PRÁCTICA 4
MVC: finalizando el programa

Diseño y programación orientada a objetos

Semestre 20192

Estudios de Informática, Multimedia y Telecomunicación

Presentación

Esta Práctica pretende que el estudiante trabaje con un entorno profesional de desarrollo y se enfrente a situaciones reales que todo programador se suele encontrar. Entre estas situaciones está la creación de tests y el desarrollo de aplicaciones siguiendo el patrón MVC (Modelo-Vista-Controlador). Gracias a todo esto, el estudiante acabará el programa iniciado con la Práctica 1.

Objetivos

Los objetivos que se desean lograr con esta Práctica son:

- Practicar los conceptos estudiados en la asignatura.
- Profundizar en el uso de un IDE, en este caso, Eclipse.
- Tener un primer contacto con la programación orientada a eventos y la creación de interfaces gráficas utilizando JavaFX y SceneBuilder.
- Conocer y tener un primer contacto con alguno de los plugins de Eclipse. Concretamente, JUnit.
- Tener un primer contacto con el desarrollo de tests unitarios.

Enunciado

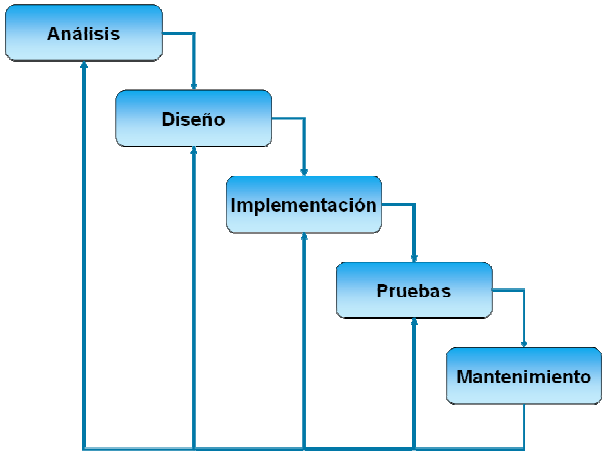
Esta Práctica está formada por 4 ejercicios.

Para hacer esta Práctica no tienes que leer más apuntes teóricos, sólo debes aplicar los conocimientos y competencias adquiridos a lo largo del semestre. Lo único nuevo que debes leer son las indicaciones que te damos en este mismo enunciado y algunos apartados de la nueva "Guía de Java (preliminar)". Quizás, para hacer algún apartado, tendrás que buscar información en Internet o en libros. También debes tener presente que se dan por asimilados conocimientos de algoritmia, que has estudiado y practicado en la asignatura anterior.

Para explicar y entender los ejercicios de la Práctica, creemos que es necesario proporcionarte información extra que es interesante a la vez que útil para ti. Por ello, te invitamos a leer detenidamente los apartados "Metodología en cascada o *Waterfall*" y "Patrón Modelo-Vista-Controlador (MVC)" que hemos escrito antes de los enunciados de los diferentes ejercicios. Asimismo, comentarte que el Ejercicio 4 tiene una parte introductoria acerca de los tests que creemos que también puede ser de tu interés.

Metodología en cascada o *Waterfall*

La metodología clásica para diseñar y desarrollar software se conoce con el nombre de metodología en cascada (o en inglés, *Waterfall*). Aunque ha sido reemplazada por nuevas variantes, es interesante conocer la metodología original. En su versión clásica esta metodología está formada por 5 etapas secuenciales (ver siguiente figura):



La etapa de **análisis** de requerimientos consiste en reunir las necesidades del producto/cliente. El resultado de esta etapa suele ser un documento de texto escrito por el equipo desarrollador (encabezado por la figura del analista) que describe las necesidades que dicho equipo ha entendido que necesita el cliente. No siempre el cliente se sabe expresar o es fácil entender lo que quiere. Normalmente este documento lo firma el cliente y es "contractual".

Por su parte, la etapa de **diseño** describe cómo es la estructura interna del producto (p.ej. qué clases usar, cómo se relacionan, etc.), patrones a utilizar, la tecnología a usar, etc. El resultado de esta etapa suele ser un conjunto de diagramas (p.ej. diagramas de clases UML, casos de uso, diagramas de secuencia, etc.) acompañado de información textual. Si nos decantamos en esta etapa por hacer un programa basado en programación orientada a objetos, será también en esta fase cuando usemos el paradigma *bottom-up* para la identificación de los objetos, de las clases y de la relación entre ellas.

La etapa de **implementación** significa programación. El resultado de esta fase es la integración de todo el código generado por los programadores junto con la documentación asociada.

Una vez terminado el producto, se pasa a la etapa de testeo o de **pruebas**. En ella se generan diferentes tipos de pruebas para ver que el producto final hace lo que se espera que haga. Evidentemente, durante la etapa de implementación también se hacen pruebas a nivel local (p.ej. a nivel de un método, una clase, etc.) para ver que esa parte, de manera independiente, funciona correctamente.

La última etapa, **mantenimiento**, se inicia cuando el producto se da por terminado. Aunque un producto esté finalizado, y por muchas pruebas que se hayan hecho, siempre aparecen errores (*bugs*) que se deben ir solucionando a posteriori.

Si nos fijamos en la figura, siempre se puede volver para atrás desde una etapa. Por ejemplo, si nos falta información en la etapa de diseño, siempre podemos volver por un instante a la etapa de análisis para recoger más información. Lo ideal es no tener que volver para atrás.

En esta asignatura hemos pasado, de alguna manera, por las cuatro primeras fases. Así pues, la etapa de **análisis** la hemos hecho desde el equipo docente. Nosotros nos “hemos reunido” con el cliente, en este caso, el usuario del UOCarium, y hemos analizado/documentado todas sus necesidades. A partir de estas necesidades, hemos ido tomando decisiones de **diseño**. Por ejemplo, decidimos que el software se basaría en el paradigma de la programación orientada a objetos y que usaríamos el lenguaje de programación Java. Una vez decididos estos aspectos clave, hemos ido identificando las diferentes clases (con sus atributos y métodos, i.e. datos y comportamientos) y las relaciones entre ellas a partir de las necesidades del cliente confirmadas en la etapa de análisis y de entidades reales que aparecen en el problema (i.e. objetos). Para ello hemos usado un paradigma *bottom-up*. Por ejemplo, a partir de casos concretos (i.e. objetos) de acuarios y elementos ubicados en ellos, hemos definido cómo debería ser las clases que modelaran estos conceptos. Como puedes imaginar, la etapa de diseño es una de las más importantes y determinantes de la calidad del software, ya que en ella se realiza una descripción detallada del sistema a implementar. Es importante que esta descripción permita mostrar la estructura del programa de forma que su comprensión resulte sencilla. Por ese motivo es frecuente que la documentación de la fase de diseño vaya acompañada de los diagramas UML, entre ellos los diagramas de clases (nosotros, con fines didácticos, hemos ido construyendo el diagrama de clases UML del software de UOCarium progresivamente a medida que iba avanzando el curso con especial

dedicación en la PRAC 3). Estos diagramas también son útiles en la fase de mantenimiento.

