Práctica 2: Proyecto, planificación y versionado

1. Planteamiento

En esta práctica vamos a utilizar una serie de herramientas que nos serán útiles para el trabajo en los proyectos a desarrollar en la asignatura. A modo de ejemplo, utilizaremos un microproyecto (el juego del coche que iniciamos en la práctica 0), para hacer en una semana una simulación de lo que nos tocará hacer en el proyecto en tres meses.

Descarga el fichero asociado. En él encontrarás básicamente lo que planteábamos en la práctica 0, con algunos cambios leves y un cambio de diseño que añade la clase *MundoJuego*, que recoge todos los cálculos que tienen que ver con los movimientos y choques (las *físicas*) para separarlos del interfaz de usuario, que se mantiene en la clase *VentanaJuego*.

2. Desarrollo

Paso 1. Vamos a planificar nuestro proyecto. Cada proyecto **parte** de un punto diferente: en nuestro microcaso, partimos de lo que ya tenemos como solución de la práctica 0 y queremos llegar a realizar todos los pasos definidos en esta práctica.

Utiliza la herramienta en la nube de www.gantter.com para crear una planificación de esta práctica. Deberás:

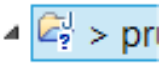
- Registrarte en gantter con tu email (o con google+, linkedin o facebook).
- Crear un nuevo proyecto para esta práctica. Ponle de nombre "práctica 2 - " y tu nombre. Guardarlo en la "nube de gantter".
- Crear una serie de tareas asociadas a los pasos de esta práctica, con los días/horas de duración y de inicio que preveas que vas a tardar.
- Añadir las columnas (menu *Ver*) *Esfuerzo* y *%Completado* a la visualización.
- Compartir con el profesor (*Proyecto* | *Compartir* | con andoni.eguiluz@deusto.es) como viewer.
- Compartir el proyecto también con algún compañero con permiso de Editor (lo tendréis que hacer en el proyecto entre todos los miembros de cada equipo).

Observa como en el menú Acciones puedes mover arriba o abajo las tareas, y también moverlas a la derecha para incluir algunas subtareas lógicamente en otras.

Observa la importancia de la precedencia entre tareas (igual que en esta práctica unas tareas necesitan que otras se hayan hecho antes).

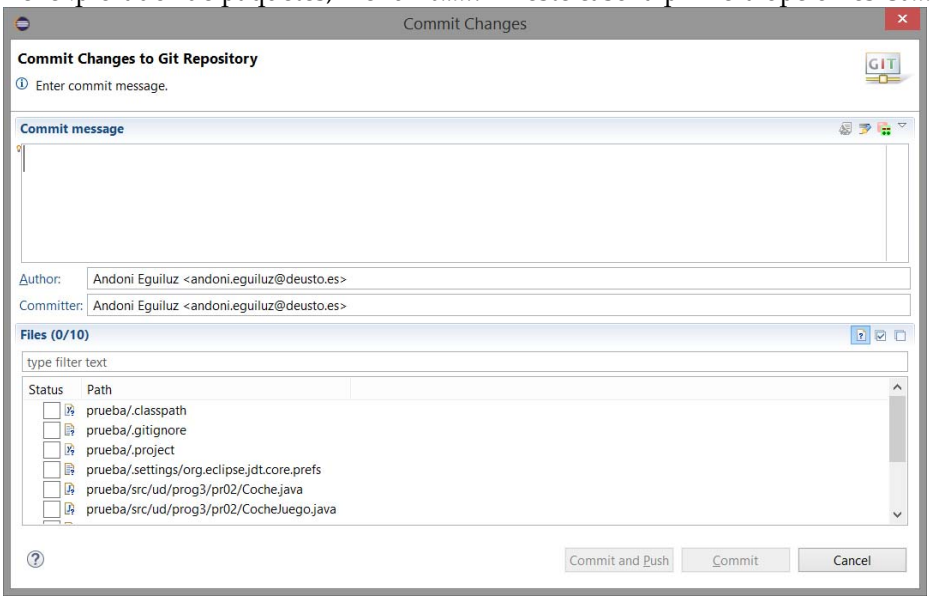
Paso 2. Vamos a empezar a utilizar un repositorio de código con gestión de versiones, utilizando la herramienta GIT, una de las más utilizadas entre los programadores, sobre el servicio en línea de repositorios github. Para ello tendrás que realizar las siguientes acciones:

