**BASE DE DATOS**

**TABLASenemigolla el enemigoieza la batalla el usuario.ientes y el escenariose guarda como una cadena de caracteres muy larga, para el**

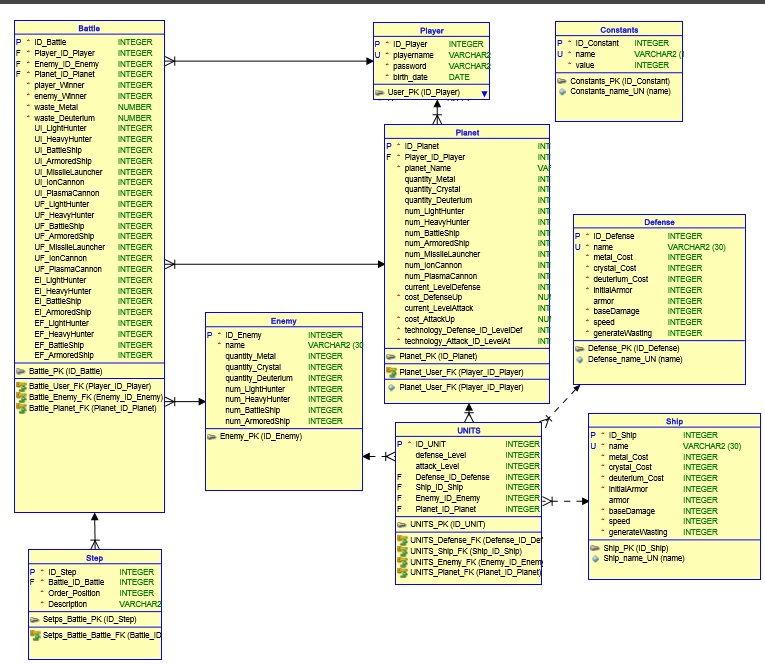


Imagen 1: Modelo relacional de las tablas con sus columnas o campos. La P significa que es una clave primaria, la F que es una clave foránea, la U que ese campo es único y no se puede repetir (Por defecto las claves primarias lo son). Todos los campos que tienen un asterisco rojo significa que son obligatorios y no pueden ser nulos.

La base de datos de este proyecto está compuesta por un total de 9 tablas.

Todas las tablas tienen una clave primaria compuesta por un numero

-**SHIP.** Esta tabla tiene los datos de las naves del juego, tiene un total de 10 campos o columnas:

-**ID\_Defense**, la clave primaria de la nave.

-**name**, nombre de la nave, formato VARCHAR, este es único.

-**metal\_Cost**, coste de metal de construir la nave, en formato INTEGER

-**crystal\_Cost**, coste de cristal de construir la nave, en formato INTEGER, si bien los datos de cristal estan presentes, no se han implementado en el proyecto.

-**deuterium\_Cost**, coste de deuterio de construir la nave, en formato INTEGER.

-**initial\_Armor**, armadura inicial con la que cuenta la nave, en formato INTEGER , con este dato junto con el nivel tecnológico se calculara la armadura final.

-**armor**, nivel de armadura final, en formato INTEGER.

-**baseDamage**, el daño base que inflige la nave, en formato INTEGER , el daño final estar también influido por el nivel tecnológico.

**-speed,** la velocidad, en formato INTEGER.

-**generateWasting,** la probabilidad de generar residuos, en formato INTEGER con un valor de entre 0 y 100.

-**DEFENSE,** esta tabla tiene los datos de las estructuras defensivas. Sus columnas de datos son exactamente las mismas que las de SHIP con las mismas propiedades.

**-PLAYER.** Esta tabla tiene el objetivo de guardar datos de los jugadores. Tiene un total de 4 campos:

-**ID\_Player**. La id del usuario.

**-playername**. El nombre del usuario, es un VARCHAR y tiene la propiedad de que es unico y obligatorio, por lo tanto no podrán haber dos usuarios con el mismo nombre.

-**password**. La contraseña que habrá registrada para el usuario, en formato VARCHAR y obligatoria.

-**birth\_date**. La fecha de nacimiento del usuario, en formato DATE, también es un campo obligatorio.

-**PLANET.** Datos de los planetas, estos pertenecen a los usuarios y almacenan recursos, naves y estructuras defensivas y tienen un nivel tecnológico, que influirá en la fuerza de las naves y las estructuras. Tiene un total de 17 campos o columnas:

**-ID\_Panet**. La id del planeta.

**-Player\_ID\_Player**. La id del usuario al que pertenecerá el planeta, es una clave foránea que liga la tabla PLANET con la tabla PLAYER, es obligatoria y esta en formato INTEGER, como todas las IDs.

**-planet\_Name**. Nombre del planeta, en formato VARCHAR obligatorio especificarlo.

**-quantity\_Metal**. Cantidad de metal que hay en este planeta, formato INTEGER, si no se especifica un valor por defecto se pone a 0.

**-quantity\_Crystal**. Cantidad de cristal que hay en este planeta, formato INTEGER, si no se especifica un valor por defecto se pone a 0.

**-quantity\_Deuterium**. Cantidad de cristal que hay en este planeta, formato INTEGER, si no se especifica un valor por defecto se pone a 0.

**-num\_LighHunter**. El número de naves de tipo LightHunter que hay en el planeta, en formato INTEGER, si no se le especifica valor por defecto se pone a 0.

**-num\_HeavyHunter**. El número de naves de tipo HeavyHunter que hay en el planeta, en formato INTEGER, si no se le especifica valor por defecto se pone a 0.

**-num\_BattleShip**.El número de naves de tipo BattleShip que hay en el planeta, en formato INTEGER, si no se le especifica valor por defecto se pone a 0.

**-num\_ArmoredShip**. El número de naves de tipo ArmoredShip que hay en el planeta, en formato INTEGER, si no se le especifica valor por defecto se pone a 0.

**-num\_MissileLauncher**. El número de estructuras defensivas de tipo MissileLauncher que hay en el planeta, en formato INTEGER, si no se le especifica valor por defecto se pone a 0.

**-num\_IonCannon**. El número de estructuras defensivas de tipo IonCannon que hay en el planeta, en formato INTEGER, si no se le especifica valor por defecto se pone a 0.

**-num\_PlasmaCannon**. El número de estructuras defensivas de tipo PlasmaCannon que hay en el planeta, en formato INTEGER, si no se le especifica valor por defecto se pone a 0.

**-current\_LevelDefense**. El nivel de tecnología defensiva actual que hay en el planeta, esta en formato INTEGER y por defecto, en caso de que no se especifique un valor, es 1, el primer nivel.

**-cost\_DefenseUp**. El coste de deuterio de subir un nivel a tecnología de defensa del planeta, es una variable en formato INTEGER y obligatoria.

**-current\_LevelAttack**. El nivel de tecnología ofensiva actual que hay en el planeta, esta en formato INTEGER y por defecto, en caso de que no se especifique un valor, es 1, el primer nivel.

**-cost\_AttackUp**. El coste de deuterio de subir un nivel a tecnología de ataque del planeta, es una variable en formato INTEGER y obligatoria.