La etapa de **implementación** (o también conocida como desarrollo o codificación) ni decir cabe que la hemos tratado en esta asignatura. Obviamente, no hemos tratado temas que se dan por sabidos de asignaturas anteriores, principalmente relacionados con la algorítmica, i.e. uso de condicionales, bucles, estructuras simples como *arrays*, etc.

Finalmente, la etapa de **test/pruebas** la hemos tratado a partir de los ficheros "Check" que se proporcionaban con los enunciados de las PRACs y con alguna prueba extra que hayas hecho por tu cuenta. Estos ficheros permitían saber si las clases codificadas se comportaban como esperábamos. No obstante, ahondaremos en esta fase de testeo en esta Práctica 4.

Así pues, hemos tocado un poco todo y de forma paralela. Según la metodología en cascada, todas estas fases se deben hacer de manera secuencial. Así pues, primero deberíamos haber tenido claras las especificaciones y necesidades del cliente/software, posteriormente deberíamos haber tomado las decisiones oportunas y crear el diagrama de clases UML completo de todo el software, para luego codificarlo, y finalmente hacer pruebas de todo el programa. Insistimos que es lógico hacer pruebas locales (p.ej. un método) mientras se codifica.

Evidentemente, la metodología en cascada no es la única que existe, hay muchas más: por ejemplo, prototipado y las actualmente reconocidas, metodologías ágiles: eXtreme Programming, TDD (*Test-Driven Development*), etc. Si quieres saber más sobre metodologías para desarrollar software (donde se incluyen los patrones) y UML, te animamos a cursar la asignatura *Ingeniería del Software* si no la has hecho ya. A estas metodologías de desarrollo de software se les debe añadir las estrategias o metodologías de gestión de proyectos, como SCRUM, CANVAS, así como técnicas específicas para la gestión de los proyectos, p.ej. diagramas de Gantt y PERT, entre otros.

Patrón Modelo-Vista-Controlador (MVC)

Si abres con Eclipse el proyecto `UOCarium` que se te proporciona con este enunciado, verás que, en la ventana *Package Explorer*, la carpeta `src` del proyecto `UOCarium` tiene tres `packages` (i.e. paquetes). Hasta ahora sólo teníamos un `package` al cual llamábamos igual que al proyecto, aunque podríamos no haberle puesto nombre (en estos

casos Eclipse lo llama (`default package`)). Concretamente tendremos tres paquetes llamados `model`, `view` (dividido en `cmd` y `gui`) y `controller`. Lo hemos organizado así porque para esta Práctica usaremos el patrón de arquitectura de software llamado MVC (*Model-View-Controller*, i.e. modelo, vista y controlador). El patrón MVC es muy utilizado en la actualidad, especialmente en el mundo web. MVC intenta separar tres elementos clave de un programa:

- **Modelo**: representa la información del programa. En muchos programas esta parte recae en una base de datos y las clases que acceden a ella. El modelo se encarga de insertar, actualizar, eliminar y consultar la información del programa, así como de controlar los privilegios de acceso a dichos datos. Una alternativa a la base de datos es el uso de ficheros de texto y/o binarios (como es el caso de esta Práctica).
- **Vista**: es el conjunto de "pantallas" que configuran la interfaz con la que interactúa el usuario. Cada "pantalla" o vista puede ser desde una interfaz por línea de comandos hasta una interfaz gráfica, diferenciando entre móvil, tableta, ordenador, etc. Cada vista suele tener una parte visual y otra interactiva. Ésta última se encarga de recibir los *inputs* del usuario (p.ej. clic en un botón) y de comunicarse con el/los controlador/es del programa para pedir información o para informar de algún cambio realizado por el usuario. Además, según la información recibida por el/los controlador/es, modifica la parte visual en consonancia.
- **Controlador**: es la parte que controla la lógica del negocio. Hace de intermediario entre la vista y el modelo. Por ejemplo, mediante una petición del usuario (p.ej. hacer clic en un botón), la vista –mediante su parte interactiva– le pide al controlador que le dé el listado de ítems que tiene el software; el controlador le solicita esta información al modelo, el cual se la proporciona; el controlador envía la información a la vista que se encarga de procesar (i.e. parte interactiva) y mostrar (i.e. parte visual) la información recibida por el controlador.

Gracias al patrón MVC, se desacoplan las tres partes. Esto permite que teniendo el mismo modelo y el mismo controlador, la vista se pueda modificar sin verse alteradas las otras dos partes. Lo mismo, si cambiamos el modelo (p.ej. cambiar de gestor de base de datos de

MySQL a Oracle), el controlador y las vistas no deberían verse afectadas. Lo mismo si modificáramos el controlador. Así pues, con el uso del patrón MVC se minimiza el impacto de futuros cambios y se mejora el mantenimiento del programa. Si quieres saber más, te recomendamos que leas el artículo de Wikipedia sobre MVC: https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador

Hasta ahora (i.e. en las Prácticas) nos hemos centrado en una de las tres partes del patrón MVC, concretamente, el modelo.

Para definir el nombre de los `packages`, hemos usado la convención recomendada por Oracle en la web de tutoriales de Java: https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html. También tienes una explicación más detallada sobre los *packages* en el apartado 4.5 de la Guía de Java.

Fíjate que las clases tienen en su primera línea una sentencia que usa la palabra reservada `package`. Esto indica a qué paquete pertenece la clase (Eclipse pone esta sentencia automáticamente por nosotros). Recuerda que Java, además de `private`, `public` y `protected`, tiene un modo de ocultación (o visibilidad o niveles de acceso) llamado `package-private`, que es el modo por defecto (i.e. si no se especifica nada). De ahí también la importancia de decir a qué paquete pertenece cada clase.

Nota: A partir de aquí, lee y relee cuantas veces haga falta los siguientes ejercicios. Una lectura pausada te ayudará a entender tanto la lógica como el modelo de datos del programa.

Si no se dice lo contrario, las cardinalidades múltiples las debes codificar como una `ArrayList`.

Ejercicio 1 – Modelo (package `model`) (3 puntos)

En la Práctica 3 se te pidió que hicieras el diagrama de clases del programa completo en Dia. Este diagrama correspondía al modelo del software que estamos desarrollando para UOCarium. Esta Práctica 4 utilizará como punto de partida el diagrama de clases que desde el equipo docente te dimos como solución de la Práctica 3.

Nota: Junto con este enunciado te facilitamos la solución en formato JPG para que puedas abrirla con cualquier editor/visor (p.ej. Paint) y hacer zoom para verlo mejor. También puedes consultar la solución de la Práctica 3.

Para hacer este ejercicio y el resto de la Práctica, tendrás que trabajar sobre el proyecto `UOCarium` que te facilitamos con este enunciado. En dicho proyecto de Eclipse encontrarás las clases que hemos codificado hasta la solución de la PRAC 3 incluida.

