pedir que se desarrollara un tema.
>
> - Preguntas de opción múltiple. Estas preguntas son las típicas preguntas de test, en las que se dan diferentes opciones y hay una única respuesta correcta. Una pregunta de opción múltiple debe tener, como mínimo, dos respuestas posibles.

Dado que ahora nos concentramos en el apartado de las preguntas, es necesario que distingamos del enunciado el texto que se refiere al mismo.

Puesto que existe una entidad abstracta denominada *preguntas*, de la cual se dice que hay de distintos tipos (textuales y de opción múltiple), podemos deducir que hay una relación de especialización/generalización (herencia) entre estas clases, que podemos modelar de la manera siguiente:



Como podéis ver, tenemos una clase abstracta, `Question` (`Pregunta`), que nos ayuda a organizar la jerarquía de clases de las preguntas. Además, hemos representado gráficamente que una `MultiOptionQuestion` (`PreguntaMultiOpcion`) tiene asociadas unas respuestas, una de las cuales es correcta.

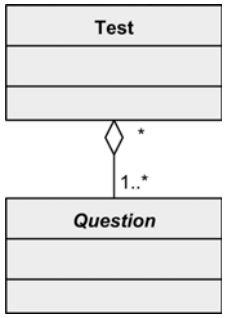
Habría que expresar que la respuesta correcta debe ser una de las respuestas que forman parte de la lista de opciones posibles. En UML se puede representar gráficamente, pero ya que no lo hemos explicado en esta asignatura, pondremos la restricción de palabra, que también es una opción válida.

Expresar condiciones sobre los valores, relaciones u otros conceptos de un diagrama UML se denomina crear una "restricción textual".

Después, al analizar la relación entre los tests y las preguntas, hallamos lo siguiente:

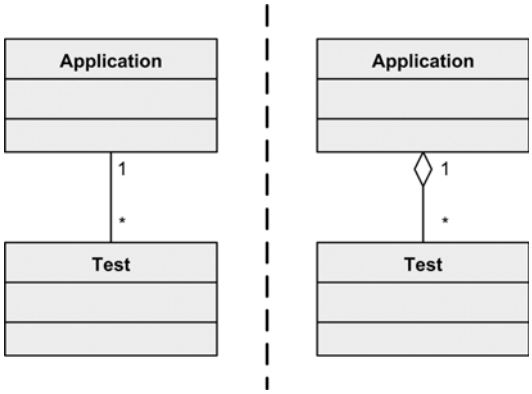
> El proceso que hay que seguir para elaborar un test consiste en distinguir qué preguntas hay que añadir de entre todas las preguntas dadas de alta en el sistema.

Por lo tanto, `Test` es una agregación de preguntas, y lo representaremos así:



Esta relación es una agregación, ya que las preguntas tienen sentido fuera del test (es decir, puede haber preguntas sin ningún test asociado). Si las preguntas no tuvieran sentido si no hubiera un test, en lugar de una agregación sería una composición.

Una vez en este punto, sólo nos queda ver cómo se organizan los tests dentro de la estructura de la aplicación; por lo tanto, puesto que la UOC quiere almacenar todos los tests y las preguntas, podemos crear dos diagramas diferentes según el significado que queremos que tenga dentro de nuestra aplicación:

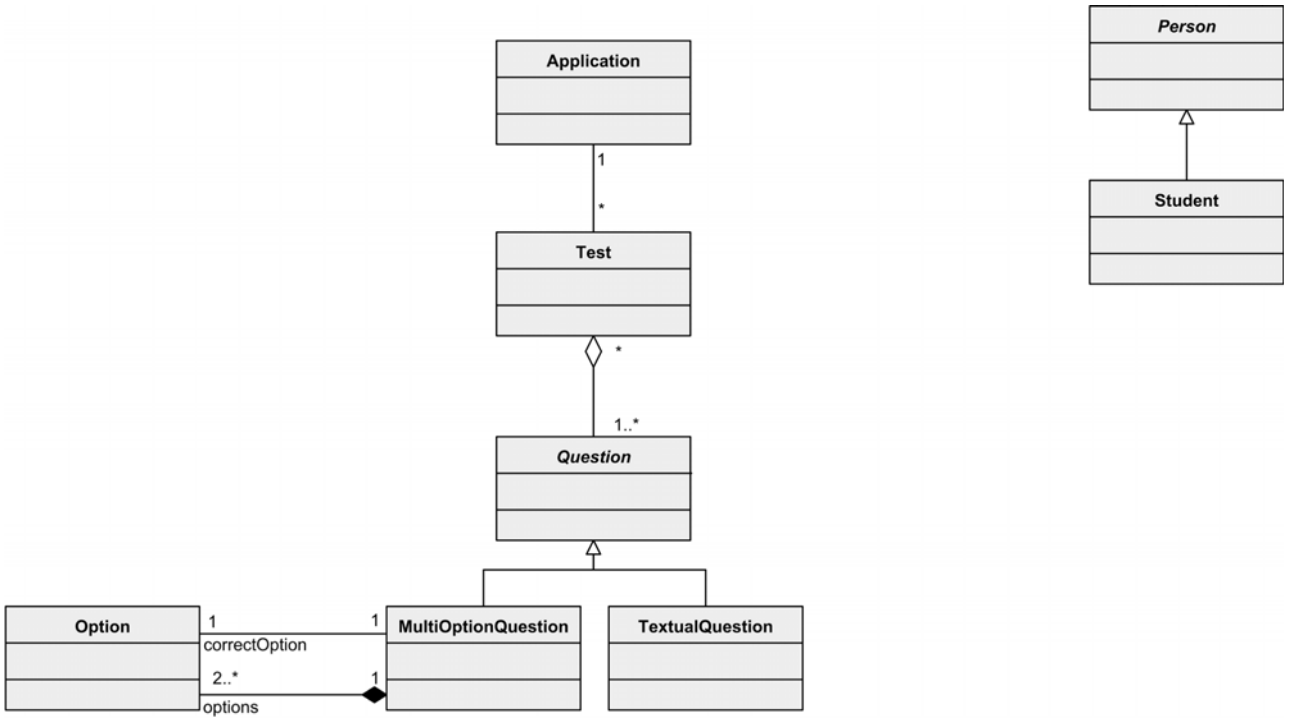


En el diagrama de la derecha, hay una relación de agregación entre `Application` (`Aplicacion`) y `Test;` por lo tanto, indicamos que la aplicación tiene tests y que éstos forman parte de la aplicación. En el otro diagrama, en cambio, tanto el test como la aplicación tienen entidad por sí mismos, y lo único que los une es una relación mutua.