- Si no la tienes, crea una cuenta en http://github.com. Te va a valer para toda la asignatura (y más allá), así que utiliza preferiblemente tu email de opendeusto y tus datos reales.
- Entra con tu cuenta y crea un proyecto en github, por ejemplo llamado *ud-prog3-pr02* (como el paquete de la práctica pero con guiones en lugar de puntos)
- Te aparecerá una referencia única de github (algo así como `https://github.com/xxx/ud-prog3-pr02.git`). Cópiala, te servirá después.
- Crea y configura tu proyecto en eclipse para esta práctica, con los ficheros fuentes que se suministran. Comprueba que la práctica se ejecuta y funciona bien.
- Vamos ahora a crear un *repositorio git* para este proyecto. Esto es, una **copia** del proyecto en otra carpeta, que irá guardando información progresiva de cambios en los ficheros fuente, con la que podremos recuperar en cualquier momento versiones anteriores de nuestros cambios. Para ello:
  - Botón derecho en el proyecto (en el explorador de paquetes de la izquierda) - *Team* | *Share project*
  - Tipo de repositorio *Git*
  - Crear repositorio (botón *Create*) en alguna otra carpeta con permisos de escritura del disco, diferente al workspace de eclipse (ahí es donde se creará la copia de ficheros de git). Finalizar el asistente.
    Ahora este proyecto está enlazado con el repositorio GIT. Verás un símbolo con un interrogante en el proyecto (icono de la derecha). Esto significa que aunque está enlazado, todavía no hemos hecho una copia de los ficheros al repositorio (*commit*).



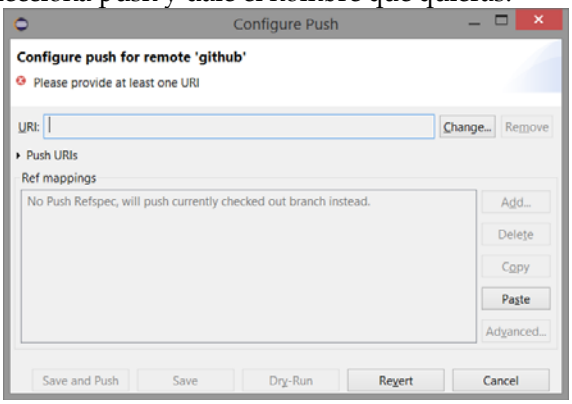
- Puedes ver el repositorio y su estado (vacío) sacando una ventana de eclipse específica para navegar por los repositorios git: *Window* | *Show View* | *Other* | *Git* | *Git Repositories*. Ahí puedes ver abriendo el árbol que nuestros fuentes (*Working directory* | *nombre proy* | *src*) aún no están.

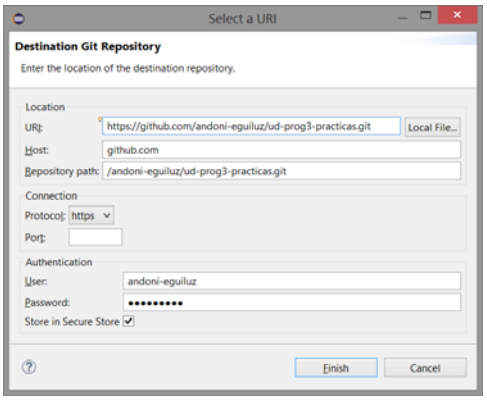
- Hagamos nuestro primer *commit*. Todas las opciones de Git están en el menú de contexto (botón derecho) del proyecto en el explorador de paquetes, menú *Team*. En este caso la primera opción es *Commit...*