Para hacer esta Práctica, además de la información proporcionada por el diagrama de clases, deberás tener en cuenta las especificaciones que se indiquen en este enunciado. A continuación te vamos a ir guiando para que puedas codificar el modelo completo.

a) Añade la clase `Keeper` que modela los cuidadores del acuario:

- Para cada cuidador (`keeper`) debemos guardar su id alfanumérico, nombre y apellido.
- También debemos guardar las referencias a los objetos de los tanques de los que el cuidador es responsable. En los tanques no debemos guardar qué cuidador/es es/son responsable/s.
- El número máximo de tanques que puede cuidar es 5.
- La clase `Keeper` sólo tendrá un constructor y será con parámetros: `id`, `name` y `surname`.
- Cada atributo de la clase tendrá un método *getter* y otro *setter*. Ambos métodos serán públicos.
- El método *setter* del atributo `id` debe comportarse de la siguiente manera:
    - Si el valor pasado como argumento es `null`, entonces debe lanzar una excepción de tipo `NullPointerException` sin mensaje.

    - Si el valor pasado como argumento no empieza por 'G' (en mayúscula), entonces debe lanzar una `KeeperException` (clase que codificaremos más adelante) con el mensaje `"[ERROR] A keeper's id must start with letter 'G'!!"`.
    - Si la longitud del valor pasado como argumento no es exactamente 5, entonces debe lanzar una `KeeperException` con el mensaje `"[ERROR] A keeper's id must have 5 characters!!"`.

- El método `addTank` añadirá el tanque pasado como argumento siempre y cuando el cuidador no tenga ya asignados 5 tanques y el tanque que se desea asignar no haya sido asignado ya al mismo cuidador. En caso de querer añadir un tanque cuando ya se han asignado 5 tanques al cuidador, se debe lanzar una `KeeperException` con el mensaje `"[ERROR] A keeper cannot take care of more than 5 tanks!!"`. Si a la hora de asignar el tanque, éste ya está asignado al cuidador, entonces este método no debe hacer nada: ni añadir el tanque ni lanzar una excepción.
- El método `toString` debe devolver un `String` con el siguiente formato:

```
[I] S, N:
    T
    T
```

  donde `I` es el identificador del cuidador, `S` su apellido y `N` su nombre. `T` es sólo el nombre del tanque que tiene asignado. Fíjate que aparecerán tantos tanques como se le hayan asignado y éstos estarán tabulados una vez respecto a los datos personales del cuidador. Cada nombre de tanque estará en una línea nueva.

(1 punto)

b) Crea la clase `KeeperException` de la misma manera que hemos ido creando el resto de excepciones personalizadas. Esta clase debe contener los mensajes de error propios de `Keeper`.

(0.25 puntos)

c) Ahora en la clase `Tank` hemos añadido un nuevo atributo llamado `id` con sus correspondientes *getter* y *setter*. Esto nos ha hecho introducir pequeñas modificaciones en su código, como por ejemplo en el constructor. En este sentido no debes hacer nada, sólo ser consciente de que hemos añadido dicho atributo.

A partir de aquí, haz que la clase `Tank` implemente la interfaz `Comparable` de la API de Java. Esto te obligará a sobrescribir el método `compareTo`. El comportamiento de dicho método debe ser que como parámetro sólo acepte un objeto de tipo `Tank` y que dados dos objetos `Tank`, los ordene alfabéticamente por el `id`, de la `A` a la `Z` (de menor a mayor; p.ej. el tanque `D1` antes que el `D2`, etc.).

(0.5 puntos)

d) En la Práctica 3 nos quedamos con que la clase `Fish` era una clase abstracta y para hacer las pruebas debíamos, o bien quitarle esta característica y luego añadírsela, o bien inventarnos una subclase. Ha llegado de añadir 3 subclases de `Fish`. Éstas serán `Danio`, `Tetra` y `Corydoras`, que son tres tipos de peces. El constructor de estas tres clases tienen la misma firma:

```
(double xCoord, double yCoord, Gender gender, int age, int energy, Tank tank)
```

Como ruta a la imagen del *sprite*, todos los objetos se deben crear con el mismo valor. Para los objetos de tipo `Danio` `"./images/danio/danio"`, para `Tetra` `"./images/tetra/tetra"` y para `Corydoras` `"./images/coyrdoras/corydoras"`.

Para los tres casos, tanto la longitud como la altura deben ser `64`.

La velocidad del `Danio` es `1`, del `Tetra` es `0.5` y del `Corydoras` `0.1`.

El valor de `requiredFoodQuantity` es `0.2`, `0.3` y `0.5` para `Danio`, `Tetra` y `Corydoras`, respectivamente.

Por su parte, el valor de `thresholdReverse` es `0.002`, `0.002` y `0.001` para `Danio`, `Tetra` y `Corydoras`, respectivamente.

El valor del color predominante es `BRONZE`, `RED` y `BLUE` para `Danio`, `Tetra` y `Corydoras`, respectivamente.

El oxígeno que consumen estos peces es `12`, `10` y `18` para `Danio`, `Tetra` y `Corydoras` respectivamente. Este valor es devuelto por el método `getOxygenConsumption`.

El método `breathe` lo dejaremos sin código, simplemente con el comentario `//TODO`. Este método lo podréis codificar fuera de la asignatura con el comportamiento que queráis si es que deseáis evolucionar el proyecto dado.

El método `toString` debe devolver el mismo `String` que la clase padre con el siguiente añadido al final: `" : Danio\n"`, `" : Tetra\n"` o `" : Corydoras\n"`, respectivamente.

(0.75 puntos: 0.25 puntos cada clase)

e) Puedes ver que ahora la clase `Item` hereda de la clase `ImageView`. La clase `ImageView` de la API de `JavaFX`, una librería que nos permite hacer interfaces gráficas. La clase `ImageView` nos permite representar mediante una imagen la clase que la hereda, en este caso `Item`. Verás que Eclipse tanto para la clase Item como para todas sus subclases te da error. Esto se debe a que no tienes añadido la librería JavaFX al proyecto. Para configurar JavaFX 11 en Eclipse, sigue los pasos que se indican en el apartado 6.2.2. de la Guía de Java. Una vez añadida la librería JavaFX, ya no deberían aparecer errores. Hablaremos sobre JavaFX más adelante, cuando tratemos las vistas de la aplicación.

f) En la clase `Item` hemos eliminado el atributo `id` , así como sus *getter* y *setter* porque ahora usamos el atributo `id` y los *getter*/*setter* heredados de la clase `ImageView`. Fíjate que en la clase `setLocation` usamos los métodos `setTranslateX` y `setTranslateY` para ubicar la imagen del ítem en el lienzo 2D que veremos más adelante. También hemos llamado al método `setImage` de `ImageView` dentro del método `setSpriteImage` para guardar la imagen (en modo gráfico) del ítem.

Por su parte, las clases `Animal` y `SubmarineToy` utilizan el método `setScaleX` de `ImageView` (lo puede usar porque ambas clases heredan de `Item` e `Item` hereda de `ImageView`... acuérdate que la herencia en Java es transitiva) para girar verticalmente la imagen.