Las dos soluciones son válidas y representan nuestro problema; la elección de una o de la otra sólo depende del significado que queremos que tenga esta relación. Nosotros utilizaremos el diagrama de la izquierda, el que representa la relación entre las clases.

2.3. Creación del diagrama de clases

Aquí ya podemos juntar todos los diagramas pequeños para crear un diagrama preliminar con todas las clases:



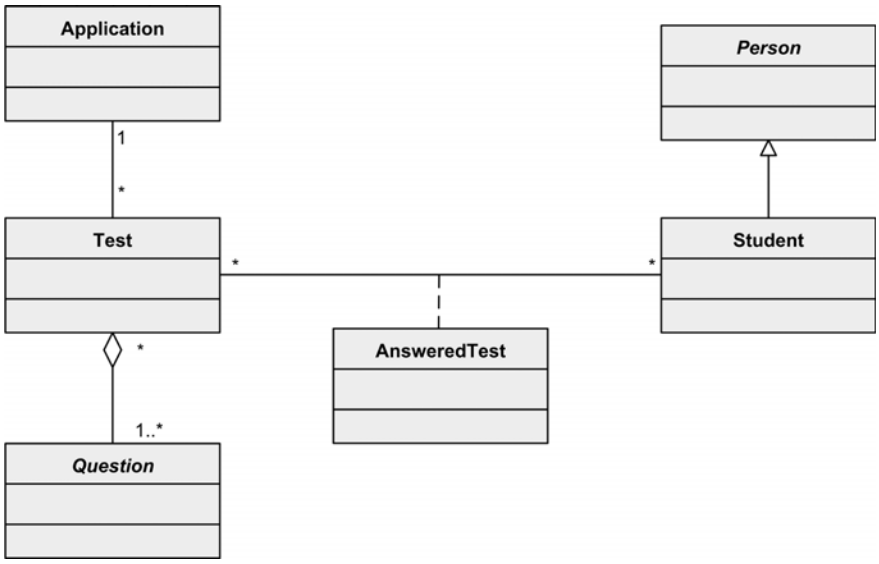
Como podéis ver, falta enlazar a los estudiantes con el resto de las clases, principalmente con la clase `Application`, y añadir todo lo relacionado con la resolución de los tests por parte de los alumnos, que empezaremos a resolver a continuación.

Revisamos una vez más el enunciado:

> Sobre el funcionamiento respecto del estudiante, quieren que pueda seleccionar de la lista de los tests disponibles dentro del sistema el test que quiere resolver. Cuando el estudiante selecciona un test para resolverlo, éste pasa a estar disponible para el estudiante. Un estudiante puede tener como máximo un test disponible para resolverlo a cada momento. El estudiante no debe tener ninguna limitación de tiempo y tiene que poder contestar las preguntas en el orden que crea necesario. Cuando el estudiante indique al sistema que ya ha acabado el test, se considerará cerrado y no se podrá modificar ninguna respuesta. En este instante, se procederá a la corrección automática del test. Todas las preguntas no contestadas se considerarán erróneas y, en las otras, habrá que comprobar las respuestas dadas con las respuestas almacenadas. Un estudiante sólo puede realizar un test una única vez.
>
> Se considera de gran utilidad que se pueda acceder a la historia de las pruebas realizadas por un estudiante. Se quiere que figuren cuáles son las pruebas que ha realizado, la nota de la prueba y, en caso de que tenga algún test a medias, es necesario que también se refleje.
>
> Se quiere mantener un registro de la respuesta que el estudiante ha dado a cada pregunta de cada test, para poder dar una realimentación o *feedback* al estudiante que indique los temas que tiene que repasar. Esta realimentación es un texto asociado con cada pregunta independiente de las respuestas del estudiante.

Por lo tanto, tenemos que un `Student` (`Estudiante`) sólo puede tener "asignado" un `Test` en cada momento para resolverlo y, una vez resuelto –es decir, cuando se considere cerrado–, este estudiante no lo puede volver a resolver nunca más. Del concepto de resolución de un `Test` por parte de un estudiante (`Student`) hay que almacenar algunos datos (nota, preguntas contestadas, etc.); por lo tanto, lo que tenemos es una clase asociativa entre `Student` y `Test`, que