-**CONSTANTS.** Tabla para guardar los valores de las constantes que regularan varios aspectos del balance del juego. Esta tabla solo tiene 3 campos:

**-ID\_Constant**. La clave primaria de la constante.

**-name**. El nombre de la constante, está en formato VARCHAR y es único y obligatorio.

**-value**. El valor de la constante, en formato VARCHAR y también obligatorio.

-**UNITS**. Cada nave y cada estructura es única y por lo tanto estará registrada en esta tabla, que guardara de que tipo es la unidad y a quien pertenece. Tiene 7 campos o columnas:

**-ID\_Unit**. La id de la unidad, mismo formato que el resto de id.

**-defense\_Level**. El nivel tecnológico defensivo que tiene esta nave o estructura, en formato INTEGER.

**-attack\_Level**. El nivel tecnológico ofensivo que tiene esta nave o estructura, en formato INTEGER.

**-Defense\_ID\_Defense**. Una clave foránea opcional, si esta unidad es una defensa, deberá tener este valor especificado, en caso de que no lo sea, será nulo.

**-Ship\_ID\_Ship**. Una clave foránea opcional, si esta unidad es una nave, deberá tener este valor especificado, en caso de que no sea una nave, será nulo. La unidad debe tener o una id de nave o una id de defense, pero no puede tener una de cada, esto está controlado en el procedimiento de inserción de nuevas filas.

**-Enemy\_ID\_Enemy**. La clave foránea del enemigo al que pertenece en caso de que pertenezca a un enemigo. Es opcional, pero junto con Planet\_ID\_Planet, conforma otro caso donde una de las dos es necesaria pero no pueden estar las dos a la vez.

**-Planet\_ID\_Planet**. La clave foránea del jugador al que pertenece en caso de que pertenezca a un jugador.

-**ENEMY.** para guardar datos de los enemigos, estos son como una mezcla simplificada del jugador y el planeta, tienen naves (no disponen de estructuras defensivas) y recursos para hacer naves. Esta tabla tiene un total de 11 columnas o campos:

**-ID\_Enemy**. La id del enemigo, con el mismo formato INTEGER de todas.

**-name**. El nombre del enemigo, en formato VARCHAR y obligatorio.

**-quantity\_Metal**. La cantidad de metal de que dispone ese enemigo, formato INTEGER, valor 0 por defecto si no se le especifica un valor.

**-quantity\_Crystal**. La cantidad de cristal de que dispone ese enemigo, formato INTEGER, valor 0 por defecto si no se le especifica un valor.

**-quantity\_Deuterium**. La cantidad de deuterio de que dispone ese enemigo, formato INTEGER, valor 0 por defecto si no se le especifica un valor.

-**num\_LightHunter**. Número total de unidades del tipo Light Hunter de que dispone ese enemigo en su flota.

**-num\_HeavyHunter**. Número total de unidades del tipo Heavy Hunter de que dispone ese enemigo en su flota.

**-num\_BattleShip**. Número total de unidades del tipo Battle Ship de que dispone ese enemigo en su flota.

**-num\_ArmoredShip**. Número total de unidades del tipo Armored Ship de que dispone ese enemigo en su flota.

-**BATTLE**, para guardar las batallas, esta es la más compleja, se une con llaves foráneas a un usuario, un planeta y un enemigo, siendo estos tres campos requisitos para conformar los contendientes y el escenario. Tiene dos booleanos para guardar quien gana, si el de usuario es cierto y el de enemigo es falso, gana el usuario, en caso contrario gana el enemigo. Tiene datos de residuos generados y registros del número de unidades con el que empiezan y terminan tanto jugador como enemigo.

**-ID\_Battle**. La id de la batalla.

**-Player\_ID\_Player**. Clave foránea que corresponde a la id del usuario que participa en la batalla, obligatoria.

**-Enemy\_ID\_Enemy**. Clave foránea que corresponde a la id del enemigo que participa en la batalla, también en formato INTEGER y obligatoria.

**-Planet\_id\_Planet**. La batalla se lleva a cabo en un planeta, este dato es otra clave foránea, la del planeta donde se lleva a cabo la batalla, formato INTEGER y obligatoria.

**-player\_Winner**. Esta es una variable booleana para guardar si el usuario ha ganado o no, por problemas con la variable de tipo BOOLEAN, se ha decidido emplear una variable de tipo INTEGER que se usara de forma que 0 sea igual a false y 1 a 0.

-**enemy\_Winner**. Esta es una variable booleana para guardar si el enemigo ha ganado o no, por problemas con la variable de tipo BOOLEAN, se ha decidido emplear una variable de tipo INTEGER que se usara de forma que 0 sea igual a false y 1 a 0. Si el booleano de usuario es cierto, el de enemigo será falso y viceversa. Esta y la anterior son también campos obligatorios.

**-waste\_Metal**. Cantidad de residuos de metal que se han generado tras la batalla, en formato INTEGER y obligatorio.

**-waste\_Deuterium**. Cantidad de residuos de deuterio que se han generado tras la batalla, en formato INTEGER y obligatorio.

**-UI\_LightHunter**. El numero inicial de unidades Light Hunters con los que empieza la batalla el usuario.

**-UI\_HeavyHunter**. El numero inicial de unidades Heavy Hunters con los que empieza la batalla el usuario.

**-UI\_BattleShip**. El numero inicial de unidades Battle Ship con los que empieza la batalla el usuario.

**-UI\_ArmoredShip**. El numero inicial de unidades Armored Ship con los que empieza la batalla el usuario.

**-UI\_MissileLauncher**. El numero inicial de estructuras defensivas Missile Launcher con los que empieza la batalla el usuario.

**-UI\_IonCannon**. El numero inicial de Ion Cannon con los que empieza la batalla el usuario.

**-UI\_PlasmaCannon**. El numero inicial de Plasma Cannon con los que empieza la batalla el usuario.

**-UF\_LightHunter**. El número final de unidades Light Hunters con los que termina la batalla el usuario.

**-UF\_HeavyHunter**. El número final de unidades Heavy Hunters con los que termina la batalla el usuario.

**-UF\_BattleShip**. El número final de unidades Battle Ship con los que termina la batalla el usuario.

**-UF\_ArmoredShip**. El número final de unidades Armored Ship con los que termina la batalla el usuario.

**-UF\_MissileLauncher**. El número final de estructuras defensivas Missile Launcher con los que termina la batalla el usuario.

**-UF\_IonCannon**. El número final de Ion Cannon con los que termina la batalla el usuario.

**-UF\_PlasmaCannon**. El número final de Plasma Cannon con los que termina la batalla el usuario.

**-EI\_LightHunter**. El número inicial de unidades Light Hunters con los que empieza la batalla el enemigo.

**-EI\_HeavyHunter**. El número inicial de unidades HeavyHunters con los que empieza la batalla el enemigo.

**-EI\_BattleShip**. El número inicial de unidades Battle Ship con los que empieza la batalla el enemigo.

**-EI\_ArmoredShip**. El número inicial de unidades Armored Ship con los que empieza la batalla el enemigo.

**-EF\_LightHunter**. El número final de unidades LightHunters con los que termina la batalla el enemigo.

**-EF\_HeavyHunter**. El número final de unidades HeavyHunters con los que termina la batalla el enemigo.

**-EF\_BattleShip**. El número final de unidades Battle Ship con los que termina la batalla el enemigo.

**-EF\_ArmoredShip**. El número final de unidades Armored Ship con los que termina la batalla el enemigo.

**-STEP**, para guardar las descripciones de los pasos de las batallas, con los cuales luego se elaborará el informe paso a paso. Su estructura es para cada fila, una id, la batalla a la que pertenece, el orden dentro de la secuencia de pasos y la descripción.