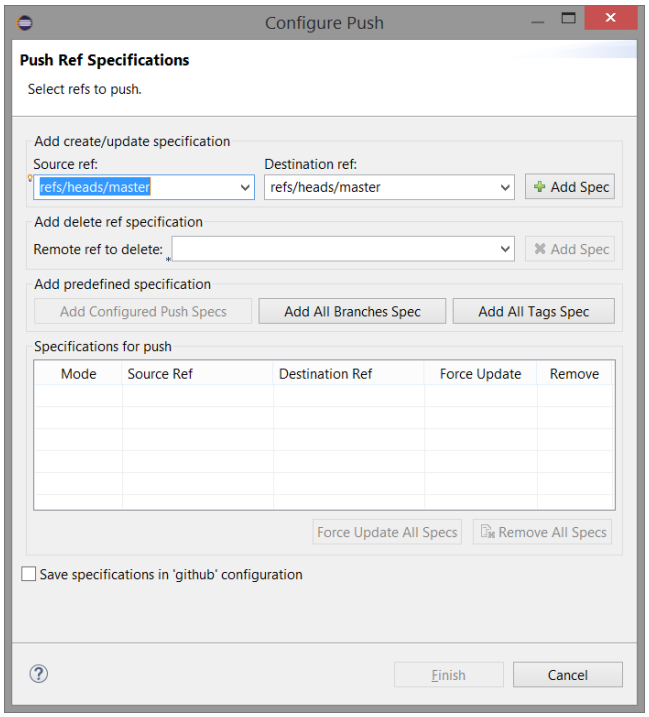
  - El mensaje (obligatorio) describe los cambios que se han hecho desde la última vez (como es el primero, ahora puedes poner "primera versión" o algo así). Autor y *commiter* deberían ser tus datos, y luego puedes seleccionar los ficheros a sincronizar (ahora deberían ser todos - botón de selección de todos a la derecha: ). Pulsa el botón *Commit*.
  - Con esto GIT ha llevado todos los ficheros al repositorio. Puedes comprobarlo en la ventana de repositorio GIT de Eclipse, o abriendo un explorador de archivos y yendo allí.

- Esto es útil, pero lo es mucho más si hacemos una sincronización remota con un directorio en otro lugar: un servidor GIT. Para eso recuperamos nuestro trabajo de github.
  - En la ventana *Git Repositories* puedes, dentro del repositorio que has creado, observar entre otras cosas un icono amarillo de disco llamado *Remotes*. Este es el enlace con un repositorio remoto, que vamos a enlazar con GitHub. Haz botón derecho - *Create Remote*
  - Selecciona push y dale el nombre que quieras.





  - En la ventana que aparece, haces *Change…* y por fin puedes configurar el camino del directorio remoto de github. Pégalo en *URI*. Deberás además meter tu usuario y contraseña de autenticación.
  - Pulsa *Finish.*
  - En la ventana de push, pulsa el botón *Advanced*:



- Esto te permite seleccionar la referencia fuerte *refs/heads/master* + botón *Add Spec* que indica que vamos a hacer el push de la "rama maestra". Activa el check "*Force Update*".
- Pulsa *Finish* y por fin *Save and Push*. Esta operación llamada *Push* en Git, copia el repositorio local en el repositorio remoto.

Ya tienes tu repositorio Git en local y en remoto listos para trabajar. Si vas al navegador y entras en tu repositorio github, deberías ver todos los ficheros.

Paso 3. Mejora de interacción de teclado.

Vamos a plantear ahora algunas mejoras de este microproyecto, para meterlas en la planificación y actualizar las versiones de nuestro repositorio.

En primer lugar, vamos a cambiar la interacción por teclado del juego. Observa que al pulsar cualquier tecla de cursor, la respuesta del coche es buena si es sólo una pulsación. Pero si mantenemos la tecla pulsada (lo verás mejor si lo pruebas con cualquier giro lateral), observa que se realiza un solo giro, pasan unas décimas de segundo, y luego se realiza un giro ya continuo. Si intentamos pulsar a la vez dos teclas de cursor de forma continua (por ejemplo acelerando y girando a la vez), no funcionará adecuadamente.

Esto se debe a que hemos gestionado solo el evento *KeyPressed*, que se lanza cada vez que una tecla se pulsa, y luego de acuerdo al sistema operativo después de un retardo (*delay*) se empiezan a generar pulsaciones repetidas (verás que es lo mismo que si en cualquier editor de texto pulsas una letra).