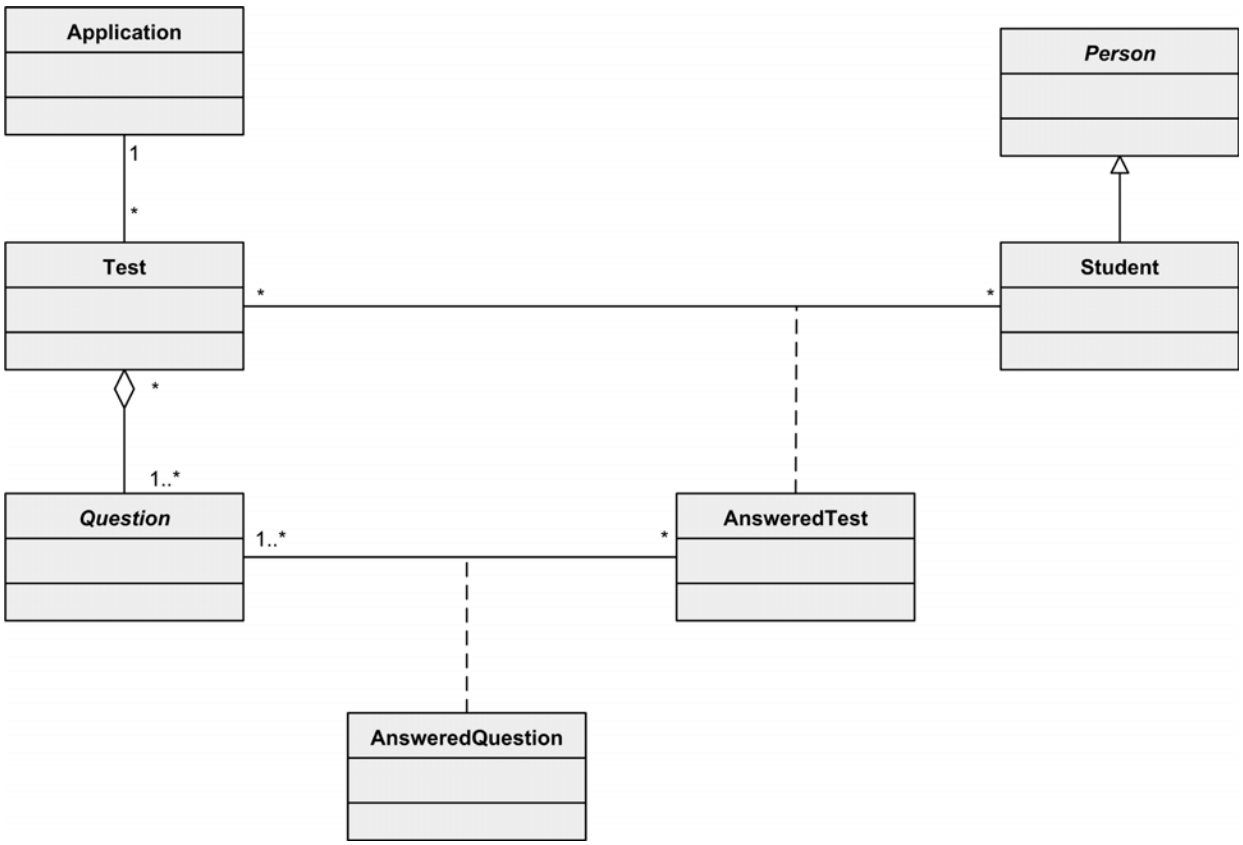
denominaremos `AnsweredTest` (`TestContestado`). Si añadimos estas modificaciones al diseño, resulta el diagrama siguiente:



Hemos eliminado del diagrama toda la estructura en lo que respecta a las `Preguntas`, ya que no aportaban ninguna información para esta parte.

Hay que representar el concepto de que un test es activo para que lo pueda resolver un alumno. Para resolver este concepto, utilizaremos un atributo de tipo booleano que pondremos en la clase `AnsweredTest` y sólo podremos tener una instancia activada para cada estudiante.

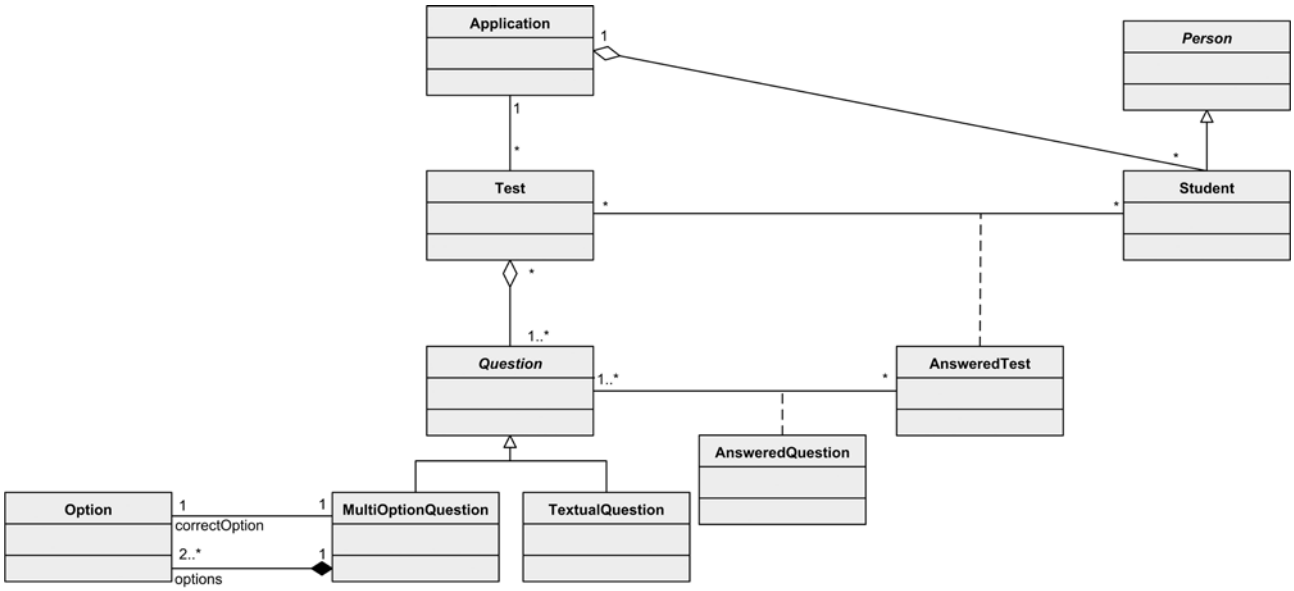
Ahora que ya tenemos representado el concepto de resolución de un test, nos falta poder mantener las respuestas dadas para cada pregunta del test, que podemos representar mediante otra clase asociativa entre `AnsweredTest` y `Question`, que denominaremos `AnsweredQuestion` (`PreguntaResuelta`).



Y entonces queda relacionar a los estudiantes (`Student`) con la aplicación, a partir de este punto del enunciado:

> Para optar a este servicio, hay que estar registrado como estudiante dentro del sistema de la UOC.