**-id\_Step**. El número de id, exclusivo para cada paso.

**-battle\_Id\_Battle**. Clave foránea correspondiente a la batalla a la que pertenece este paso. Formato INTEGER y obligatorio.

**-order\_Position**. Dentro de la batalla en que se ubica el paso, este puede ser el primer paso, el segundo, el octavo etc. Esta variable indica a que posición de este orden pertenece el paso. Formato INTEGER y obligatorio.

**-description.** Descripción de una secuencia de la batalla, una unidad atacando a otra e infligiendo daño. Formato VARCHAR y obligatoria, pues esta información es el motivo de existir de esta tabla.

**PROCEDIMIENTOSenemigolla el enemigoieza la batalla el usuario.ientes y el escenariose guarda como una cadena de caracteres muy larga, para el**

Para iniciar la base de datos hay un conjunto de procedimientos básicos que se tienen que ejecutar. De esta forma tendremos la base de datos lista para guardar registros del juego.

**-DROP\_TABLE.** Borra todas las tablas del proyecto.

**-CREATE\_TABLE.** Crea de nuevo las tablas del proyecto.

**-INSERT\_DATA.** Inserta los datos de las naves y las estructuras defensivas en la base de datos. Estos son en total 4 filas o registros para SHIP y 3 filas para DEFENSE.

**-INITIALIZE.** Este proceso llama a los 3 procedimientos anteriores: DROP\_TABLE, para eliminar las tablas, paso previo a rehacer la base de datos en su estado inicial. Posteriormente llama a CREATE\_TABLE para crear las tablas. Por último, llama al procedimiento INSERT\_DATA para insertar los datos de las naves y las estructuras defensivas.

De esta forma tenemos la base de datos rehecha y con los únicos datos de naves y estructuras, que no cambiaran, por lo que podríamos en cierto modo entenderlos como constantes. El resto de datos se tendrán que ir insertando desde la ejecución del código de Java.

Algo que aplica a todos los procedimientos es que salvo para las consultas, cuando se ejecutan sentencias que modifican datos de las tablas o de crear o eliminar tablas, en vez de escribir directamente las instrucciones, se emplea EXECUTE IMMEDIATE de un VARCHAR. Es decir, se define primero una secuencia de caracteres con una sentencia de PLSQL (una sentencia es una instrucción que finaliza con punto y coma “;”).

Esto se hace para evitar algunos errores, como que la integridad de las tablas se vea alterada por procedimientos que se ejecutan justo antes de este (Esto es algo que sucedió con los scripts para inicializar la base de datos, que los INSERTs de datos de naves y estructuras defensivas fallaban porque justo antes se eliminaban las tablas, aunque luego se volvieran a crear). Además, el EXECUTE IMMEDIATE es la única forma de ejecutar sentencias DDL, es decir de modificación de la base de datos, por ende, la única opción para crear y eliminar tablas.

Una vez creadas las tablas, hay una serie de procedimientos comunes entre ellas para hacer las operaciones de insertar registros o filas, obtener la información de un registro, modificar los datos de un registro que ya existe y por último eliminar un registro que ya existe.

Estos procedimientos son 4: Los de tipo GET, los de tipo INSERT, los de tipo SET y los de tipo DELETE. La nomenclatura que siguen es el nombre del tipo de procedimiento mas el nombre de la tabla, por ejemplo, para las tabla UNITS tenemos GET\_PLAYER, INSERT\_PLAYER SET\_PLAYER y DELETE\_PLAYER. Estos procedimientos actúan todos a nivel de una sola fila o registro. Se fundamentan en la ID numérica para saber sobre que fila operan, Estas Id se crean automáticamente en los procedimientos INSERT, los de inserción de una fila nueva.

Las tablas de SHIP y DEFENSE solo disponen del método GET ya que su única función es servir para consultar unos datos. Las otras 7 tablas además de GET, disponen de INSERT, SET y DELETE para abrir posibilidades de trabajo y pruebas, aunque no todas las tablas necesitaran todos estos procedimientos en la ejecución del código principal una vez preparado el proyecto, estos han ayudado a hacer pruebas con la base de datos y testear su funcionamiento.

Los 4 procedimientos comunes (TABLA = Nombre de la tabla sobre la que queramos trabajar) son

-**GET\_TABLA(id IN, campo1 OUT , campo2 OUT , campo3 OUT, …)**

Este procedimiento tiene la función de a partir de una id numérica que le entramos, devolvernos todos los datos de la fila o registro de esta id. Debido a eso, la id que le entramos es una variable de tipo IN y el resto serán variables que se devuelvan, por ello son variables de tipo OUT.

La estructura básica de este procedimiento es se llama y se le especifican la id de entrada y las variables de salida, cuyo número variara según la tabla sobre la que trabaje. Al estar todas estas variables declaradas, en la parte de declaración de variables (IS) se declaran únicamente una variable tipo numérica (id\_Encontrada) para comprobar si la id que hemos entrado existe y una excepción (excepción\_Id) hecha para activarse en caso de que la id no se haya encontrado. La variable numérica se obtiene haciendo un conteo de la cantidad de IDs que hay en esta tabla, los resultados posibles son 0, no está esta id en la tabla, o 1, se ha encontrado una sola fila con esa id. Si no se ha encontrado se activa la excepción.

Este sistema para comprobar la existencia de la id es el mismo en los otros procedimientos, salvo en los INSERT donde el requisito para continuar es que la id no exista (pues se inserta y no puede estar repetida)

Si la excepción no se ha activado, entonces la ejecución del procedimiento continuará y se hará un SELECT de todos los parámetros de la fila de datos de esa id, excepto la id misma. Estos parámetros seleccionados se depositarán en las variables de salida, de esta forma los valores estarán disponibles en la salida del procedimiento para ser recogidas por el código de Java.

**-SET\_TABLA(id IN, campo1 IN , campo2 IN , campo3 IN, …)**

Este procedimiento es para modificar un registro que ya existe. Para especificar que registro se quiere modificar, el procedimiento pide que le entremos la id, y luego que le entremos para cada columna o campo, un valor, cada uno de estos valores será el nuevo valor que actualizaremos. Si a alguno de estos campos le entramos un valor nulo (‘’) este procedimiento no actualizará este campo y mantendrá el valor antiguo, de esta forma es posible modificar solo algunos campos, excepto la id, que siempre será fija e inmutable para el registro.

Para cada variable que se actualizará se declara (en el IS) una variable con el mismo nombre y el añadido “\_Insertar”. Cada una de estas variables será la que se insertará. El porqué de esta aparente duplicidad se explicará a continuación. Aparte de las variables de entrada y las de inserción, también declara una variable para comprobar si existe la id y una excepción por si la id no se ha encontrado. Además, declara una variable de tipo VARCHAR, que será el script que se ejecutará posteriormente con el comando *execute immediate*.

