Escuela Técnica Superior de Ingeniería Informática
Universitat Politècnica de València

Aplicación JavaFX para editar contenidos de un videojuego en XML

Proyecto Final de Carrera

Ingeniería Informática

Autor: Heine Sánchez Frade

Director: Mario Gómez Martínez

2015

Resumen

Command Ops: Battles from the Bulge es un videojuego de estrategia centrado en la Segunda Guerra Mundial, concretamente recrea la ofensiva sorpresa alemana y la contraofensiva de los aliados ocurridas en la región de Ardenas en 1944.

Toda la información que detalla el armamento, el personal y los vehículos del juego están contenidos en un fichero XML que se denomina "*estab*" (abreviación del inglés *establishment*).

El propósito de este proyecto es crear un editor de *estabs* complementario al original con la principal característica de poder trabajar con dos *estabs* simultáneamente, uno en modo lectura y el otro en modo edición.

Palabras clave: JavaFX, Java, estab, editor, BFTB, Bulge.

Tabla de contenidos

Tabla de figuras

Introducción

1.1 Sobre el juego

Command Ops: Battles from the Bulge es un videojuego del género RTS (*real time strategy*) que recrea los escenarios de batallas que ocurrieron en las Ardenas[1], jugando desde cualquiera de los dos bandos contra otro jugador o contra la inteligencia artificial.

En este juego destaca la profundidad de detalles tanto a nivel histórico como a nivel de jugabilidad. Los escenarios por defecto son una recreación fiel a las batallas libradas en el pasado. La inteligencia artificial está increíblemente bien desarrollada, permite elegir el grado de independencia de tus tropas y en el caso de ser enemigo crea situaciones tensas difíciles de contrarrestar.

Una de las mejores características es la posibilidad de crear escenarios propios con los editores de mapas y unidades que vienen incluidos en el paquete. Esto lleva a infinitas horas de juego en nuevas campañas que no tienen por qué estar relacionados con el tema original.

1.2 Motivación

Una interfaz gráfica es una manera inteligente de gestionar grandes cantidades de información. Una de las ventajas más importantes es que reduce el error humano puesto que garantiza la consistencia de los datos.

A la hora de editar un archivo cargado de contenido es necesario mantener el orden y la coherencia. Una aplicación gráfica es la herramienta perfecta para mantener ambos sin esfuerzo por parte del usuario.

Este proyecto está basado en un editor ya creado por los propios desarrolladores del juego. El editor original de unidades (editor de *estabs*) proporciona la coherencia de datos arriba mencionada, pero carece de una funcionalidad: la edición en paralelo de dos archivos de *estab*.

Es esta carencia lo que motiva al desarrollo de una nueva aplicación, teniendo el objetivo de proporcionar la edición en paralelo y la compatibilidad de *estabs* tanto con el juego como con el propio editor original. Además, por supuesto, de una mejora en la interfaz simplificándola y dándole un toque más intuitivo.

1.3 Estructura de la memoria

La estructura de esta memoria sigue los pasos cronológicos que se siguieron para la creación del proyecto. Siguiendo el ciclo de desarrollo tradicional se empezó por el análisis, seguido del diseño y por último la implementación.

- Capítulo 2: Se analiza el problema, se definen términos específicos y se detallan los requisitos del proyecto.
- Capítulo 3: Diseño de la solución y la metodología del desarrollo.
- Capítulo 4: Explicación de la configuración de la interfaz gráfica.
- Capítulo 5: Descripción detallada del estructura de la aplicación, su implementación y las herramientas utilizadas.
- Capítulo 6: Conclusiones finales del proyecto.

Al final de la memoria se incluye un glosario en el que se definen términos relevantes al proyecto que puedan ser poco conocidos o de difícil interpretación.

Análisis

El objetivo de este apartado es explicar los conceptos esenciales en los que se basa el proyecto, definir el modelo del dominio sobre el que se trabaja y especificar los requisitos que se han de cumplir.

2.1 Definición del estab

Estab es el término más importante del proyecto, todo gira en torno a este concepto. Es por esta razón que es vital definirlo y entenderlo.