Vamos a cambiarlo gestionando las pulsaciones y sueltas de tecla directamente, y memorizando todas las teclas pulsadas (para tener presentes pulsaciones múltiples). Define un nuevo atributo array de booleanos. Como vamos a controlar solo cuatro teclas (los cursores), valdrá con un array de 4 posiciones. Cada vez que ocurra el evento de *pressed*, pon el booleano correspondiente a true, y con el evento de *released*, ponlo a false. (quitando todo el código de control de velocidad y giro que había antes).

Con esto, quien controla el cambio de la velocidad y el giro ya no puede ser el gestor de eventos de teclado (sólo se lanza una vez en cada pulsación). Lo que vamos a hacer es aprovechar el mismo hilo de visualización y cada 40 milésimas, igual que se refresca el visual, revisamos las teclas pulsadas, y actuamos sobre las que en ese momento estén pulsadas -array a true-, que pueden ser ninguna o varias.

Comprueba que ahora el coche responde al teclado más finamente, y que además puedes acelerar o frenar y girar a la vez. Este tipo de control se utiliza mucho en juegos y en otros sistemas con control interactivo de teclado en tiempo real. (Ojo, aunque en este caso hemos aprovechado el hilo de visualización para la gestión de teclas, normalmente se suelen hacer distintos - los tiempos de refresco no tienen por qué ser los mismos)

Paso 3b. Aprovechemos estos cambios para ver una utilidad clave de los sistemas de versionado. Poder comparar código y cambios, o diferencias entre dos versiones distintas de la misma clase (un caso especialmente necesario cuando hay varios programadores en los mismos proyectos). Ve a la clase *VentanaJuego.java* que acabas de modificar en el explorador de paquetes de eclipse, y haz botón derecho + *Compare With | Commit*

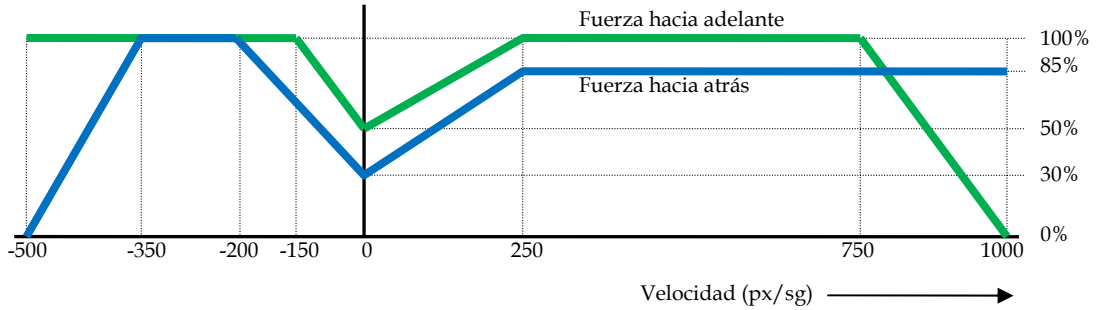
Verás un editor especial en el que puedes observar a la vez dos versiones (izquierda y derecha) del mismo fichero, con los cambios resaltados. Muy útil. Ahora no hay nada que hacer, simplemente tenlo en cuenta para cuando lo necesites en un futuro.

(Eclipse guarda también una *Local History* de los ficheros con la que puedes también observar diferencias y volver a versiones anteriores, pero esto solo en el espacio de trabajo tal y como lo tengamos configurado: lo perderíamos si lleváramos los fuentes a otro ordenador o proyecto nuevo).

Paso 4. **Aceleración realista**. *(necesita el paso 3)*

Vamos a hacer un poco más realista la relación entre el pulsado del acelerador/freno y la aceleración que sufre el coche. Repasemos la física básica (simplificada) que ocurre con un coche que se quiere acelerar:

- El motor aplica una fuerza. Esa fuerza es la que se transmite al coche y provoca su aceleración.
  - Sin embargo, un motor real no puede aplicar siempre la misma fuerza (si no el coche, como en nuestro programa hasta ahora, podría acelerar infinitamente). El motor tiene un régimen en el que aplica la fuerza, dependiendo de la velocidad de giro. Vamos a suponer que la fuerza de nuestro coche sigue estas consideraciones:
    - La fuerza base de marcha adelante es 2000 "Newtixels" (Kg * pixel / seg^2). Marcha atrás, 1000.
    - A esa fuerza se le aplica un % de reducción, de acuerdo al siguiente gráfico:



- La fuerza del motor se reduce con una fuerza inversa de rozamiento, que tiene un componente de rozamiento con el suelo y otro con el aire, según la siguiente fórmula:

$$F_{rozamiento} = masa * coef_{suelo} + velocidad * coef_{aire}$$

- Partamos de la masa coche de 1 Kg, el coef. de rozamiento con el suelo de 15,5. Con el aire, 0,35.