Primero hace las comprobaciones de que la id a modificar existe, si existe procede a hacer una comprobación para cada variable que queremos actualizar. Esta comprobación consiste en ver si la variable que queremos entrar como nuevo valor en este campo o columna es nula. Si la variable de entrada no es nula, el valor que se insertara es el que hemos entrado y se hará variable\_Entrada = variable\_Entrada\_Insertar, para hacer el procedimiento de actualización que queríamos hacer originalmente. En caso de que hayamos entrado un valor nulo, la variable que insertaremos adquirirá el valor que había previamente en la columna en ese registro.

Por ejemplo, para el campo contraseña, si al procedimiento le entramos una nueva contraseña, lo actualizara ya que la variable que insertaremos para actualizar cogerá el valor de contraseña que le hemos entrado, en caso de que se entra un valor nulo para la contraseña, se insertara el valor de contraseña antiguo y a efectos prácticos no habrá habido modificación de la contraseña.

Una vez hemos establecido los valores de todas las variables que usaremos para actualizar los registros, preparamos el script que se ejecutará, que será un UPDATE TABLA con un SET para cada columna o campo a modificar salvo la id (por ejemplo password = contraseña\_Insertar), esta última nos servirá para indicar al SET donde hacer dicha modificación con un WHERE id\_Tabla = id\_Player.

Se hace un EXECUTE IMMEDIATE del script y posteriormente se muestran mensajes de confirmación de que el procedimiento se ha ejecutado correctamente y se ha modificado cada una de las variables mostrando el nuevo valor. Estos mensajes de confirmación son solo para comprobar en el programa con el que se escriben los scripts de PLSQL que el procedimiento se ha ejecutado y ha insertado estos valores, no tienen ninguna utilidad para el código de Java que recoge los datos. Posteriormente a los mensajes de confirmación se hace un ROLLBACK para confirmar la ejecución de los cambios en la base de datos. En las excepciones, donde esta la de la id no encontrada y la de errores generales (WHEN OTHERS), justo tras el mensaje de error, hay un ROLLBACK por precaución para deshacer cualquier cambio que se haya podido hacer en la base de datos.

**-INSERT\_TABLA(columna1 IN , columna2 IN , columna3 IN, …)**

Procedimiento para crear en la tabla un nuevo registro o fila, creando una id automáticamente a partir de la más elevada que ya exista en la tabla. Al ser similares, es importante diferenciar entre SET, para actualizar una fila ya existente y INSERT para crear una nueva fila que no existe.

En la declaración de variables declara una variable para la id que insertara y una variable para guardar la id más elevada que ya está presente en la tabla. La id que insertara es la id más elevada sumandole 1. También declara el script que se ejecutara con EXECUTE IMMEDIATE.

El procedimiento empieza obteniendo la id máxima y definiendo la id de la nueva fila como esa id máxima más 1. En caso de que no haya ninguna fila insertada, la id a insertar pasaría a ser 1. procede a hacer la inserción. El script de inserción tiene la instrucción INSERT INTO nombre de tabla VALUES y entre paréntesis dentro de los VALUES a insertar, cada una de las variables.

De forma similar a la función UPDATE, nos muestra un mensaje de confirmación con cada valor que hemos insertado y hace un COMMIT. En las dos excepciones tiene también un ROLLBACK.

**-DELETE\_TABLA(id IN)**

El procedimiento más sencillo, este solamente pide una id, con la cual hará un borrado de toda la fila que corresponda a este id.

Declara las mismas variables que los otros dos procedimientos anteriores, SET y INSERT con una variable para comprobar la id y una excepción que se activa si la id no se existe en esta tabla, pues no se puede borrar una fila que no existe.

Al preparar el script de borrado, usara la instrucción DELETE FROM especificando con WHERE que sea donde la id de la tabla sea igual a la id que hemos entrado. Muestra un mensaje de confirmación de que se ha borrado la id y hace un COMMIT. De igual manera hay dos excepciones, una por id no encontrada y otra de uso general, con un ROLLBACK cada una.

Aparte de estos procedimientos comunes, existen algunos otros más específicos, estos son.

**-GET\_CONSVAL\_BYNAME (nombre IN, valor OUT)**

Este procedimiento tiene el objetivo de devolver el valor de la constante a partir del nombre de esta, de manera que desde Java ya se pueda pedir por el nombre (Nombre que coincidirá con el que le dimos cuando entramos la constante en la base de datos con su valor), de esta forma tenemos un procedimiento más directo, ya que desde Java identificamos las constantes por el nombre.

Este procedimiento se fundamenta en el nombre de la constante, que es la variable que le entramos, al igual que hacían otros procedimientos con la id, comprueba que ese nombre de constante ya exista, en caso de no existir lanzara una excepción.

En caso de que exista el nombre, hará una QUERY sencilla para obtener el valor (columna value) donde el nombre de la variable sea el que hemos entrado. Este valor lo depositaremos en la variable de vuelta (OUT) que será la que recogerá el código de Java.

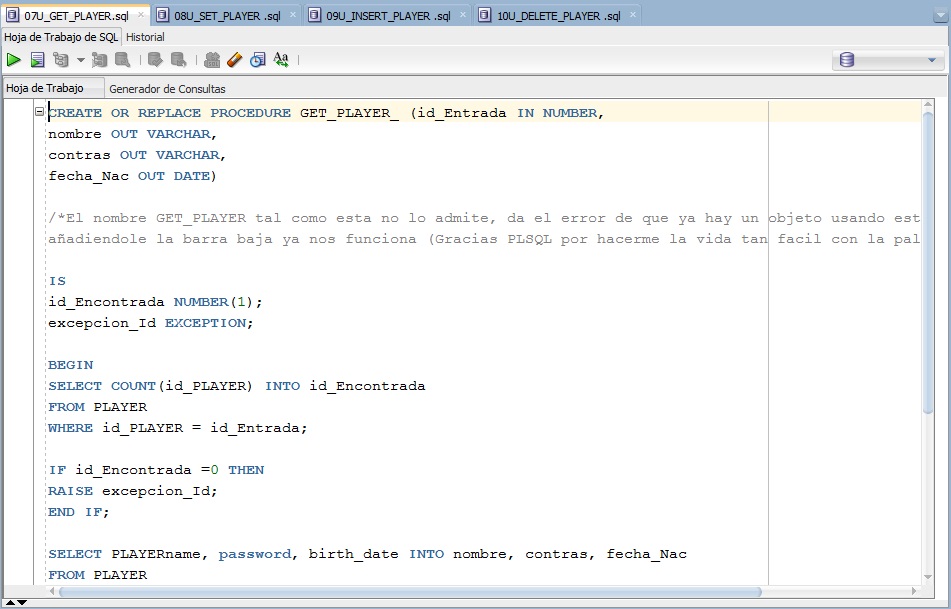
**-COUNT\_ARMY**. Procedimiento para las tablas PLANET y ENEMY, este procedimiento permite actualizar a partir de la información que hay en la tabla UNIT, las cantidades de naves y estructuras que figuran en las tablas PLANET y ENEMY. Proporciona una alternativa a actualizar manualmente desde Java.