Llegados a este punto, debes modificar el método `setStatus` de la clase `Animal` para que gestione qué imagen debe usarse en cada momento para el ítem de tipo `Animal` en función de su `status`. Si te fijas en la carpeta `images` que viene dentro del proyecto `UOCarium`, verás que las imágenes de los animales, especialmente de los peces, terminan con una coletilla, p.ej. `"_healthy"` o `"_dead"` que indican cómo está el pez representado. Así pues, si el estado de, por ejemplo, un pez de tipo `Danio` es `HEALTHY`, entonces la imagen a asignarle (y que se mostrará) será `"danio_healthy.png"`, en cambio si el `status` es `DEAD`, entonces la imagen debe ser `"danio_dead.png"`.

(0.5 puntos)

Ejercicio 2 – Controlador (`package controller`) (4 puntos)

Dentro del proyecto hay un `package` llamado `edu.uoc.uocarium.controller` que contiene aquellas clases que hacen de controladores, es decir, de intermediarias entre la vista y el modelo. Para este proyecto sólo necesitamos dos clases.

La primera de ellas, `Database`, carga los datos de la aplicación a partir de la información guardada en los diferentes ficheros de texto que hay en el carpeta `files`. Esta carga de datos se hace al abrir el programa y la información se guarda en objetos que pertenecen a las clases del modelo. Es decir, esta clase junto con los ficheros de texto de la carpeta `files` hacen de base de datos. **Esta clase te la damos codificada completamente**.

La segunda clase, `UOCariumController`, es la que interactúa con las clases del modelo una vez cargados los datos. Verás (lee el siguiente apartado) que desde las vistas `cmd` y `gui` se utiliza un objeto de la clase `UOCariumController`. La clase `UOCariumController` tiene un objeto de tipo `Database`.

Ahora se te pide abrir la clase `UOCariumController` y:

a) Codificar el método `getTanks` a partir del comentario `TODO` que encontrarás en el código de manera que este método devuelva una `List<Tank>` con los tanques ordenados alfabéticamente por el nuevo atributo `id` de los tanques.

(0.5 puntos)

b) Codificar el método `getMovableItems` de manera que devuelva una `List<Item>` que sólo incluya aquellos ítems que implementan la interfaz `Movable`.

(0.75 puntos)

c) Codificar el método `addFish` para que en el tanque que tenemos seleccionado en el momento actual (`tankSelected`), añada un pez. El tipo de pez dependerá de un valor aleatorio entre `0` (incluido) y `10` (excluido). Si el valor es menor a `3`, el pez a añadir será `Danio`; si es mayor o igual que `3` y menor que `6`

será de tipo `Tetra`; y si es igual o mayor a `6` será de tipo `Corydoras`.

En cualquier caso, los valores de `xCoord` e `yCoord` será un valor aleatorio entre `0` y `300`, el género dependerá de un valor aleatorio de tipo `boolean` (si es `true`, el valor del género será `MALE`; si es `false`, será `FEMALE`), la edad será `0` y la energía `100`.

Este método lanzará hacia arriba (i.e. `throws`) todas las excepciones que lancen los métodos a los que llame.

(1 punto)

d) Codifica el método `toggleSubmarineToy`. Este método añade al tanque seleccionado en el momento actual un objeto de tipo `SubmarineToy` si no existe uno ya en el mismo tanque. Es decir, en un tanque sólo puede haber un objeto `SubmarineToy`. Si el objeto ya existe, entonces lo elimina. De esta manera este método añade y quita el submarino del tanque.

Cuando el método añade el submarino, éste es creado con los valores `xCoord = 50`, `yCoord = 50`, `length = 100` y `height = 50`.

Este método lanzará hacia arriba (i.e. `throws`) todas las excepciones que lancen los métodos a los que llame.

(0.5 puntos)

e) Codifica el método `removeDeadItems`. Este método elimina todos los ítems muertos del tanque y devuelve una `List<Item>` con todos los ítems muertos que han sido eliminados del tanque.

(1.25 punts)

Vista consola con interacción por teclado (`package view.cmd`)

Dentro del proyecto hay el `package edu.uoc.uocarium.view` que contiene aquellas clases que hacen de interfaz para el usuario y que, por lo tanto, le permiten interactuar con el programa. En concreto hay dos *subpackages* dentro del `package view` llamados `cmd` y `gui`.

Nota: Aunque Eclipse pone estos dos *subpackages* a la misma altura que los `packages model` y `controller`, la realidad es que están dentro del paquete `view`. Esto lo puedes comprobar yendo al directorio `workspace` de Eclipse y mirando la carpeta `src` del proyecto.

El primero de estos dos paquetes, `cmd`, crea una interfaz basada en consola en la que el cliente/usuario interactúa mediante teclado (o línea de comandos). Por su parte, el segundo paquete (`gui`) crea una interfaz gráfica que permite al cliente interactuar con el ratón.

Ahora mismo nos vamos a centrar en el primer paquete, `edu.uoc.uocarium.view.cmd`. Este paquete sólo incluye una clase llamada `CmdApp` que ya te la damos codificada (i.e. no tienes que hacer nada con ella, pero te recomendamos que le eches un vistazo). Si has resuelto los apartados anteriores de esta práctica, incluida la configuración de JavaFX (se necesita porque `Item` hereda de `ImageView`), no deberían aparecerte errores. La clase `CmdApp` se encarga de efectuar las siguientes operaciones:

1. Crea un objeto de la clase `UOCariumController` para controlar el acuario. La gestión es incompleta ya que desarrollarla supone muchas horas y no es nuestra intención hacer un acuario virtual entero. No obstante, es funcional.
2. Ejecuta el flujo del programa.
3. Captura los *inputs* de teclado, procesa las peticiones delegando su ejecución en la clase `UOCariumController` y, finalmente, muestra por pantalla la información solicitada.

Nota: Configura Eclipse para ejecutar el `main` de la clase `CmdApp` a través de este IDE (en principio ya te lo damos hecho). En `Arguments` → `VM arguments` debes escribir lo siguiente:

```
--module-path "C:\Program Files\Java\javafx-sdk-11.0.2\lib" --add-modules javafx.controls,javafx.fxml
```

Siendo el texto en rojo la ruta donde tengas instalado JavaFX. Como verás en el siguiente ejercicio, hay otro `main` en la clase `UOCariumApp`. Gracias a `CmdApp` puedes comprobar que gran parte del modelo y del controlador funcionan correctamente.

Ejercicio 3 – Vista gráfica con interacción por mouse (package `view.gui`) (1.25 puntos)

Como ya hemos comentado anteriormente, tenemos dos interfaces (no de orientación a objetos) con las que un usuario puede interactuar con el programa. Obviamente, la vista de consola no nos servirá, ¡¡imagínate la cara de los usuarios cuando tengan que interactuar con un programa similar a los de los años 80!!

Es por eso que debemos crear una interfaz más visual y usable. Esto lo haremos con las clases que encontrarás en el paquete `edu.uoc.uocarium.view.gui`. Muchas de estas clases ya te las damos codificadas. Recuerda configurar correctamente el programa con `Run→Run Configurations…` para que ejecute la vista gráfica (`UOCariumApp`) y no la de línea de comandos.