La palabra ***estab*** es una abreviación de *establishment* en inglés. Cuando hablamos de un *estab* en general nos referimos al conjunto de propiedades de unidades y de equipamiento militar. Esto puede variar desde la munición de un arma, tanques o motocicletas, hasta las naciones y los bandos que luchan en las guerras. Por extensión, también se le puede llamar *estab* al archivo en el que se guarda esta información.

Como se verá más adelante, se denomina **elemento** a cualquiera de los objetos definidos en el *estab* como los que hemos mencionado anteriormente: armas, vehículos, naciones, etc.

Dado que el objetivo principal de la aplicación es la edición de dos *estabs* distinguimos a cuál nos referimos mediante dos términos: origen y destino.

- El *estab* **origen** se refiere al *estab* que se abre en modo lectura, destinado para acceder y mostrar la información de los elementos sin la posibilidad de modificarlos.

- El *estab* **destino** es el *estab* que se abre en modo escritura. En él se pueden pegar elementos del *estab* origen, duplicar los ya existentes, eliminarlos o simplemente editarlos.

2.2 Modelo del dominio

En el caso de este proyecto se da la particularidad de que las clases y los conceptos sobre los que se trabaja ya están definidos y estructuradas. La manera en la que se ha procedido a crear el modelo del dominio ha sido mediante ingeniería inversa a partir de la información contenida en los diferentes *estabs* que proporciona el paquete original del juego.

Sorprendentemente muchas de las clases incluidas en el modelo de los *estabs* son totalmente redundantes, con atributos innecesarios que carecen de lógica. Por esta razón se decidió meter las clases originales en una capa de persistencia separada de la del modelo de la aplicación. En vez de modificar estas clases, independientemente de su utilidad, el proyecto se adaptó a la estructura del *estab.*

Cabe mencionar que aunque se garantice la compatibilidad en ambos sentidos, existen clases, atributos y otros campos adicionales que se han añadido para mantener información sobre del estado de los elementos y el *estab*, como puede ser por ejemplo las fechas de edición o la versión del editor.

En el diagrama de clases (Figura 1) se puede apreciar la relación entre las clases básicas del proyecto que representan los elementos del *estab*:

- *Estab*: La única funcionalidad de esta clase es proporcionar los contenedores donde almacenar el resto de elementos.

- Vehículos (*Vehicle*)*:* Plataformas para transportar personal y/o armas. Se define por sus características geométricas, el peso y su tipo (tanque, motocicleta, blindado…). El rendimiento de un vehículo viene determinado por la capacidad de personal, el tamaño del depósito, la velocidad media y máxima en diferentes terrenos, y el valor de armadura en cada área del vehículo.