Veamos que tenemos que crear una relación entre `Application` y `Student` para tener el enunciado representado completamente en nuestro diagrama, con lo cual quedaría de la manera siguiente:



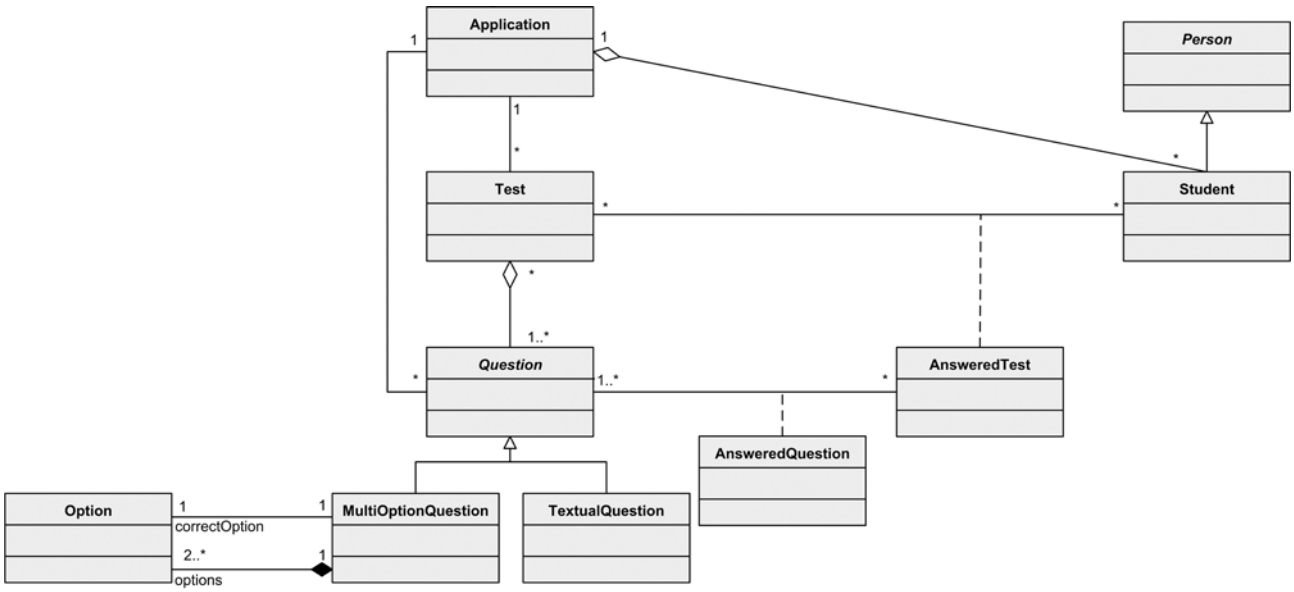
Aunque parezca acabado, hay que repasar el diagrama varias veces para asegurarnos de que cumple todos los requisitos del problema dado. Debemos tener en cuenta que el enunciado del problema es un texto que puede inducir a distintas interpretaciones según la persona que lo lea, pero un diagrama de clases tiene una única interpretación.

Si repasamos el enunciado una vez más, podemos llegar a ver que nos falta una de las relaciones que el enunciado menciona, concretamente la relación entre preguntas y aplicación.

> El equipo docente podrá crear y añadir preguntas en el sistema, a partir de las cuales se podrán generar los tests. De esta manera, se pueden aprovechar preguntas para más de un test.

Por lo tanto, hay que añadir una nueva relación entre `Application` y `Question`. En este caso, existe el mismo problema que había antes entre `Application` y `Test`. Hay dos interpretaciones posibles: nosotros, como hemos hecho antes, representaremos una relación binaria entre las clases, y no utilizaremos la agregación.

Finalmente, el diagrama de clases resultante es el siguiente:



2.4. Asignación de atributos y métodos

Una vez tenemos terminado el diagrama de clases, hay que llenarlo añadiendo los atributos y métodos que el enunciado nos pide, y también aquéllos resultantes de las decisiones de diseño que hayamos podido tomar durante el proceso anterior.

Es decir, para cada clase del diagrama anterior tenemos que indicar los atributos y los métodos.

2.4.1. Clase `Person`

Como indica el enunciado, la clase `Person` debe tener los atributos siguientes:

- Un identificador único dentro del sistema
- Nombre
- Apellidos
- Dirección
- Teléfono
- Correo electrónico

Puesto que necesitamos tener un identificador único en el sistema para poder asignar a cada persona un identificador diferente, es necesario que definamos un atributo de clase para almacenar este valor.

Por otra parte, tendremos un método constructor al cual deberemos dar todos estos atributos (a excepción del identificador, que dado que será único en el sistema, se asigna en el momento de crear la instancia).

Entre otros métodos que podemos necesitar, existen los que nos permiten consultar los datos de la persona, también denominados *getters*, que se definen de esta manera:

```
fieldType getFieldName();
```

También tenemos que definir los métodos que nos permitirán modificar los datos, denominados *setters*, que se definen de manera similar a los *getters*.

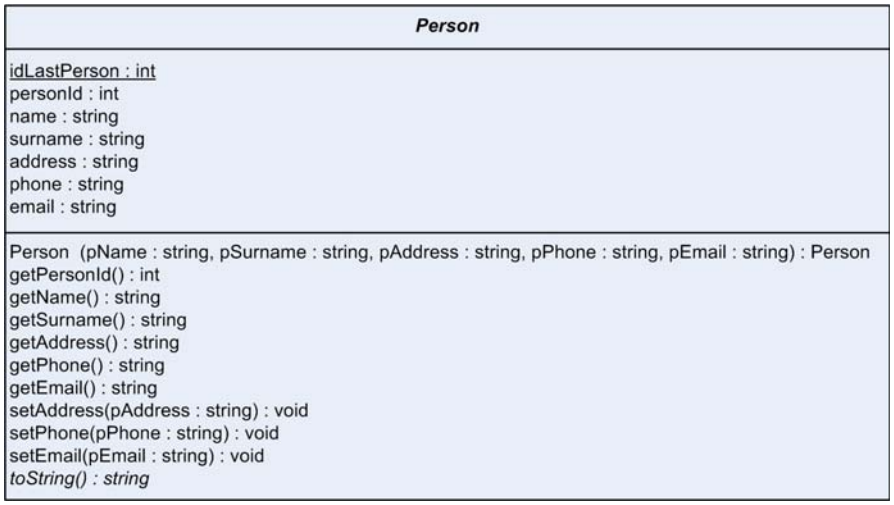
El nombre de *getters* y *setters* deriva del significado de las palabras *get* y *set* en inglés ("recuperar" y "poner").

```
void setFieldName(fieldType value);
```

En este caso no necesitamos todos los métodos *setter*, ya que en una persona no cambiaremos el identificador, ni el nombre y apellidos, pero sí que podría ser que tuviéramos que cambiar la dirección, el teléfono o el correo electrónico.