Las interfaces gráficas (en inglés, *Graphical User Interface*, GUI) son un elemento básico hoy en día. Java tiene tres bibliotecas para hacer interfaces gráficas: AWT, Swing y JavaFX. Resumiendo mucho, se podría decir que Swing es más avanzado que AWT y que JavaFX ha llegado para ser el sustituto de Swing. Hasta hace relativamente poco, Swing era la librería predominante, pero JavaFX está comenzando a ganarle terreno. No obstante, en un programa puedes combinar componentes (i.e. botón, campo de texto, desplegable, etc.) de ambas librerías, aunque no es muy recomendable. Oracle no ha dado por obsoleto a Swing, pero parece ser que no le va a incorporar mejoras en el futuro. Es por esto que en esta práctica nos centraremos en JavaFX. Asimismo, con JDK 11 Oracle ha decidido quitar del núcleo del JDK la librería JavaFX para que ésta sea independiente y tenga su propio ritmo de desarrollo, liberando así su evolución. Esto, evidentemente, conlleva algunos cambios, especialmente a la hora de configurar el entorno de trabajo (i.e. Eclipse).

Swing y JavaFX comparten muchas cosas, como es el uso de componentes (también conocidos como *widgets*), pero tienen cosas totalmente diferentes, como por ejemplo la forma de trabajar, además de sus capacidades. Si quieres saber más, lee esta comparativa-resumen: http://www.dummies.com/programming/java/10-differences-between-javafx-and-swing/. También puedes saber más sobre las ventajas de JavaFX en este vídeo (en inglés): https://www.youtube.com/watch?v=9YrmON6nlEw

Llegados a este punto, si la clase `Item` no heredara de `ImageView`, ahora deberíamos configurar Eclipse para que fuera capaz de interpretar código y clases de JavaFX, pero esto ya lo has hecho anteriormente.

Modelo en JavaFX

Una de las ventajas de JavaFX es que usa el patrón MVC para crear los programas, lo cual, como hemos visto, es muy interesante. En esta práctica no trabajaremos la parte "modelo" al estilo JavaFX, es decir, para sacarle partido a todo el potencial de JavaFX. Esto te lo dejamos a ti, por si quieres indagar en JavaFX. Si estás interesado/a, busca los conceptos `properties` y `binding` de JavaFX. Nosotros nos limitaremos a usar el modelo de manera clásica/tradicional, es decir, con las clases que hemos codificado en el paquete `edu.uoc.uocarium.model` siguiendo el paradigma POO.

Vistas en JavaFX: parte visual (0.5 puntos)

Lee el apartado 6.2.4 de la Guía de Java para entender un poco más qué es una vista en JavaFX así como para aprender a instalar SceneBuilder y configurarlo para usarlo dentro de Eclipse.

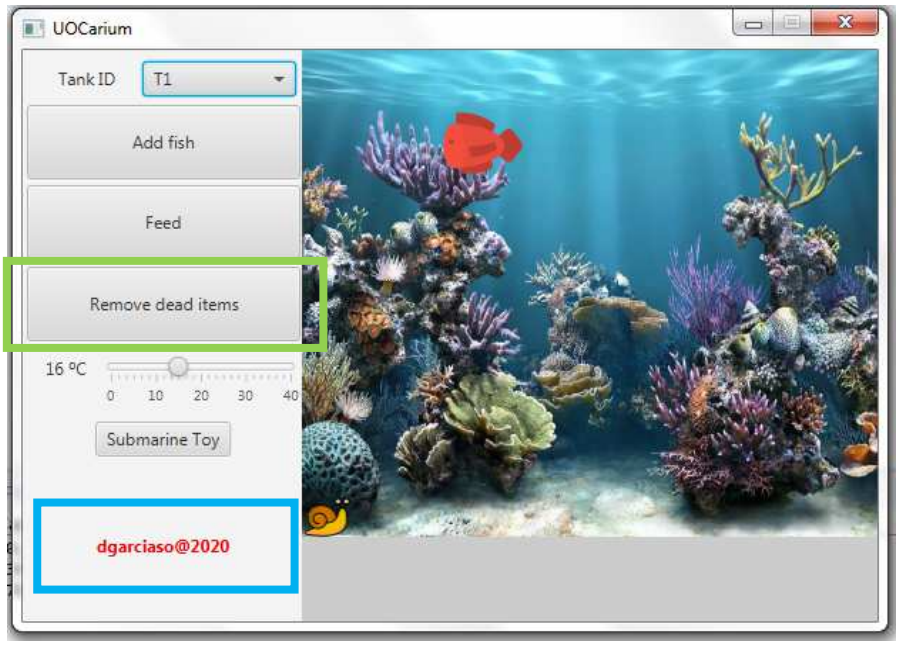
Con el proyecto `UOCarium` ya te damos las vistas/pantallas del programa hechas. En verdad el programa sólo tiene una pantalla/vista llamada `MainView.fxml`. No queremos que pierdas demasiado tiempo creando interfaces (no estás en una asignatura de diseño gráfico y usabilidad). Decimos "demasiado tiempo" porque a continuación se te pide añadir en la pantalla `MainView.fxml` que hay en el paquete `edu.uoc.uocarium.view.gui`:

a) Añadir un texto con tu login UOC seguido de `"@2020"`. Este texto debe estar en negrita y debe ser de color rojo (ver recuadro azul en la siguiente imagen).

(0.25 puntos)

b) Un botón con el texto `"Remove dead items"` (ver recuadro verde en la siguiente imagen).

(0.25 puntos)



Nota: Las vistas, es decir, cada FXML puede ser personalizado mediante un fichero CSS (o incluso varios FXML pueden compartir un mismo fichero CSS). Un fichero CSS (*Cascade Style Sheet*) es un documento en el que se especifican diferentes estilos, desde generales hasta concretos (p.ej. para un botón determinado). Estos ficheros son esenciales cuando se programan webs, ya que facilitan la escalabilidad, el mantenimiento y la gestión de posibles cambios. JavaFX usa CSS aunque utiliza sus propias propiedades, a pesar de que muchas son similares a las propiedades web, por lo que es sencillo usarlas si se sabe CSS web. Nosotros esto no lo trataremos aquí.

Si en `MainView.fxml` seleccionas el `AnchorPane` que hay (lateral izquierda, `Document→Hierarchy`) y luego miras el panel `Properties` que hay a la derecha del programa (ver figura), dentro encontrarás un apartado llamado `JavaFX CSS`. Este apartado permite indicar estilos directamente (apartado `Style`), asignarle una clase de un fichero CSS y añadir un fichero CSS. Justo en el apartado `Stylesheets`, hemos puesto `style.css` que es el nombre del fichero CSS de la aplicación. En el caso del color y el tamaño de un texto (i.e. `label`) o un botón (i.e. `button`), SceneBuilder ya trae un apartado para ellos dentro de `Properties` sin necesidad de usar propiedades CSS (¡¡encuéntralo!!). Para poner el texto en negrita sí deberás usar una propiedad CSS.