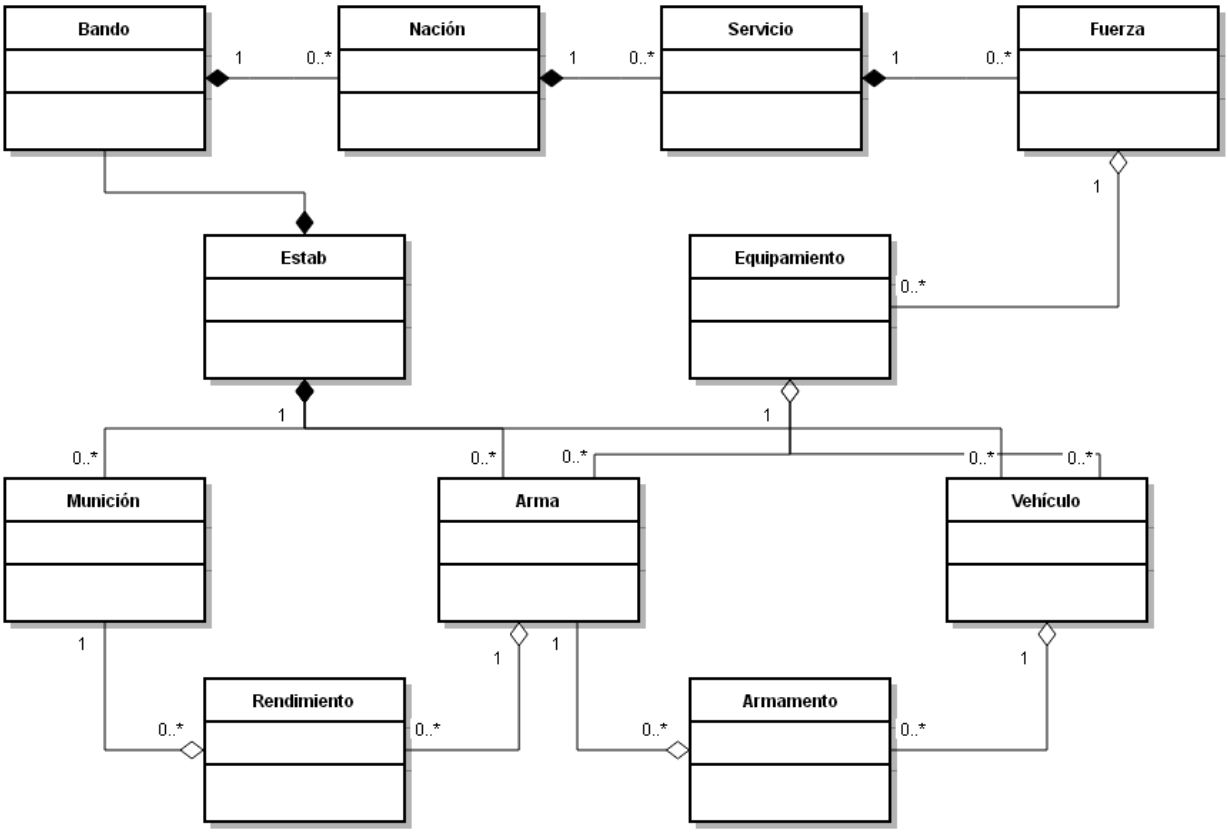

Figura 1. Diagrama de clases conceptuales.

- Armas (*Weapon*): Instrumentos que utiliza el personal militar para combatir. El tipo de arma depende de la nación, del pelotón y del tipo de combate, permitiendo realizar diferentes tácticas militares como bombardear, fuego de supresión, combate antiaéreo, etc.

- Rendimiento (*Performance):* Se pueden especificar perfiles de rendimiento para cada arma dependiendo del tipo de combate en el que se incluya el tipo de munición, la puntería, el alcance, la penetración, el calibre, el número de personas necesarias para operar el arma, la velocidad de disparo y el peso.

- Munición (*Ammo*): La munición no es exclusiva de un arma específica, puede ser utilizada por diferentes armas y al mismo tiempo es posible que un arma utilice varios tipos de munición.

- Armamento (*Armament*): Las fuerzas son capaces de contener suministros de los tipos vehículo y arma. A esto se le denomina armamento y sirve para especificar qué vehículos y armas tendrá una fuerza así como la cantidad de estos.

- Fuerza (*Force*): Tipos de personal militar. Existen diferentes tipos de fuerza como infantería, artillería, blindados o equipos de logística. Además es posible detallar en más profundidad el tipo de una fuerza con el subtipo, si por ejemplo se trata de infantería motorizada, de montaña, etc. El tamaño de la fuerza se determina por los términos brigada, batallón, regimiento (entre otros) y el número de personas que la componen. Una fuerza tiene varios roles según su posición en el campo de batalla (primera línea o soporte), dispone de varios medios de desplazamiento (aire, agua, en ruedas, a pie…) con su velocidad media y máxima, y también el tiempo que tarda en realizar diferentes acciones como atrincherarse, desplegarse, etc.

- Servicio (*Service*): Agrupación de fuerzas bajo un sistema de organización determinado (con su propia estructura, rangos y emblemas). Generalmente las fuerzas de un país se dividen en varios servicios, como Armada (naval), Ejército del Aire y Ejército de Tierra.

- Nación (*Nation*): Una nación se compone de varios servicios y representa al ejército de un país. El tipo de vehículos, munición, armas y fuerzas son específicos de cada nación.

- Bando (*Side*): En una partida un bando es el equivalente a un jugador. Un bando puede estar formado por varias naciones como los Aliados o el Eje. Es posible también que haya un tercer bando neutro independiente de los otros dos.

Por último y para concluir esta sección, remarcar que existen otros elementos con los que no se ha trabajado llamados efectos de formación y radios. En ciertas ocasiones aparecen mencionados aunque no estén incluidos en el dominio y no han sido eliminados por si las funcionalidades que los utilizan se añaden posteriormente.