Podríamos añadir un *getter* y un *setter* para el atributo `personId` (`identificadorDePersona`), pero puesto que es un atributo de clase que sólo leeremos y modificaremos en el método constructor de la clase, nos podemos ahorrar añadirlos al diagrama con el objetivo de ganar claridad.

La clase `Person` resultante es la siguiente:



2.4.2. Clase `Student`

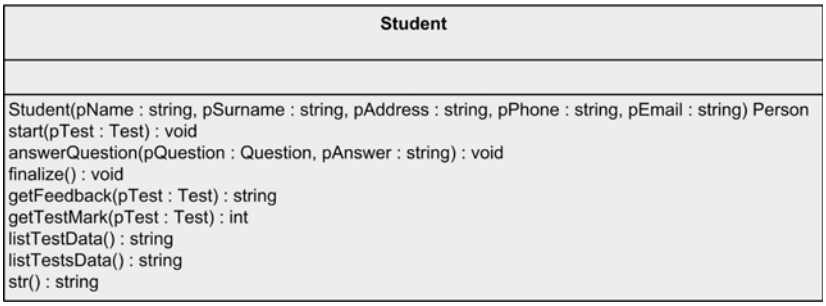
En este caso, puesto que la clase `Student` hereda de la clase `Person`, debemos tener en cuenta que todos los atributos y métodos definidos en la clase `Per-`

`son` es como si estuvieran definidos en la clase `Student`; por lo tanto, tenemos que analizar qué diferencia a la persona del estudiante para ver los atributos y métodos que debemos definir en esta clase.

En cuanto a los atributos, el enunciado no nos indica que tengamos que añadir ninguno. En lo que respecta a los métodos, aparte de necesitar un método constructor con los mismos parámetros que la clase `Person`, deberemos permitir crear una lista de los datos de los estudiantes (una especie de resumen de todos los datos) y también los tests realizados por el estudiante; por lo tanto, hará falta que añadamos un par de métodos que ofrezcan esta información en forma de cadena de texto.

Además, tendremos que añadir un método para empezar a resolver un test, otro para contestar las preguntas y otro para finalizar su resolución. Y también necesitaremos un par de métodos para recuperar la nota y los comentarios de los tests ya realizados.

Por lo tanto, la clase `Student`, añadiendo los métodos propuestos, queda de la siguiente manera :



2.4.3. Clase `Question`

Como indica el enunciado, una pregunta debe tener los campos siguientes:

- Un identificador único dentro del sistema
- Enunciado de la pregunta
- Realimentación (en caso de respuesta errónea)
- Respuesta correcta
- Puntuación de la pregunta

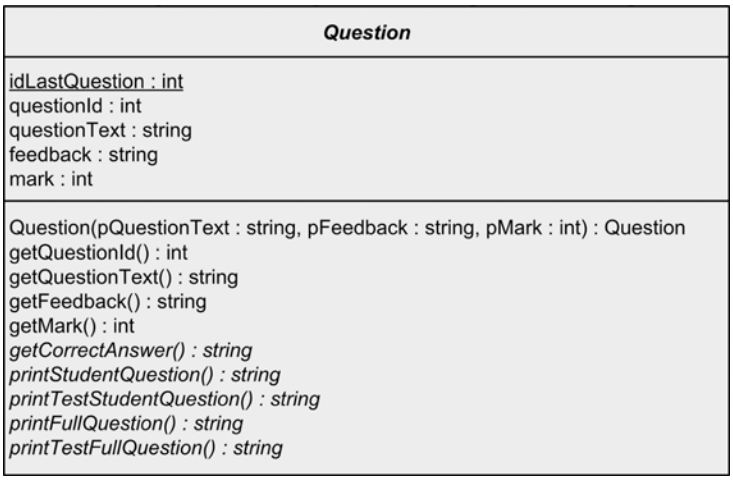
Como nos pasa con la clase `Person`, necesitamos un atributo de clase para almacenar el último identificador asignado y otro de instancia para almacenar el identificador de cada instancia.

El enunciado de la pregunta y su realimentación y puntuación también serán atributos y, puesto que estamos definiendo una clase abstracta, dejaremos para las clases que lo especializan la definición de la respuesta correcta, ya que en cada una de las mismas habrá que definirla de una manera o de otra.

Sobre los métodos de esta clase, tendremos, como en casos anteriores, los *getters* de los atributos y la operación constructora que, a pesar de tenerla

definida, redefiniremos en las clases que la especializan. De los *getters*, habrá uno que definiremos como abstracto, porque cada subclase lo definirá de una manera o de otra. También definiremos un método que nos permita mostrar todos los datos de la pregunta y otro que nos dé los datos de la pregunta preparados para ser mostrados al estudiante.

A partir de las premisas anteriores, la clase `Question` queda definida así:

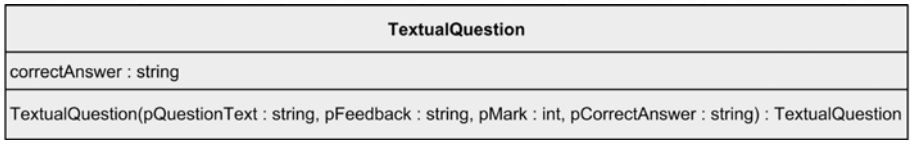


2.4.4. Clase `TextualQuestion`

Como en el caso del `Student`, la clase `TextualQuestion` (`PreguntaTextual`) hereda de otra clase, en este caso de `Question` ; por lo tanto, como hemos hecho antes, no hay que volver a definir los atributos y métodos ya descritos.