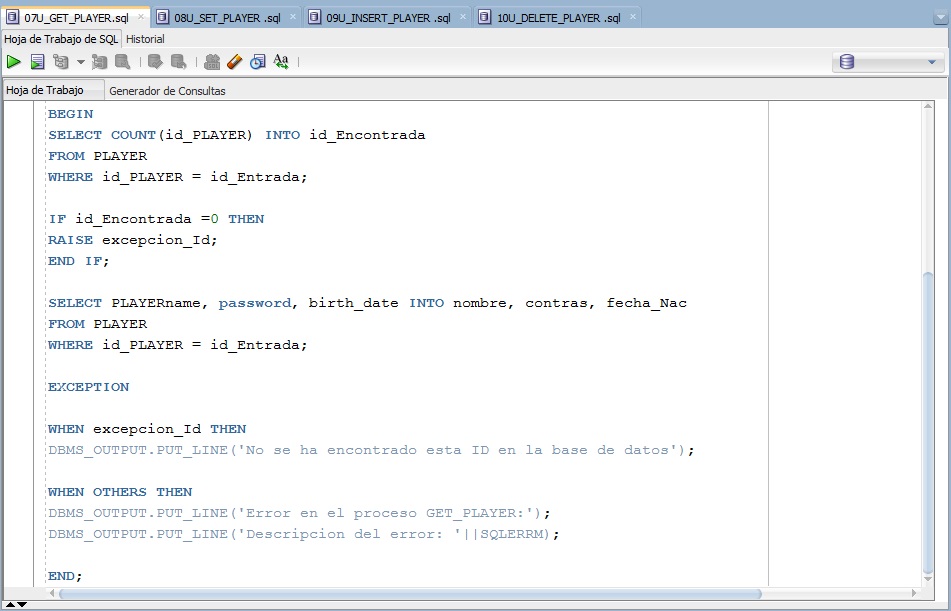
Este procedimiento pide que se le introduzca la Id del planeta o del enemigo. Declara una variable para cada unidad a contar (7 para los planetas y 4 para los enemigos), que serán las cantidades que se actualizarán. Tiene también las variables típicas para comprobar que esa id exista y un VARCHAR para hacer un script de ejecución de código PLSQL.

Para cada cantidad de una unidad, el procedimiento cuenta en la tabla UNITS cuantas filas (y por lo tanto unidades concretas) hay que pertenezcan al planeta (se comprueba a partir de la id de planeta que hemos entrado) y que pertenezcan al tipo que queremos ver, por ejemplo si son “Light Hunter”, que tengan la Id correspondiente o directamente el nombre. La cantidad extraída se deposita en la variable de cantidad que se insertara.

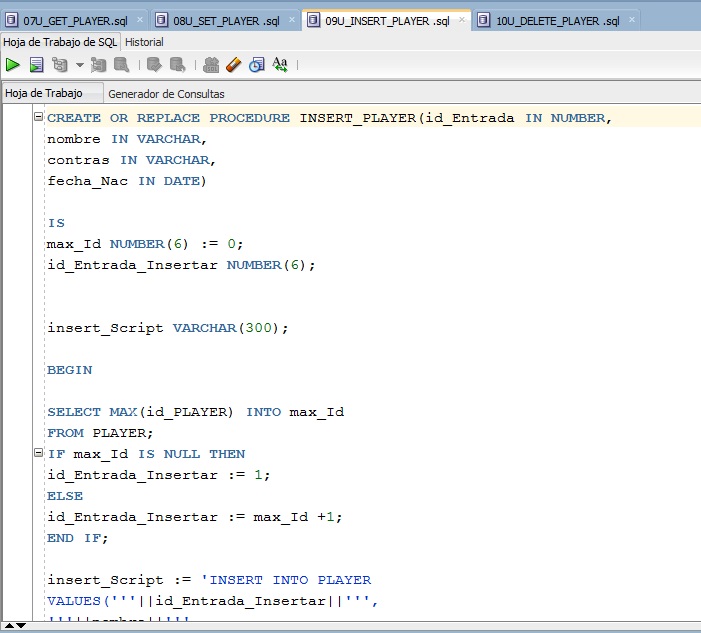
Una vez extraídas todas las cantidades se introducen en el script de tipo INSERT INTO tabla y se ejecuta. Como es una actualización, tiene también COMMIT al final del procedimiento y ROLLBACK en las excepciones.