Con estas consideraciones, vamos a cambiar el planteamiento de la aceleración. Prográmalo del siguiente modo:

- Añade las constantes a la clase `Coche` de masa, fuerza base adelante y atrás, y coeficientes de rozamiento.
- Añade un método estático en nuestro `MundoJuego` para calcular la fuerza de rozamiento partiendo de la masa, la velocidad y los coeficientes (según la fórmula). Por ejemplo:

```
public static double calcFuerzaRozamiento( double masa, double coefRozSuelo,
                                            double coefRozAire, double vel ) {
      double fuerzaRozamientoAire = coefRozAire * (-vel);  // En contra del movimiento
      double fuerzaRozamientoSuelo = masa * coefRozSuelo * ((vel>0)?(-1):1);  // Contra mvto
      return fuerzaRozamientoAire + fuerzaRozamientoSuelo;
}
```

- Añade otro método estático para partiendo de la fuerza se calcule la aceleración:

```
public static double calcAceleracionConFuerza( double fuerza, double masa ) {
      // 2ª ley de Newton: F = m*a --->  a = F/m
      return fuerza/masa;
}
```

- Por último en esta clase *MundoJuego*, añade un método para poder aplicar la fuerza al coche. Observa cómo si no hay fuerza externa, la única fuerza que se aplica es la de rozamiento hasta que el coche se para:

```
public static void aplicarFuerza( double fuerza, Coche coche ) {
      double fuerzaRozamiento = calcFuerzaRozamiento( Coche.MASA ,
            Coche.COEF_RZTO_SUELO, Coche.COEF_RZTO_AIRE, coche.getVelocidad() );
      double aceleracion = calcAceleracionConFuerza( fuerza+fuerzaRozamiento, Coche.MASA );
      if (fuerza==0) {
            // No hay fuerza, solo se aplica el rozamiento
            double velAntigua = coche.getVelocidad();
            coche.acelera( aceleracion, 0.04 );
            if (velAntigua>=0 && coche.getVelocidad()<0
                || velAntigua<=0 && coche.getVelocidad()>0) {
                  coche.setVelocidad(0);  // Si se está frenando, se para (no anda al revés)
            }
      } else {
            coche.acelera( aceleracion, 0.04 );
      }
```

```
}
```

- En la clase Coche tendrás que añadir métodos para calcular la fuerza de aceleración del coche, de acuerdo al mapa de fuerza que aparece en las figuras de arriba. Te proponemos este método para la fuerza adelante, inclúyelo y añade el que corresponda para la fuerza atrás:

```
/** Devuelve la fuerza de aceleración del coche, de acuerdo al motor definido en la práctica 2
 * @return  Fuerza de aceleración en Newtixels
 */
public double fuerzaAceleracionAdelante() {
      if (miVelocidad<=-150) return FUERZA_BASE_ADELANTE;
      else if (miVelocidad<=0)
            return FUERZA_BASE_ADELANTE*(-miVelocidad/150*0.5+0.5);
      else if (miVelocidad<=250)
            return FUERZA_BASE_ADELANTE*(miVelocidad/250*0.5+0.5);
      else if (miVelocidad<=250)
            return FUERZA_BASE_ADELANTE*(miVelocidad/250*0.5+0.5);
      else if (miVelocidad<=750)
            return FUERZA_BASE_ADELANTE;
      else return FUERZA_BASE_ADELANTE*(-(miVelocidad-1000)/250);
}
```