2.3 Especificación de requisitos

Definimos los requisitos como condiciones necesarias que se han de cumplir a la hora de dar por concluida la aplicación. Estas condiciones afectan tanto a la interfaz gráfica como a las funcionalidades que debe proporcionar el código.

- Requisitos generales de la aplicación:
    1) Gestionar *estabs* simultáneamente: Será posible abrir hasta dos archivos *estabs* al mismo tiempo, uno en modo lectura sin posibilidad de ser modificado y otro en modo edición.

    2) Utilizar un formato compatible con el editor original y el juego. Los *estabs* guardados con la aplicación se podrán utilizar en el juego y el editor original. Del mismo modo, la aplicación podrá gestionar *estabs* del editor original.

    3) Detección y corrección automática de errores Se restringen acciones que puedan causar incongruencias e incoherencias en los *estabs*. Idealmente el acceso al archivo de *estab* nunca se realizará de forma manual, el estado esperado es el de un archivo cerrado por la aplicación o por el editor original.

    4) Creación de fuerzas mediante composición. Se da la posibilidad de establecer una jerarquía entre fuerzas con con un compositor, de manera que varias fuerzas puedan estar contenidas en otra (por ejemplo, varios pelotones dentro de una compañía).

- Requisitos sobre elementos básicos de un *estab*:
    1) Leer: Se mostrará la información guardada de cada elemento tanto en el *estab* de origen como en el de destino.

    2) Añadir: En el caso del *estab* destino, se podrán crear nuevos elementos y almacenarlos de forma persistente. La adición contempla los casos en los que el elemento sea totalmente nuevo, que sea una copia del *estab* de origen o que sea una réplica duplicada del propio *estab* destino.

3) Editar: Será posible modificar los elementos que se encuentren en el *estab* destino.

4) Eliminar: Si un elemento está contenido en el *estab* destino, se da la posibilidad de eliminarlo.

- Requisitos sobre el *estab* origen:
  1) Estará bloqueado en modo lectura sin posibilidad de cambios.

  2) Se podrá copiar los elementos seleccionados.

- Requisitos sobre el *estab* destino:
  1) Se abrirá en modo escritura mientras sea diferente del estab origen.

  2) Se podrán pegar los elementos copiados en el estab origen.

  3) En caso de que los elementos seleccionados estén en el propio estab destino, se duplicarán dichos elementos.

- Requisitos sobre listas de elementos en ambos *estabs*:
  1) Selección total: Marcar todos los elementos de una lista como seleccionados).

  2) Selección parcial: Marcar únicamente determinados elementos.

  3) Anular selección: Deshacer cualquier selección de elementos realizada.

  4) Selección en cascada: En el caso de ser un bando, nación o servicio, seleccionar dicho elemento creará un efecto en cascada seleccionado todos sus subelementos.

  5) Preservación de la jerarquía: Seleccionar una nación, servicio o fuerza también seleccionará de manera automática los elementos superiores que lo contiene. Así mismo, al copiar uno de estos elementos se asegura la reconstrucción de la jerarquía en el destino.

Diseño

Este capítulo aborda las decisiones de diseño tomadas a la hora de organizar el código y la interfaz gráfica. Comienza explicando la arquitectura utilizada basada en el patrón MVC, detallando la funcionalidad de cada capa. Posteriormente se presentan todas las secciones de la interfaz gráfica y el porqué del uso de JavaFX.

3.1 Arquitectura

3.1.1 Patrones MVC y Observador

El diseño de este proyecto se ha planteado según el patrón **Modelo-Vista-Controlador** (MVC). Esto significa que se ha optado por dividir las clases en tres capas distintas: la de datos (modelo), la interfaz (vista), y la de negocio que actúa como mediador entre las otras dos (controlador).

Seguir este patrón conlleva una serie de ventajas:

- La vista no contiene ningún tipo de lógica y puede ser trabajada por diseñadores sin necesidad de programar.
- Se pueden crear módulos de código reutilizables y sustituibles sin afectar a la totalidad del proyecto.
- La modularidad facilita las pruebas de unidad y el mantenimiento del código.

En la capa de interfaz únicamente se encuentran clases y archivos que describen y definen el diseño gráfico. La capa de modelo gestiona los datos que la aplicación procesa, es donde se encuentran las clases responsables de la creación, eliminación y actualización de los elementos del *estab*. La última capa de controlador es donde se rige la comunicación y el flujo de información entre las otras dos.