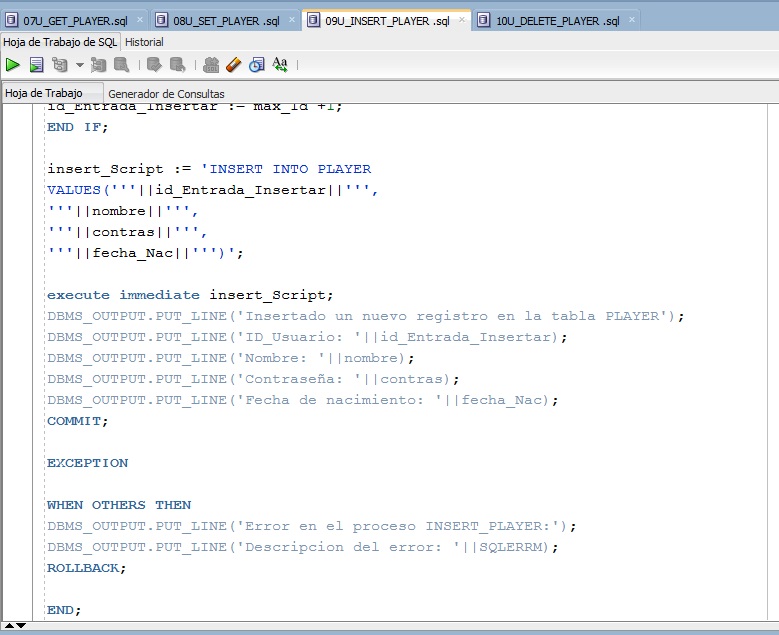
**Capturas**



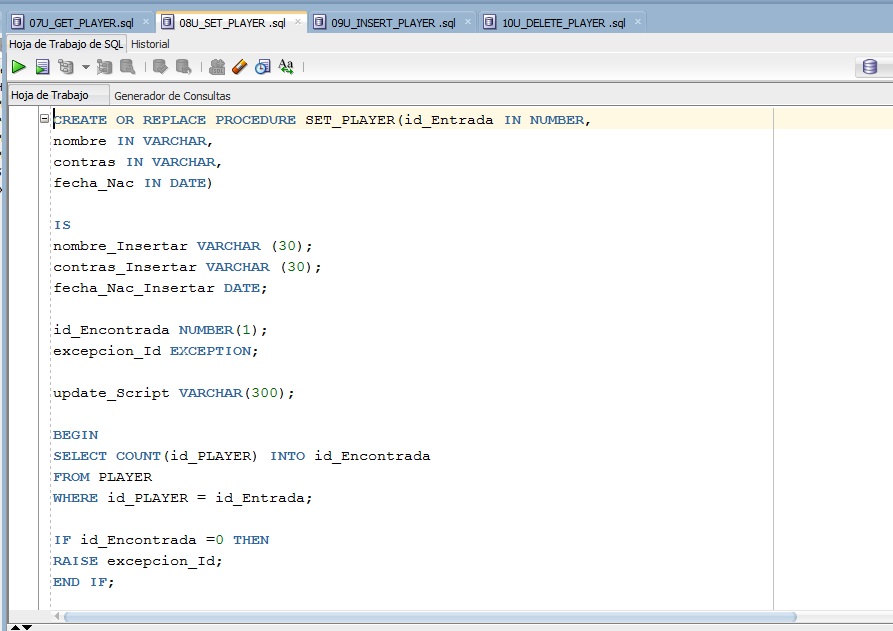


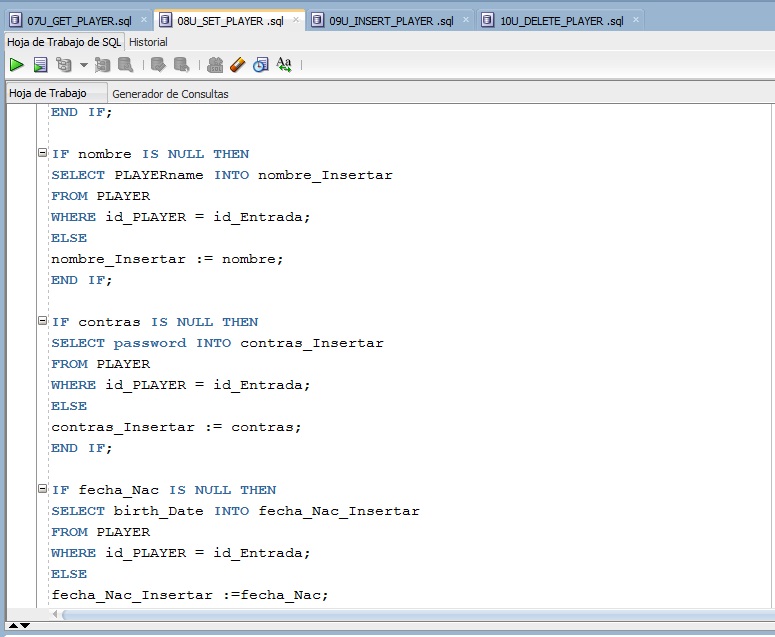
Imágenes 2 y 3 : Ejemplo de un procedimiento GET, en este caso GET\_PLAYER.

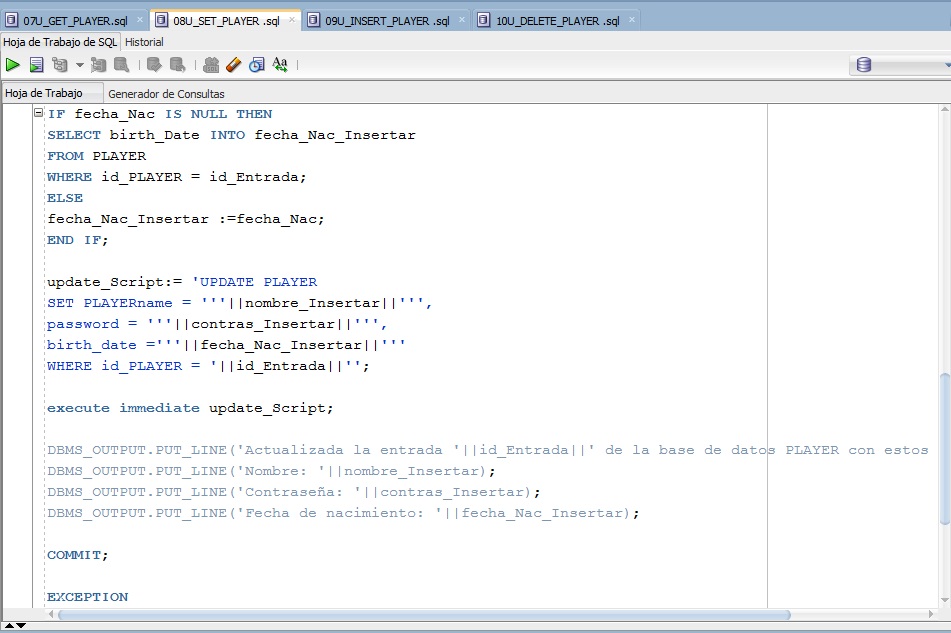


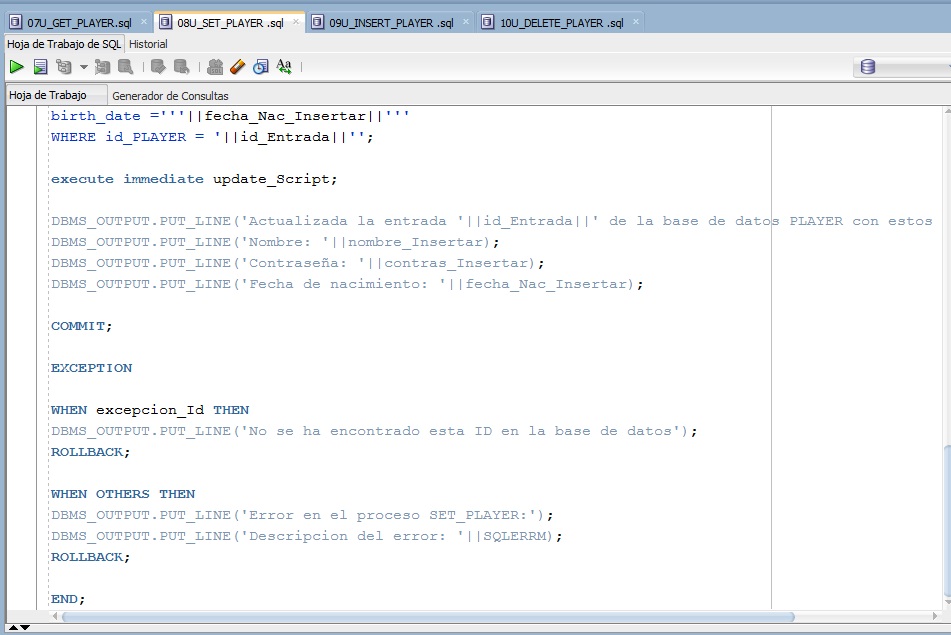


Imágenes 4 y 5: Procedimiento INSERT\_PLAYER.