Lo que sí deberemos hacer es definir la representación de la pregunta correcta que, para la clase `TextualQuestion`, será un atributo de tipo `String`, y también tendremos que definir su operación constructora.

De este modo, la clase `TextualQuestion` queda definida como en la siguiente figura:



2.4.5. Clase `MultiOptionQuestion`

En el caso de la `MultiOptionQuestion`, nos sucede algo parecido a la del caso anterior, pero dado que tenemos que almacenar las diferentes opciones, la respuesta correcta y el número de la opción, deberemos modificar la opera-

ción constructora para que refleje estos datos y tendremos que añadir el atributo que nos permita almacenar la respuesta correcta.

MultiOptionQuestion
correctAnswer : Option
MultiOptionQuestion(pQuestionText : string, pFeedback : string, pMark : int, pCorrectAnswer : int, pOptions : Option[]) : MultiOptionQuestion

2.4.6. Clase `Option`

La clase `Option` sólo tiene que almacenar el texto de cada pregunta; por lo tanto, tendrá un atributo con el texto de la opción, una operación constructora y otra consultora del texto almacenado.

Option
optionText : string
Option(pOptionText : string) : Option getOptionText() : string

2.4.7. Clase `Test`

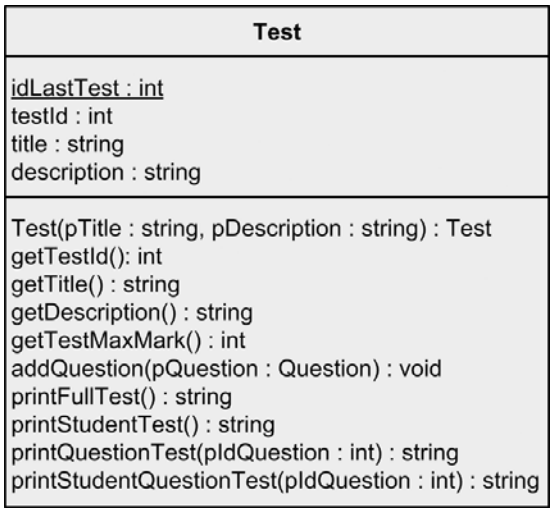
Esta clase debe almacenar los datos referentes al test que hemos indicado anteriormente:

- Un identificador único dentro del sistema
- Título (nombre que se le da a la prueba)
- Descripción (descripción corta, genérica, que hace referencia a toda la prueba)

Dados estos atributos y los métodos típicos (constructor, *getters* y *setters*), veamos qué nos dice el enunciado que tenemos que poder hacer sobre un test:

> Es necesario que los tests sean dados de alta en el sistema para que los estudiantes los puedan resolver. El proceso que hay que seguir para elaborar un test consiste en distinguir qué preguntas hay que añadirle de entre todas las preguntas dadas de alta en el sistema. Una característica que se quiere que tenga el sistema es que permita crear los tests en diferentes instantes de tiempo, es decir, que permita empezar a crear un test, dejarlo a medias y continuarlo más adelante. Hasta que un test no esté acabado, no puede estar disponible para los estudiantes.
>
> ...
>
> Sobre el funcionamiento respecto del estudiante, quieren que pueda seleccionar de la lista de los tests disponibles dentro del sistema el test que quiere resolver. Cuando el estudiante selecciona un test para resolverlo, pasa a estar disponible para el estudiante. Un estudiante puede tener como máximo un test disponible para resolverlo en cada momento. El estudiante no debe tener ninguna limitación de tiempo, y tiene que poder contestar las preguntas en el orden que crea necesario. Cuando el estudiante indique al sistema que ya ha acabado el test, éste se considerará cerrado y no se podrá modificar ninguna respuesta. En este instante, se procederá a la corrección automática del test. Todas las preguntas no contestadas se considerarán erróneas y, en las otras, habrá que comprobar las respuestas dadas con las respuestas almacenadas.

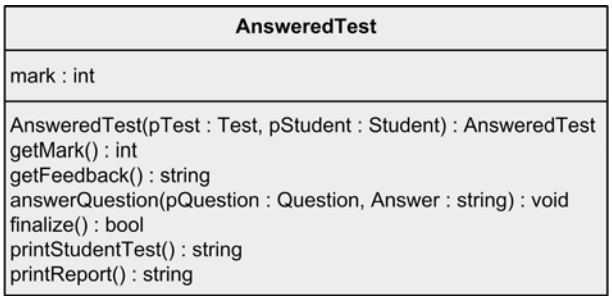
Por lo tanto, debemos ofrecer también un método para añadir una pregunta al test, un método que nos permita crear una lista con los datos de una pregunta y otro que nos permita crear una lista del test completo (tanto para el alumno como para el equipo docente); finalmente, tenemos que dar la nota máxima del test para que el estudiante pueda evaluar posteriormente su resultado.



2.4.8. Clase `AnsweredTest`

En este caso, en un test contestado, al tratarse de una clase asociativa, debemos almacenar la relación entre un objeto de la clase `Test` y otro de la clase `Student`.

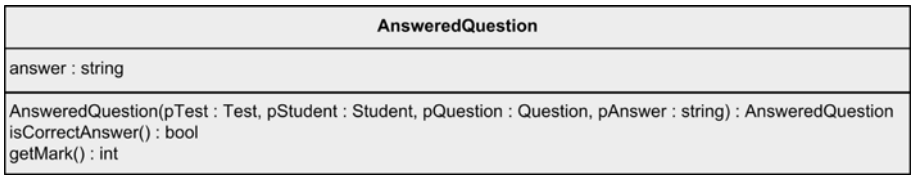
En esta clase, aparte de la relación entre las clases, tenemos que almacenar la nota del test y si el alumno tiene el test a medias. Definiremos este concepto de la siguiente manera: un test a medias es aquel test que el alumno tiene asignado para su resolución, pero que todavía no tiene una nota asignada. Si nos fijamos en los métodos que esta clase debe tener, necesitamos un método que nos permita dar el test por acabado –y por lo tanto, poder corregirlo y asignarle la nota–, y también otro método para almacenar la respuesta de cada una de las preguntas.