Hay varias alternativas a la implementación de este patrón. Por ejemplo, aislando totalmente las capas de modelo y vista la una respecto de la otra, siendo únicamente responsabilidad del controlador el paso de información entre ambas capas. Otra alternativa más común es que el controlador prepare las dos capas

para que respondan ante eventos independientemente de la capa en la que se han originado basándose en el patrón Observador.

Se habla de patrón Observador cuando existe un sistema en el que un objeto sujeto está pendiente del estado de otros objetos asociados. En vez de hacer que los objetos se comprueben constantemente el estado entre sí, la idea es que cuando un objeto cambie de estado notifique a todos los que están interesados en dicho cambio.

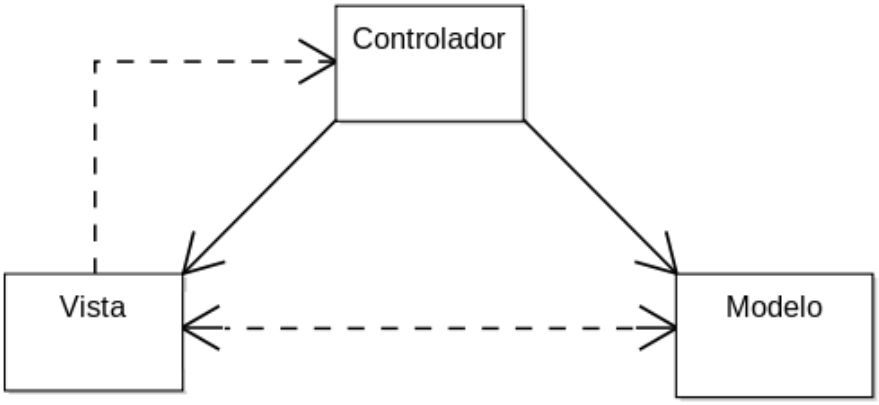

Figura 2. Capas del patrón MVC. Las líneas continuas indican una asociación directa y las discontinuas una asociación indirecta que muestran el patrón Observador.

En este caso el patrón Observador ocurre entre las capas de vista y modelo mediante los enlaces de propiedades. El concepto de encapsular varios objetos en otro ya existe con las JavaBeans. Estas clases disponen de métodos para acceder y modificar los objetos encapsulados (*getters* y *setters*). El modelo de propiedades es una extensión de esta noción, proporcionando además otros métodos como el de añadir observadores y enlaces (*bindings*). Enlazar una propiedad sincroniza el valor de ésta con su destino, de manera que ambos siempre contendrán el mismo valor.

3.1.2 Separación entre modelo y capa de persistencia

En el apartado de análisis se ha comentado que las clases de elementos sobre las que nos basamos están ya diseñadas y estructuradas en los archivos *estab.* Al realizar el diseño del proyecto se ha seguido una política de no modificar de ninguna manera estas clases, lo que implica:

- Las clases básicas solo contienen atributos y métodos para acceder y modificarlos, sin ningún tipo de lógica.

- Crear una capa de persistencia aislada donde almacenar las clases básicas. Esta capa únicamente se asociará a la capa de modelo de forma directa.

- Crear adaptadores que permitan acceder a los objetos de estas clases y añadirles funcionalidad necesaria.

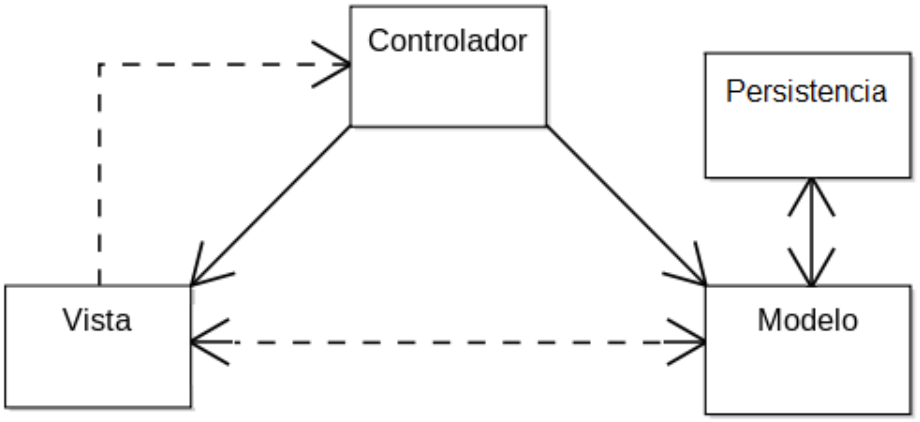

Figura 3. Capa de persistencia añadida al patrón MVC.