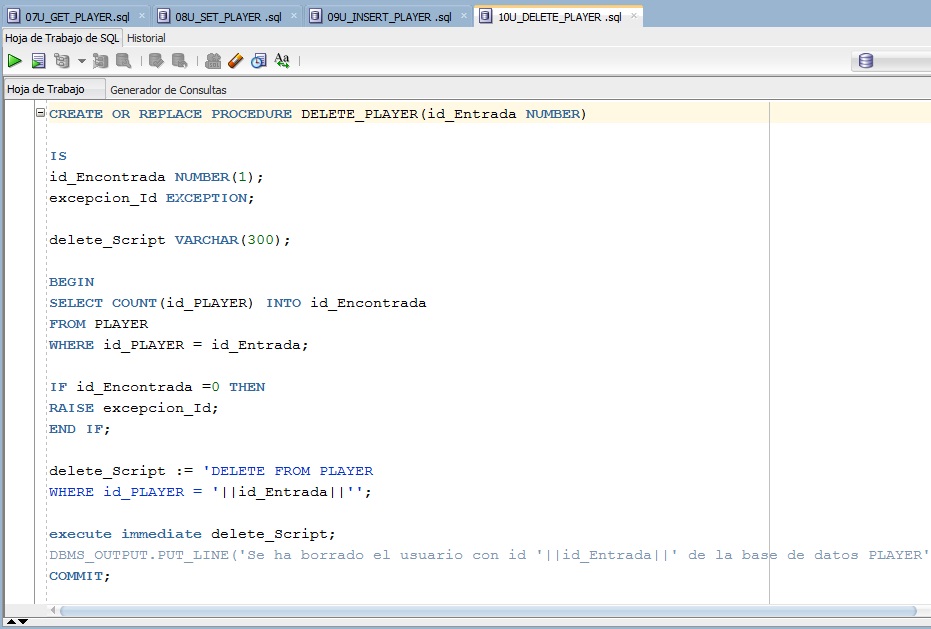


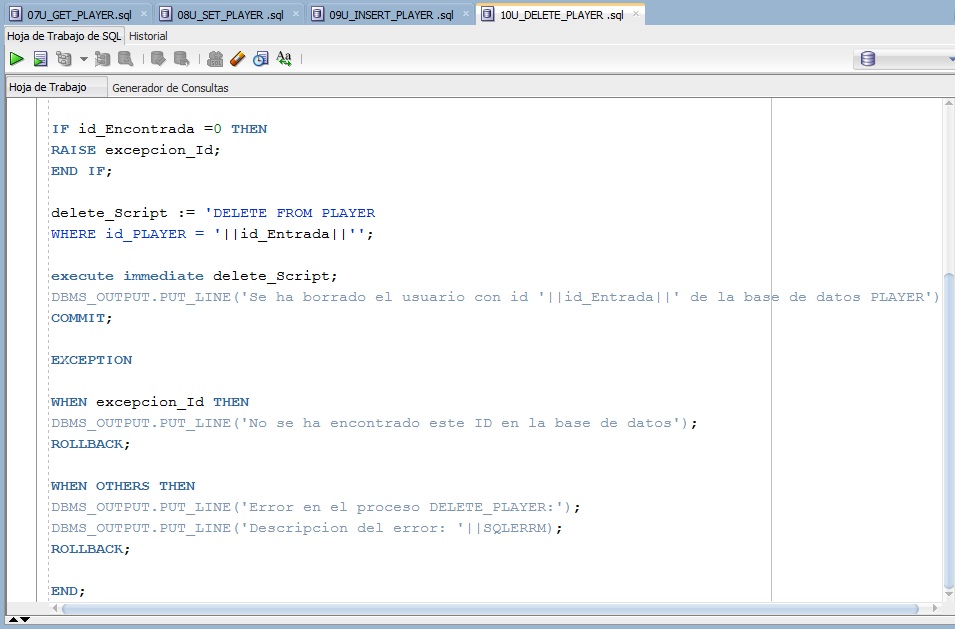






Imágenes 6, 7, 8 y 9: Procedimiento SET\_PLAYER como ejemplo de procedimiento SET.





Imágenes 10 y 11 : Ejemplo de un procedimiento DELETE, en este caso DELETE\_PLAYER.