2.4.9. Clase `AnsweredQuestion`

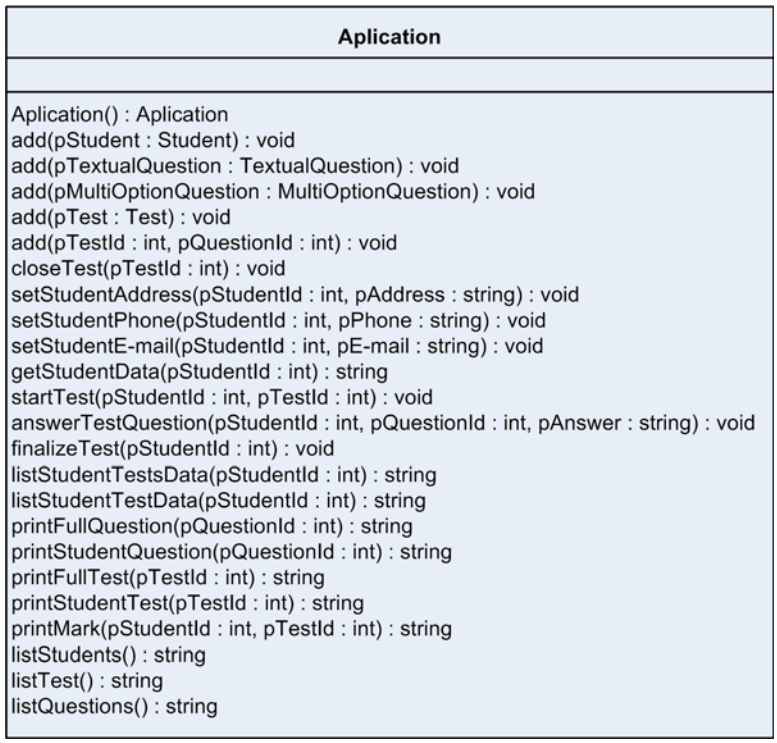
Esta clase nos servirá para almacenar las respuestas dadas por un estudiante a una pregunta de un test. No tenemos que almacenar ningún otro dato, pero

sí que debemos definir un método que nos permita comprobar si la respuesta dada es correcta o no.



2.4.10. Clase `Application`

En esta clase, en cambio, según lo planteado en el problema que hay que resolver, tenemos que definir una gran cantidad de métodos para poder realizar todas las tareas que necesitamos que haga.



2.5. Construcción del diagrama completo

Una vez hemos completado todas las clases, es recomendable dibujar el diagrama con las clases completas para ver si necesitamos algún dato más o si nos hemos dejado algún método.

Notad que la distribución ha cambiado para facilitar su dibujo.

2.6. Inclusión de la navegabilidad en el diagrama de clases

Puede parecer que el diagrama anterior ya está acabado y que ya nos podemos poner a teclear el código directamente, pero todavía nos falta un último paso para poder decir que hemos completado el diagrama: añadir la navegabilidad.

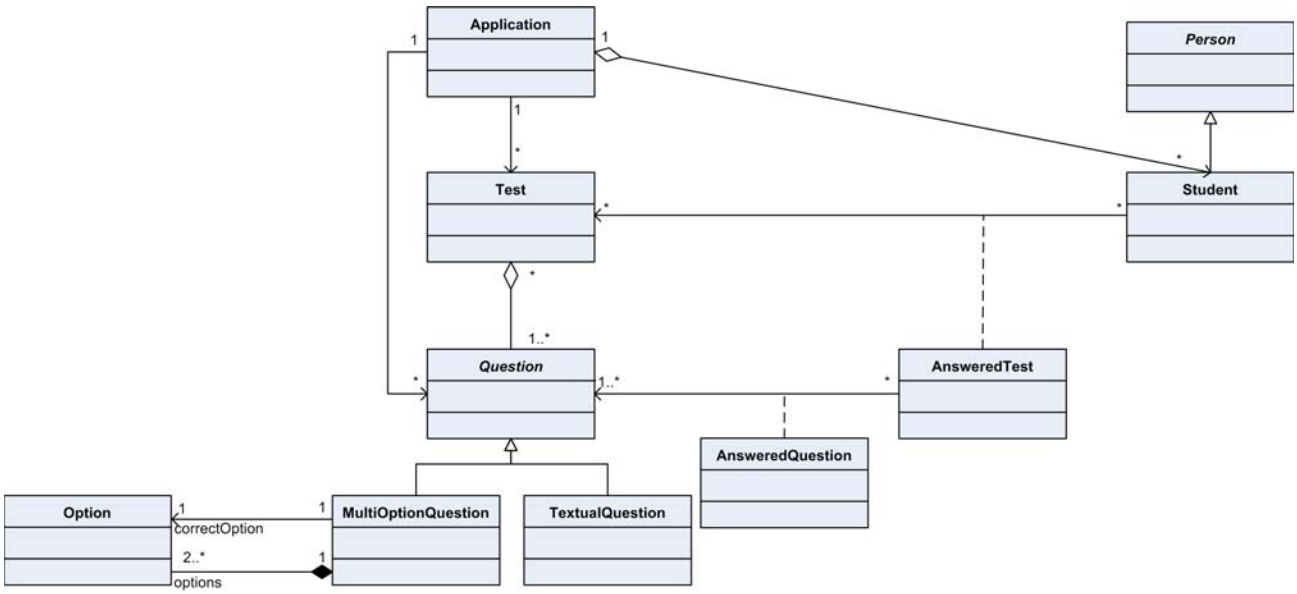
Como ya hemos visto, la navegabilidad nos muestra qué clases tienen constancia de la existencia de las otras; por ejemplo, en nuestro caso, la aplicación debe saber qué tests tiene, pero un test no hace falta que sepa en qué aplicación está, ya que no necesita invocar ninguna operación de ésta.

Para asignar la navegabilidad correctamente, hay que volver a leer el texto del problema y asignar la navegabilidad donde corresponda.