Vistas en JavaFX: parte interactiva (0.75 puntos)

Hasta ahora sólo hemos visto cómo crear la parte visual de una interfaz/vista/pantalla pero, ¿cómo le damos vida? ¿Cómo interactúa el usuario con ella? Pues bien, esto se hace mediante eventos (tanto en JavaFX como en Swing y AWT). Esto implica una nueva forma de programar que es denominada programación orientada a eventos (POE). En la POE, el flujo de ejecución del programa viene definido por las acciones/eventos (p.ej. clic, doble clic, presionar una tecla, etc.) realizados por un agente externo al programa, p.ej. un usuario, otro programa, etc. La POE no es incompatible con la POO. De hecho, con Java, seguiremos trabajando con objetos que son capaces de responder a eventos. Así pues, POO y POE se complementan. Por ejemplo, tendremos un objeto de tipo `Button` que estará escuchando (con lo que se denomina un *Listener*) que ocurra un evento. Cuando ocurra un evento concreto como puede ser hacer clic en el botón, si tiene codificado un método asociado a "hacer clic", dicho método se ejecutará al producirse el evento. Es decir el *Listener* invocará al método asociado.

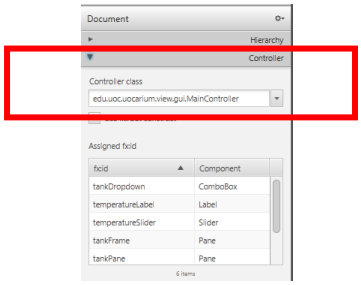
Como ya hemos dicho, una pantalla/interfaz/vista tendrá, por un lado, su fichero que define la parte visual (i.e. un fichero FXML) y, por otro, un fichero Java que será el encargado de gestionar los eventos producidos sobre la vista (i.e. fichero FXML) y seguir la lógica del programa según estos eventos. Como habrás visto en el proyecto Eclipse, dentro del paquete `view.gui` hay unos ficheros Java llamados igual que los FXML pero cuyos nombres acaban con la coletilla `Controller`. Ésta es una norma no escrita que se utiliza para denominar a estos ficheros Java que se encargan de la parte interactiva de los ficheros FXML. Así sabemos que el fichero Java encargado de la parte interactiva de un fichero FXML se llama igual que el fichero FXML pero con la coletilla `Controller`. A partir de ahora llamaremos controladores a estos ficheros.

Nota: SceneBuilder sólo detecta/encuentra las clases (i.e. ficheros Java) que están en el mismo paquete que el fichero FXML que se está editando.

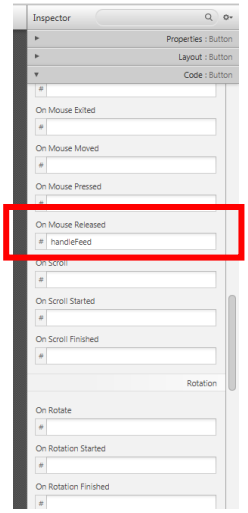
Como ya habrás podido intuir, es esencial vincular un controlador/parte interactiva (fichero Java) a una vista (fichero FXML). Esto se puede hacer de dos formas:

(1) Editando directamente el código XML del fichero FXML: abre el fichero `MainView.fxml` con Eclipse. Si indagas en el código, verás que en la raíz del tag/etiqueta `AnchorPane` (primera etiqueta del fichero) hay al final un campo/atributo llamado `fx:controller` que tiene como valor la clase controladora `MainController` (incluyendo el paquete en el que está ubicado).

(2) Usando SceneBuilder: abre el fichero `MainView.fxml` con SceneBuilder. Verás que hay una opción llamada `Controller` dentro de la opción `Document` del menú izquierdo (ver figura). En ella hay un campo denominado `Controller class` en el que puedes asignarle la clase controladora. SceneBuilder detectará todas aquellas clases que estén en el mismo paquete que el fichero FXML que se está editando. Para cada FXML deberás escoger del desplegable la opción (i.e. controlador) que corresponda.



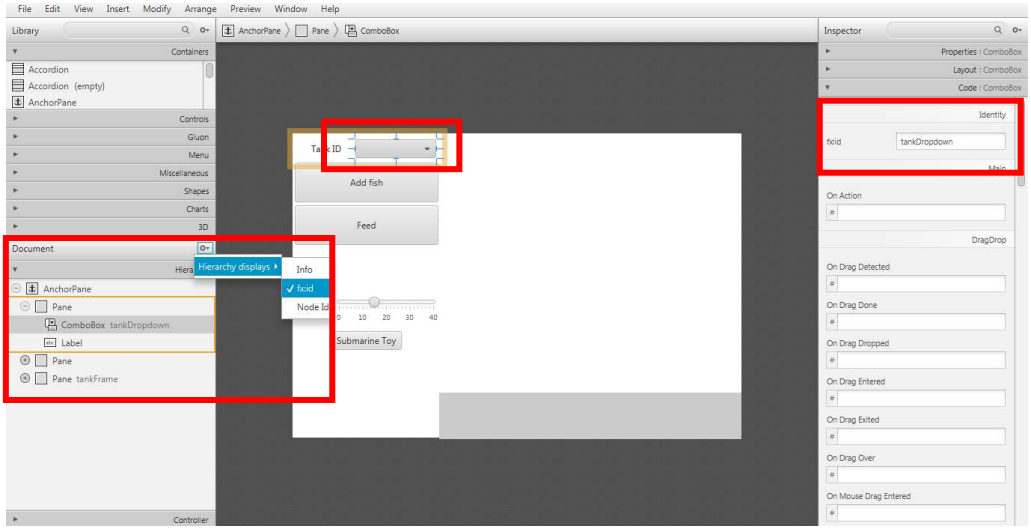
Ahora abre la clase `MainController` que es la clase que muestra la información de la pantalla. En ella verás un método al que hemos llamado `handleFeed`. Este método será el encargado de alimentar a los ítems que haya en el tanque actual. Hemos hecho que este método se llame cuando el usuario suelte después de hacer click (*on mouse released*) sobre el `Button` con texto `Feed`. Esta asignación `Button-handleFeed` la hemos hecho con SceneBuilder. Concretamente, hemos abierto el FXML `MainView.fxml` con SceneBuilder, hemos seleccionado el `Button` y en el apartado `Code` del lateral derecho hemos buscado la opción `On Mouse Released` (ver figura). Allí hemos escrito/seleccionado el nombre de método a ejecutar, en este caso, `handleFeed`. Esta asignación también la puedes ver reflejada, obviamente, en el código FXML del fichero `MainView.fxml`(puedes verlo con Eclipse).



Nota: Es importante que te des cuenta de que en `MainController`, la firma del método `handleFeed` va precedida de la anotación `@FXML`. Esto permite que las vistas realizadas con SceneBuilder tengan acceso a los atributos y métodos del controlador, incluso aunque sean privados. Si son públicos, entonces no hace falta anteponer la anotación `@FXML`, aunque es recomendable.