Las clases de modelo de elementos son adaptadores que funcionan como un puente entre la aplicación y los respectivos elementos del *estab*, de manera que existe una clase por cada tipo de elemento. Siguiendo la nomenclatura que se ha usado en el proyecto, este proceso se describe como “obtener el modelo del elemento” y se puede ver con más claridad en el diagrama de secuencias (Figura 4).

El objetivo de estos adaptadores consiste en proporcionar una versión del elemento para la capa modelo que contenga: mismos valores de los atributos; métodos para acceder, modificarlo y proveer funcionalidad adicional como la de comprobar si dos elementos son iguales; y proporcionar propiedades que el controlador utilizará para enlazar la vista con el modelo siguiendo el patrón Observador.

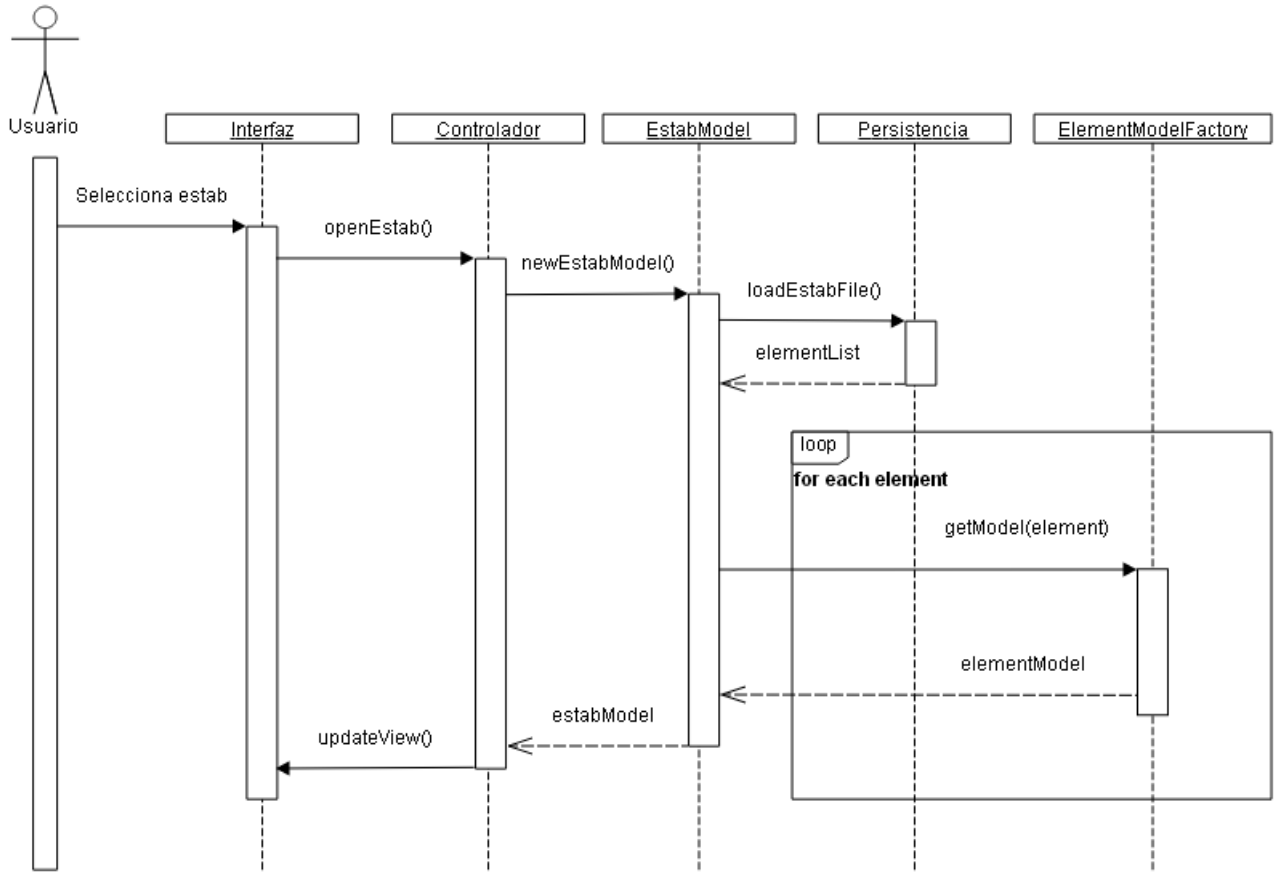

Figura 4. Diagrama de secuencia en el que se describe la carga de un estab.