Después de leer de nuevo el enunciado, podemos definir las navegabilidades siguientes:

- `Application` → `Test`
- `Application` → `Student`
- `Application` → `Question`
- `Student` → `Test`
- `MultiOptionQuestion` → `Option`
- `AnsweredTest` → `Question`

El diagrama de clases final queda de la manera siguiente (hemos eliminado los atributos y métodos nuevamente para facilitar la comprensión).



2.7. Codificación

Sobre la codificación, sólo comentaremos algunos cambios que se han producido según el diseño original (el diagrama de clases anterior), debido a las facilidades/limitaciones que tiene el lenguaje de programación Java que utilizaremos.

Para empezar, la clase `Option` ha sido eliminada y transformada en una lista de `strings` dentro de la clase `MultiOptionQuestion`. Esta decisión ha sido tomada porque la clase no tiene ninguna otra utilidad que almacenar unas respuestas para cada pregunta y, de esta manera, se simplifica ligeramente el diseño.

Para la implementación, hay que comentar que no se debería hacer todo al mismo tiempo; es decir, se tiene que estructurar la codificación inicialmente en partes independientes y, poco a poco, las tenemos que juntar para crear finalmente la aplicación pedida.

Una recomendación es empezar implementando las clases que no dependen de ninguna otra clase, que en nuestro problema son `Person` y `Question`.

Una vez las hemos implementado, no las podemos probar directamente, ya que son clases abstractas. Por lo tanto, tenemos que continuar implementando las clases que heredan de las dos clases anteriores, `Student`, `TextualQuestion` y `MultiOptionQuestion`. Podemos implementar y probar estas clases creando pequeños programas que accedan a todos sus métodos para saber si funcionan correctamente.

Una vez ya hemos implementado la parte relativa a las preguntas y a las personas, lo más adecuado será implementar la clase `Test` (ya tenemos la clase `Question` y las clases que heredan); por lo tanto, podemos implementar sin ninguna dificultad la clase `Test` y la agregación de test y preguntas.

Llegados a este punto, es interesante crear ya la clase `Application`, porque de esta manera establecemos el vínculo entre tests, preguntas y personas y, posteriormente, implementaremos las clases que nos permiten utilizar las clases asociativas `AnsweredTest` y `AnsweredQuestion`.

No podemos implementar totalmente la clase `UOC` sin haber implementado las clases `AnsweredTest` y `AnsweredQuestion`, pero una vez hayamos implementado estas clases, lo podemos dejar casi todo terminado para acabar definitivamente todo el código.

Cómo podéis ver, resolver un problema se reduce a resolver problemas más pequeños que, todos juntos, forman el problema inicial.

El código y la documentación necesarios para resolver este enunciado los podéis encontrar en vuestra aula.

Resumen

En este módulo hemos visto cómo, desde un texto que nos describe un problema, hemos ido descomponiendo el problema en unidades pequeñas que hemos resuelto de manera casi mecánica.

Inicialmente hay que identificar las entidades del problema, cosa que hemos hecho siguiendo un método sencillo que nos servirá en la mayoría de los casos, aunque la experiencia es siempre la mejor amiga en estas tareas.

Posteriormente hemos creado relaciones entre estas entidades sin tener en cuenta el contenido exacto de éstas, pero pensando en el concepto que representan y la relación que hay entre las entidades del mundo real que queremos representar.

Más adelante, hemos llenado de contenido estas entidades añadiéndoles atributos y métodos, de manera que no sólo son entidades abstractas, sino que las hemos dotado de significado.

Una de las tareas en las que tenemos que ser más críticos es la comprobación de que el modelo construido representa nuestro problema, dado que nos podría pasar que alguna cosa de las que necesitamos representar no se puede representar por alguna limitación de nuestro modelo.

Posteriormente, hemos procedido a codificar el modelo en el lenguaje determinado.

Ejercicios de autoevaluación

1. Se decide crear un nuevo tipo de pregunta tipo test de respuesta múltiple. ¿Qué se tendría que modificar para poder introducir este nuevo tipo?

2. ¿Qué se debería cambiar del modelo original si no nos interesara mantener un historial de los tests resueltos?

3. ¿A qué clases afectaría la creación de un nuevo atributo en la clase `Persona`?

4. Si queremos añadir un contador para cada test que nos diga cuántas veces se ha resuelto, ¿en qué clase tenemos que poner este atributo? ¿A qué clases afectaría?

Solucionario

1. Para incluir este nuevo tipo de pregunta, lo que tendríamos que hacer es crear una nueva subclase de `Pregunta` con características parecidas a las de la clase `PreguntaMultiOpcion`, pero que permitiera almacenar respuestas múltiples.
Otra manera de resolver este problema sería modificar la clase `PreguntaMultiOpcion` y permitir que acepte directamente más de una respuesta válida. En este caso, si sólo hay una respuesta válida, funcionará de la misma manera que la clase del modelo original.

2. Únicamente tendríamos que cambiar la multiplicidad de la relación entre la clase `Estudiante` y la clase `Test`. Deberíamos modificar el símbolo `*` del lado de la clase `Test` por un 1, y de esta manera sólo tendremos un test almacenado para cada estudiante.
Consiguientemente, deberemos eliminar de la clase `UOC` y la clase `Estudiante` los métodos que permiten acceder a la información de este registro histórico.

3. La inclusión de un nuevo atributo en la clase `Persona` hace que tengamos que modificar la clase `Estudiante` (el método constructor debe permitir un parámetro nuevo) y la clase `UOC` para que acepte este parámetro nuevo, tanto en la creación de estudiantes como en los métodos de acceso (*accessors*) y modificadores.
No tenemos que modificar ninguna otra clase más, ya que este cambio es transparente para las otras clases.

4. Este atributo lo tenemos que poner únicamente en la clase `Test`; debe ser un atributo privado con un único método de acceso para consultar su valor y otro para incrementarlo en una unidad.
Las clases afectadas únicamente serían dos: la clase `Test`, por la inclusión del atributo y del método, y la clase `Estudiante`, que debería invocar el método cada vez que el estudiante inicia la resolución de un test.