Si ahora abres el controlador `MainController` verás que tiene unos atributos privados precedidos por la anotación `@FXML`. Esto permite, como ya hemos comentado, que la vista pueda acceder (y usar) a estos atributos del controlador. Para que lo veas más claro, abre con SceneBuilder el fichero `MainView.fxml`. Haz clic en el desplegable gris que hay al lado del texto "Tank ID" del menú izquierdo de nuestra interfaz que es de tipo `ComboBox`. También puedes seleccionar este componente mediante la opción `Hierarchy` del menú izquierdo de SceneBuilder, concretamente es el `Combobox` que hay dentro del primer `Pane` (ver figura). En este caso hemos habilitado que SceneBuilder nos muestre los identificadores de los componentes (si tienen) dentro de este apartado de `Hierarchy`. Si tienes seleccionado el componente, en el menú derecho, en la opción `Code`, encontrarás un campo llamado `fx:id`. En él verás que tiene como identificador el nombre `tankDropdown`, es decir, el mismo nombre que hemos puesto a uno de los atributos del controlador. De esta forma, el controlador puede acceder a este componente para modificarlo o consultarlo, y la vista a la información del atributo de la clase de tipo controladora. O dicho de otro modo, el atributo de la clase controladora hace referencia al componente de la vista porque tienen el mismo nombre y porque en

la clase controladora hemos puesto `@FXML` delante del atributo. Esta vinculación también la puedes ver en el código FXML, si abres el fichero `MainView.fxml` con Eclipse.



Después de haber leído los conceptos básicos sobre qué es y cómo funciona JavaFX (evidentemente hay muchas más cosas para aprender sobre esta API), se te pide:

c) Añadir, al botón `"Remove dead items"` que has añadido en la pantalla `MainView.fxml` según las indicaciones del apartado anterior, la funcionalidad de que al ser cliqueado elimine los ítems muertos en el tanque actual (el que se está mostrando).

Nota: Para codificar esta funcionalidad deberás seguir el patrón MVC. Concretamente tendrás que crear un método en la parte interactiva (`MainController.java`) para que se ejecute cada vez que se pulse en el botón `"Remove dead items"` de la vista `MainView.fxml` y que este método a su vez se comunique con el controlador de la aplicación (`UOCariumController`) para ejecutar el método `removeDeadItems` quien realmente hará toda la tarea (i.e. gestionar la lógica del programa).

Inspecciona el código que te damos para saber cómo hacer este apartado.

(0.75 puntos)

Ejercicio 4 – Tests unitarios con JUnit (1.75 puntos)

Una vez hemos hecho el desarrollo del programa, se pasa a la etapa de *testing*/testeo/pruebas. Pero como ya hemos dicho anteriormente, durante el desarrollo también se hacen pruebas locales (y no tan locales).

Tipos de test

Los tests los podemos ver desde dos perspectivas: (1) funcionalidad del test, y (2) quién ejecuta el test.

Funcionalidad del test

En esta categoría veremos dos tipos: los tests funcionales y los no-funcionales.

- **Test funcional**: en este tipo de pruebas se comprueba que el software cumple las funciones por las que se ha construido. Por ejemplo, en una calculadora con la operación de sumar, se debe comprobar que la operación sumar se lleva a cabo correctamente.
- **Test no-funcional**: este tipo de test comprueba otros elementos importantes en un software, p.ej. seguridad, escalabilidad, volumen de datos a manejar o a cargar, niveles mínimos aceptables de rendimiento, etc. La comprobación de algunos de estos elementos dan nombre al test, como por ejemplo:
    - **Test de estrés**: busca forzar un fallo del software a través del consumo excesivo de sus recursos (p.ej. espacio en disco, memoria, etc.).
    - **Test de carga**: mira la concurrencia máxima, es decir, la capacidad máxima que tiene el software para atender a un conjunto de usuarios de manera simultánea.
    - **Test de rendimiento (*performance*)**: comprueba que el tiempo de respuesta del software a una petición sea aceptable.

Quién ejecuta el test

En esta categoría podemos clasificar los tests en si los ejecuta el cliente o el equipo de desarrollador. En el primer caso, al test se le llama test de aceptación o de cliente, mientras que en el segundo caso, existen tres tipos de tests: (1) test unitario, (2) de integración y (3) de sistemas.

Test realizado por el cliente (conocido como test de aceptación/cliente)

En una metodología en cascada, al cliente se le ve al principio –para saber los requisitos– y al final, es decir, para entregarle el producto. En otras metodologías, como las ágiles (donde se incluye SCRUM), al cliente se le muestra el estado del producto cada cierto tiempo, p.ej. cada 15 días o cada mes. Cuando el cliente define el test (que lo puede hacer antes de que el equipo desarrollador escriba una sola línea de código), éste se usa para que el cliente valide el software. Por ello a dicho test se le llama test de aceptación o de cliente. Así pues, quien ejecuta el test es el cliente.

El test de aceptación/cliente es un test que permite comprobar que el software cumple con un requisito de negocio. Así pues, un test de aceptación describe, en lenguaje natural, un escenario/ejemplo que debe cumplir el software desde la perspectiva del cliente. Estos ejemplos/tests son consensuados con el cliente. Esto implica que al final del proyecto se le presenta al cliente el resultado de dichas pruebas que, si son favorables, obligan al cliente a dar por válido el software y, por ende, por cerrado el proyecto. Ejemplos de tests de aceptación son:

- El producto 045 con precio 35€ al aplicarle el IVA del 21% su precio final es de 42,35€.
- Cuando la facturación llegue a 21.000€ se debe enviar un e-mail al encargado de facturas.
- La web debe ser compatible con tablets.
- La web realizada debe cargarse en menos de 1 segundo.
- La web debe permitir 1.000.000 de usuarios concurrentes.

Como se puede ver, un test de aceptación puede ser un test funcional (un ejemplo de este tipo está en verde) o no-funcional (un ejemplo de test de carga está en rojo, otros ejemplos serían: seguridad, escalabilidad, volumen de datos a manejar, niveles mínimos aceptables de rendimiento, etc.).

Test realizado por el equipo desarrollador

Durante el desarrollo los programadores hacen, principalmente, tres tipos de tests:

- **Tests unitarios (*unit test*)**: son las pruebas de más bajo nivel en la programación. Con ellas se comprueba el funcionamiento de las unidades lógicas independientes, normalmente los métodos. Son pruebas muy rápidas de llevar a cabo y son las que los desarrolladores están constantemente pasando para verificar que su software funciona adecuadamente.

- **Tests de integración**: con estas pruebas se llevan a cabo comprobaciones de varias unidades de software diferentes. En este tipo de pruebas se hacen comprobaciones de que diferentes unidades lógicas funcionan adecuadamente entre sí. Normalmente se llevan a cabo comprobaciones del paso de mensajes entre estas unidades lógicas. Por ejemplo, un método que llama a otro; un objeto que interactúa con otro, etc.

- **Test de sistema**: una vez que se han comprobado las unidades lógicas de software mediante tests unitarios y de integración, se comprueba la aplicación completa en un entorno de desarrollo similar al de producción.

Test unitario con JUnit 5

En esta Práctica nos centraremos en los tests unitarios. Puedes leer sobre ellos en el apartado 6.3.1 de la Guía de Java.