3.2 Diseño de la capa de Persistencia

La capa de persistencia se deriva directamente de los datos a partir de un esquema XSD en el que está especificada la estructura del *estab.* Todas las clases que representan elementos, incluyendo enumeraciones y tipos de datos, están contenidas en esta capa. En el anexo B se puede ver dicho esquema con las clases y sus atributos.

El diagrama de clases completo muestra estas clases y la manera en la que están relacionadas. En la primera parte (Figura 5) las clases a remarcar son la de munición (`Ammo`), la de arma (`Weapon`) y la de vehículo (`Vehicle`) que giran en torno a la clase contenedora `EstabData`. La segunda parte (Figura 6) sigue la conexión de la clase de bando (`Side`) hacia la parte donde se encuentran las clases de nación (`Nation`), de servicio (`Service`) y de fuerza (`Force`).

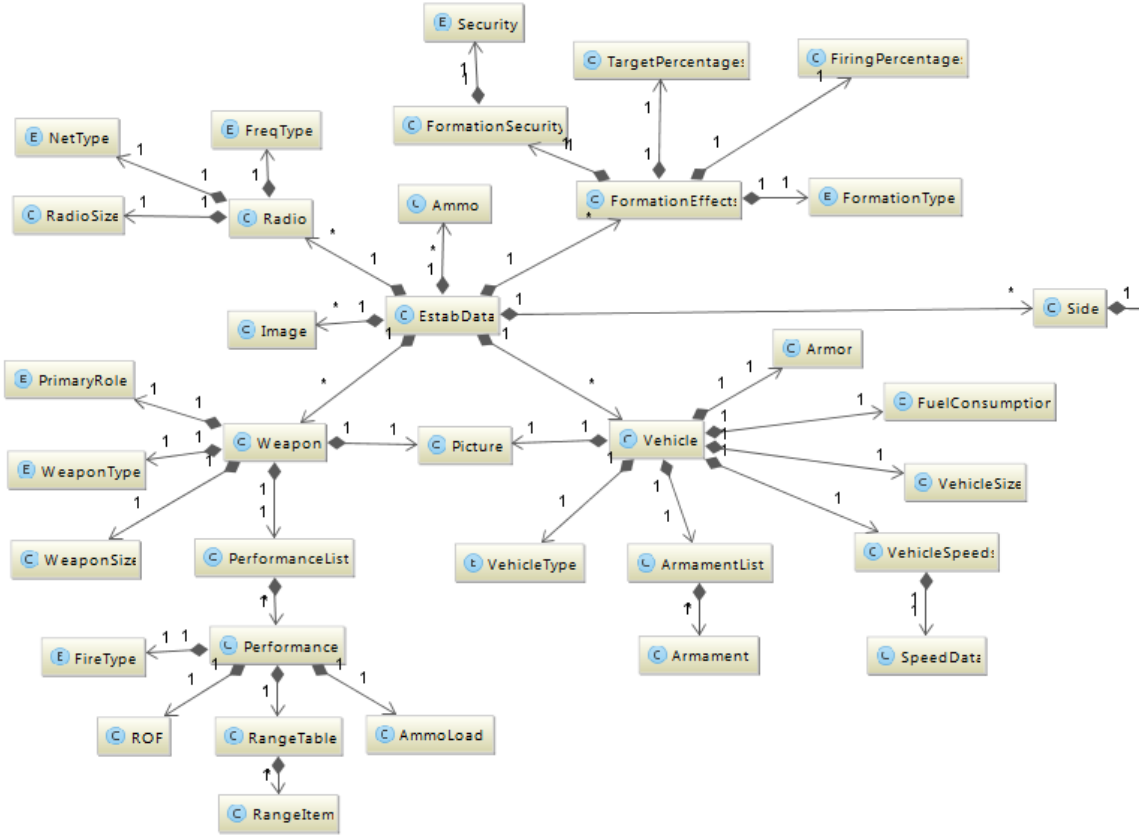

Figura 5. Diagrama completo de clases. Parte 1.