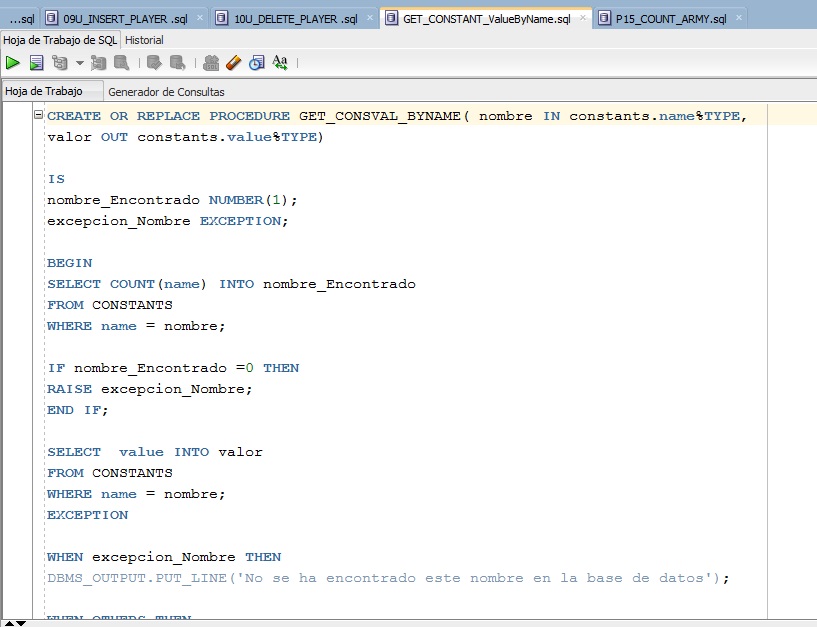
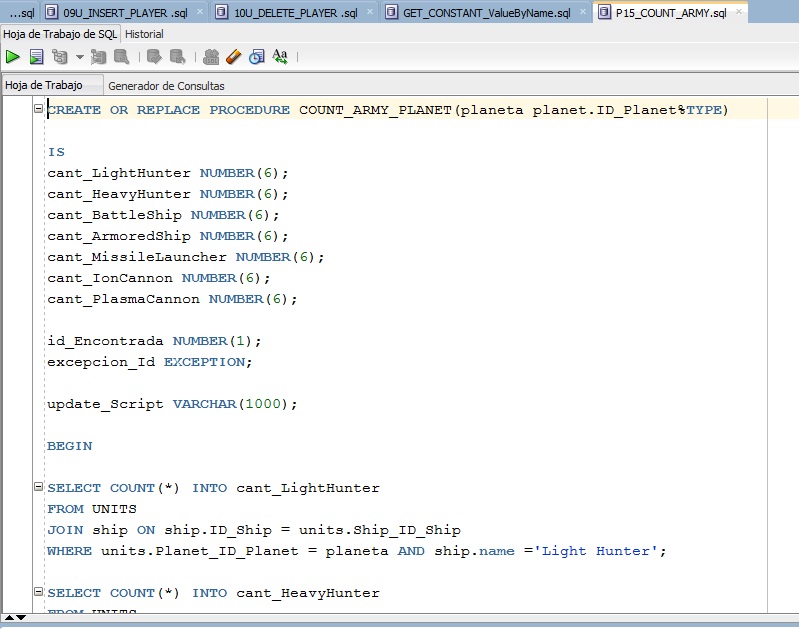
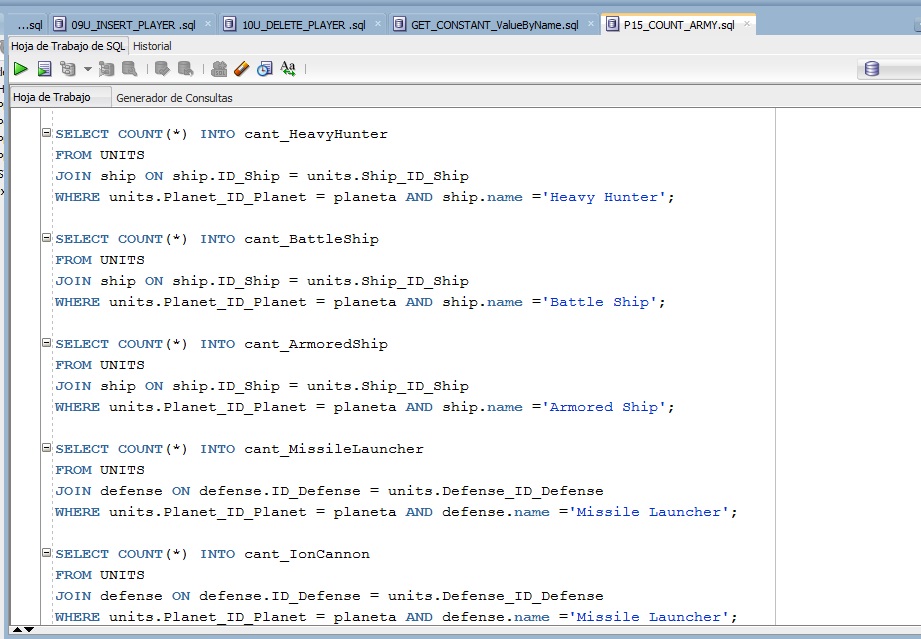
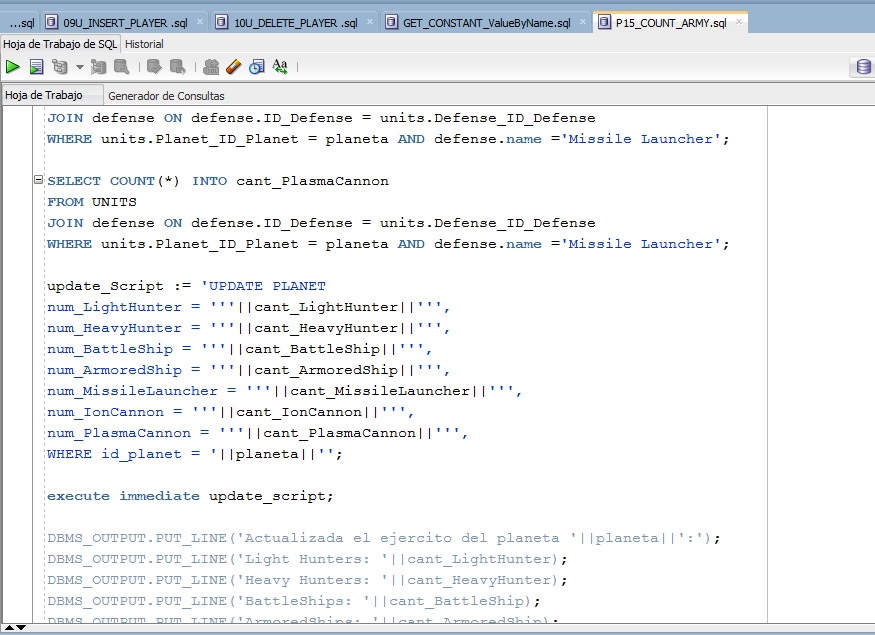
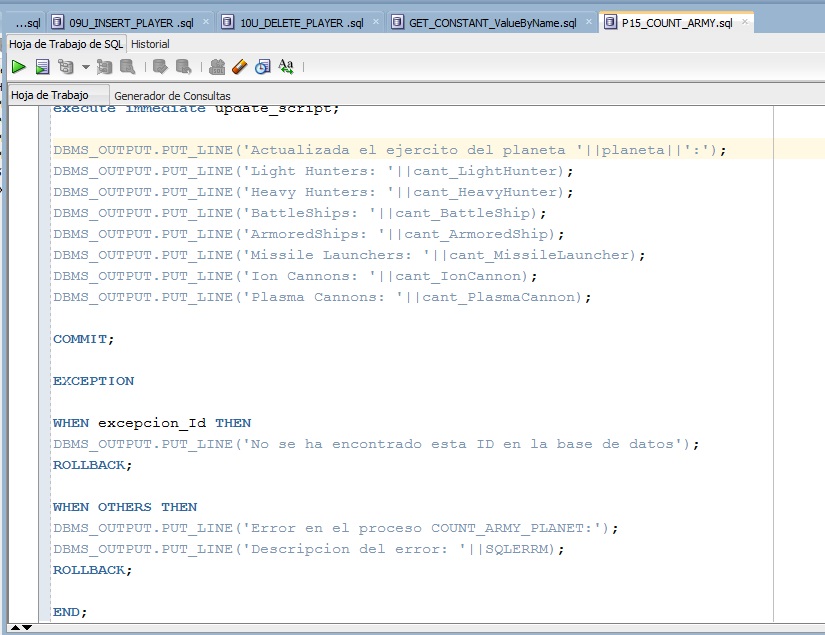


Imagen 12: Procedimiento GET\_CONSTANT\_ValueByName.









Imágenes 13, 14, 15 y 16: Ejemplo de procedimiento COUNT\_ARMY.

**Glosarioenemigolla el enemigoieza la batalla el usuario.ientes y el escenariose guarda como una cadena de caracteres muy larga, para el**

**-Jugador:** Un usuario se corresponde con una cuenta de jugador en el juego, con sus datos, sus planetas, recursos y naves.

**-Unidad:** Una unidad es un elemento del juego que participa en las batallas como combatiente y/o defensor, formando parte del ejército de un planeta y teniendo capacidad ofensiva sobre otras unidades pudiendo dañarlas y destruirlas. Existen dos tipos de unidades, las naves, con capacidad para desplazarse a otros planetas para luchar en ellos, así como defender su propio planeta y las estructuras defensivas, centradas exclusivamente en la defensa de un planeta y sin capacidad de moverse para atacar otros planetas.

**-Planeta:** Dentro del universo del juego, los planetas son localizaciones donde se llevan a cabo los acontecimientos del juego. Estos pertenecen a un usuario, generan recursos y tienen un ejército compuesto por naves y estructuras defensivas, al ser el escenario principal, son también campos de batalla.

**-Inicializar la Base de Datos:** Inicializar la base de datos es reestablecerla al estado inicial, el decir con todas las tablas vacías excepto las de naves y estructuras defensivas que tendrán los datos insertados de entrada.

**-Fila/Registro**: Dentro de una tabla de la base de datos, una fila corresponde a un conjunto horizontal de datos, teniendo un dato de cada columna. Cada fila corresponde a la información completa de un elemento determinado que se pretende almacenar en la base de datos, por ejemplo, las naves, cada fila correspondería a una nave con toda su información.

**-Campo/Columna**: Dentro de una tabla de la base de datos, una columna corresponde a un conjunto vertical de datos, teniendo un dato de una tipología determinada en cada fila con la que cruza. Volviendo al ejemplo de la nave en la definición de fila, una columna contendría el nombre de todas las naves de la tabla Se puede entender la tabla como un conjunto de columnas y filas.

**-Query:** Trozo de código en lenguaje SQL cuya ejecución realiza una consulta sobre la base de datos y devuelve uno o varios resultados, que se pueden visualizar directamente o guardar en variables

**-Constante (en Base de Datos):** Una constante es una variable cuyo valor modula alguna mecánica concreta del juego. Ejemplos de variables son las tasas de generación de recursos o las probabilidades que intervienen en las batallas.