- Ya puestos, podrías ver si estos métodos están bien definidos probando si cumplen la gráfica desde un JUnit para la clase Coche… por ejemplo:

```
@Test
public void testFuerzaAceleracionAtras() {
      double[] tablaVel =    { -500, -425, -300, -250, -200, -100,   0,   125,  250,  500,
            1100 };
      double[] tablaFuerza = {    0,  0.5,    1,    1,    1, 0.65, 0.3, 0.575, 0.85, 0.85,
            0.85 };
      for (int i=0;i<tablaVel.length;i++) {
            c.setVelocidad( tablaVel[i] );
            assertEquals( "Velocidad " + tablaVel[i], tablaFuerza[i]*Coche.FUERZA_BASE_ATRAS,
            c.fuerzaAceleracionAtras(), 0.0000001 );
      }
}
```

- Y acabamos este paso. Ya en la clase *VentanaJuego*, cambia la llamada desde las teclas de cursor arriba/abajo para que calculen la fuerza (con los métodos que acabas de definir en la clase *Coche*), y aplicar esa fuerza al coche con el método de *MundoJuego*.

Prueba el coche moviéndose ahora. Debería notarse cómo acelera diferente dependiendo de la velocidad que ya tenga, y cómo si no se acelera se va parando por la fuerza de rozamiento hasta que se detiene. Puedes ajustar las aceleraciones y las inercias cambiando las distintas constantes que has definido.

Paso 5. Estrellitas. *(Independiente)*

El objetivo de nuestro juego va a ser "pillar" las estrellitas que aparezcan en el escenario en posiciones aleatorias. Para ello:

- Crea una clase JLabelEstrella como JLabelCoche, pero con el propio tamaño del fichero (40x40) y de radio de choque 17 píxels. Tienes el gráfico "estrella.png" para construirla. Deja también el giro que usaremos para que esté girando en pantalla continuamente. Añade un atributo para guardar la hora en la que se crea cada estrella (en milisegundos).
- Vamos a adaptar el *MundoJuego* para crear las estrellas. Primero define un método de creación:

```
/** Si han pasado más de 1,2 segundos desde la última,
 * crea una estrella nueva en una posición aleatoria y la añade al mundo y al panel visual */
public void creaEstrella()
```

  Define un atributo interno como *ArrayList* de estrellas, al que debes añadir la estrella que se crea. También necesitarás guardar el milisegundo de la última creación de estrella.
- También tendremos que quitar las estrellas. Para eso define el método (observa que cada estrella tiene su tiempo de creación: tendrás que usarlo para calcular el tiempo que lleva en pantalla):

```
/** Quita todas las estrellas que lleven en pantalla demasiado tiempo
 * y rota 10 grados las que sigan estando
 * @param maxTiempo     Tiempo máximo para que se mantengan las estrellas (msegs)
 * @return  Número de estrellas quitadas */
public int quitaYRotaEstrellas( long maxTiempo )
```

- Prueba a que vayan saliendo estrellas. En el hilo del juego, llama cada 40 milisegundos a estos dos métodos para ver cómo se van creando y destruyendo estrellas (haz que cada 6 segundos se quiten las estrellas).
- Obviamente, queremos que el coche "coma" a las estrellas si pasa por ellas. Define y llama al método

```
/** Calcula si hay choques del coche con alguna estrella (o varias). Se considera el choque si
 * se tocan las esferas lógicas del coche y la estrella. Si es así, las elimina.
 * @return  Número de estrellas eliminadas
 */
```

```
public int choquesConEstrellas()
```

Paso 6. Puntuación. *(Necesita el paso 5)*
La botonera ya no nos sirve de mucho. Quita los botones y pon en su lugar un JLabel *lMensaje*.

Utilízalo para sacar mensajes cuando se pillen estrellas, y visualiza la puntuación (5 puntos por cada estrella pillada).

El juego se acaba cuando han desaparecido de pantalla, sin que podamos pillarlas, 10 estrellas.

Pasos generales. A lo largo del desarrollo de esta práctica:

- Según vayas realizando las tareas, vete actualizando la planificación en gantter, modificando el % completado de cada tarea.
- Cuando tengas cambios suficientes realizados, o quizás cuando hayas acabado un paso completo o vayas a dejar el trabajo hasta el día siguiente, realiza *Commits* sobre tu repositorio Git. Eso irá guardando versiones sucesivas de tus ficheros fuente. A partir de ahora, siempre podrás volver a recuperar u observar un cambio, navegando por las versiones anteriores; e incluso revertir el estado del proyecto a un cambio anterior, si el último no ha merecido la pena.