Para hacer los tests unitarios usaremos la librería JUnit. JUnit viene con Eclipse, pero se debe configurar. Así pues, en primer lugar, debes configurar JUnit en tu entorno de trabajo (es decir, Eclipse). Para ello, sigue las indicaciones dadas en el apartado 6.3.2 de la Guía de Java. Una vez hayas configurado JUnit y hayas tenido un primer contacto

siguiendo las indicaciones del apartado 6.3.3 de la Guía de Java, se te pide usar la librería JUnit que viene integrada en Eclipse para testear los siguientes métodos de la clase `Keeper`:

d) Método `setId`. Recuerda que este método lanza una excepción con un mensaje de error si el `id` no tiene `5` caracteres, no empieza por `G` o si el valor del `id` es `null`. Asimismo ten presente que el método `setId` también se llama en el constructor de clase `Keeper`.

(1 punto)

e) Método `addTank`. Debe ser capaz de añadir hasta 5 tanques y lanzar una excepción si nos pasamos de esta cantidad.

(0.75 puntos)

Para hacer los dos apartados anteriores deberás crear un fichero `KeeperTest.java` que contenga los métodos de testeo `testSetId` y `testAddTank`.

Entregando el programa (es opcional y no se debe entregar)

Si estás leyendo esto, es que ya has terminado la Práctica. Quizás te estés preguntando: *¿Cómo hago para pasarle el programa a alguien que no tenga ni Eclipse ni el JDK instalado?* Buena pregunta. Debes crear un archivo ejecutable, concretamente, un JAR (*Java ARchive*). Un fichero `.jar` es un tipo de fichero –en verdad, un fichero comprimido en formato `.zip` con la extensión cambiada– que permite, entre otras cosas, ejecutar aplicaciones escritas en el lenguaje Java. Gracias a los ficheros `.jar`, cualquier persona que tenga instalado el JRE (*Java Runtime Environment*), lo podrá ejecutar como si de un fichero ejecutable se tratase. Normalmente, los PCs cuentan con el JRE instalado.

Para crear un fichero `.jar` con Eclipse debes:

1. Ir al menú superior de Eclipse y escoger ***File→Export…***

2. En la ventana que te aparecerá debes escoger ***Java→Runnable JAR file*** y hacer clic en el botón ***Next***.

3. En la siguiente ventana escoge la configuración de ejecución en el campo ***Launch configuration***, es decir, cuál es el `main`, qué argumentos se le pasa, etc. Esta configuración es una de las que has creado en *Run Configurations…* Luego, en ***Export destination***, escoge un nombre y la ubicación donde quieres que te guarde el `.jar` generado. Finalmente, escoge la primera opción que aparece en ***Library handling***. Para saber la diferencia entre las tres opciones lee: http://blog.notfaqs.com/2013/04/java-what-is-difference-between.html. Si tienes clases con *warnings*, puede que te avise en este momento. Omítelos y continúa con el proceso.

4. Si generas el `.jar` del proyecto `UOCarium`, **algo que no te pedimos con esta práctica**, acuérdate de poner al lado del `.jar` las carpetas auxiliares (i.e. `files`, `images`, …) con sus correspondientes ficheros.

5. Para ejecutar el `.jar` generado, haz simplemente doble clic en él. Si has puesto como `main` el que está ubicado en `UOCariumApp`, entonces se te abrirá la interfaz gráfica. Si por el contrario has escogido el `main` que hay en `CmdApp`, al tratarse de una interfaz de

línea de comandos, no servirá hacer doble clic. En este caso debes ejecutarlo por línea de comandos de la siguiente manera:

```
java –jar nombreFichero.jar
```

donde `nombreFichero` es el nombre que le has puesto al fichero `.jar`.

Nota: En un programa en el que hay imágenes incrustadas, es posible que si abres la aplicación gráfica no te aparezcan las imágenes incrustadas con SceneBuilder. Esto es debido al *path* que ha puesto SceneBuilder en el atributo `url` del *tag* `Image` en el FXML a la hora de incrustar las imágenes. Para que se vean, deberías cambiar la dirección que hay en los `url` de los *tag* `Image` de la siguiente manera:

* Antes: `@../../../../../../images/image1.jpg`

* Ahora: `file:./images/image1.jpg`

De esta manera, te irá tanto al probar el programa en Eclipse como al generar el *runnable jar*. Una buena práctica es cambiar las direcciones (i.e. *paths*) a medida que se van creando los FXML. Como vemos, SceneBuilder no es perfecto y su uso introduce un pequeño problema.

Recursos

Esta práctica intenta trabajar todos los conceptos estudiados en la asignatura y algunos nuevos, dando una especial importancia a la codificación. Con esta práctica se proporcionan diferentes guías para los diferentes apartados. Recuerda que también dispones en el aula de una nueva guía sobre Java denominada **"Guía de Java (preliminar)"**.

Es muy normal que surjan dudas cuando se programa, aunque llevemos muchos años programando con un lenguaje y un paradigma concretos. Es por eso que hay que tener muy asimilada la competencia de buscar información.

Hoy en día, el mejor lugar donde encontrar información y de manera rápida es Internet. En el caso de Java, tienes toda la documentación de su API (i.e. librerías con clases ya hechas) en la siguiente web: https://docs.oracle.com/en/java/javase/13/docs/api/index.html.

También puedes utilizar foros como *stackoverflow*, tanto la versión inglesa como la española:

- Inglés: http://stackoverflow.com/questions/tagged/java
- Español: http://es.stackoverflow.com/questions/tagged/java

O utilizando directamente el buscador Google: www.google.com

Si quieres ampliar conocimientos, siempre puedes comprar libros sobre POO y Java en alguna librería o pedirlos prestados en cualquier biblioteca (incluyendo la de la UOC, http://biblioteca.uoc.edu).

Si a pesar de buscar información, no solucionas tus problemas, puedes hacer preguntas teóricas, pedir aclaraciones de los enunciados o preguntar sobre aspectos técnicos (p.ej. Java) en el foro del aula. **No puedes usar los foros ni ningún otro medio para pedir a otro compañero la solución de lo que se pide en los enunciados. Tanto la persona que pide como quien proporciona la solución, serán penalizadas siguiendo la normativa académica de la UOC.**

Criterios de valoración

El reparto de los puntos de esta práctica es el siguiente:

- Ejercicio 1: 3 puntos
- Ejercicio 2: 4 puntos
- Ejercicio 3: 1.25 puntos
- Ejercicio 4: 1.75 puntos

El detalle del reparto de los puntos en cada parte está en el enunciado.

Formato y fecha de entrega

Se tiene que entregar un fichero *.zip, cuyo nombre tiene que seguir este patrón: `loginUOC_PRAC4.zip`. Por ejemplo: *dgarciaso_PRAC4.zip*.

Este fichero comprimido tiene que incluir el proyecto Eclipse llamado `UOCarium` finalizado según las especificaciones del enunciado y del diagrama de clases facilitado. Encontrarás este proyecto en tu *workspace* de Eclipse. **No hay que entregar el fichero *.jar que permite a un usuario final ejecutar la aplicación**.

El último día para entregar esta práctica es el **14 de junio de 2020 a las 23:59**. Cualquier práctica entregada más tarde será considerada como no presentada.