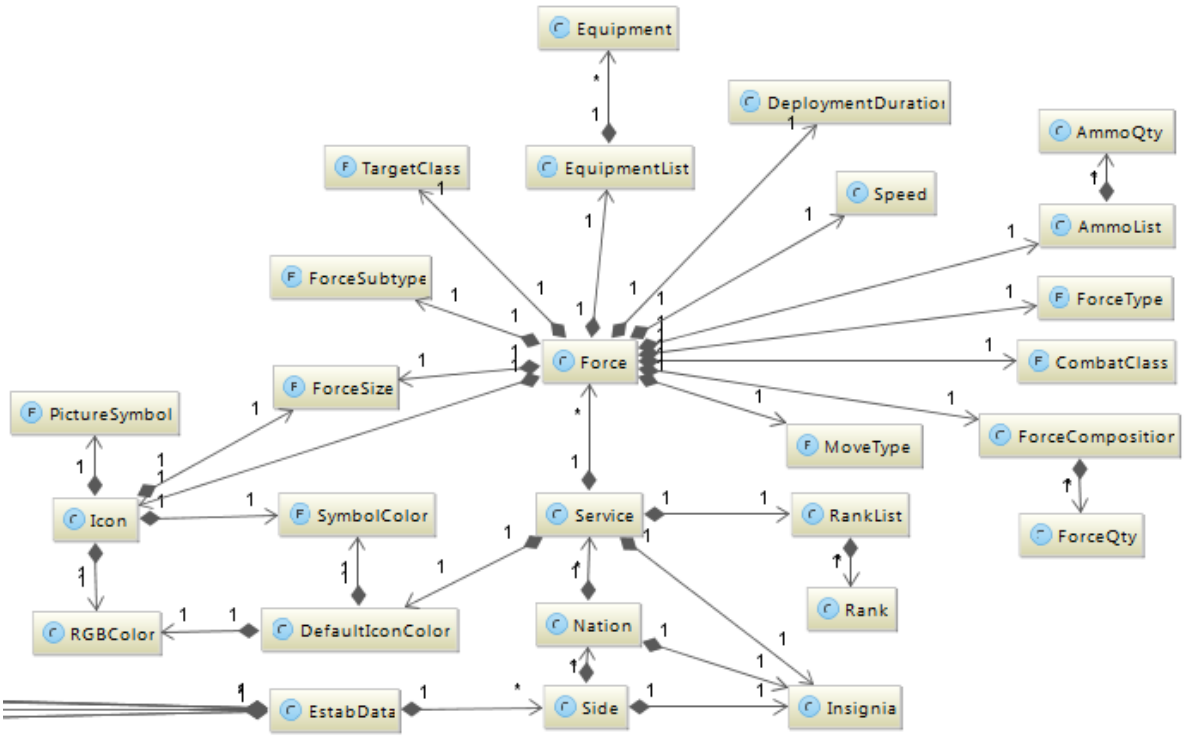

Figura 6. Diagrama completo de clases. Parte 2.

3.3 Diseño del Modelo

La idea del modelo es que se centre en la lógica referente a la administración de objetos de elementos. Para ello hace uso de los adaptadores que le permiten acceder a los elementos de las clases de persistencia y del sistema de referencias que mantiene las relaciones entre los objetos.

Por la forma en la que está diseñado el *estab* se concluye que existe una restricción de integridad que exige identificadores únicos, tanto entre elementos del mismo tipo como de diferentes tipos.

Como se explica en la sección anterior en el perfil de rendimiento de un arma, el objeto `AmmoLoad` guarda el identificador de la munición utilizada. No existe ningún otro campo que asegure que el identificador sea de tipo munición. Por lo tanto, se entiende que este valor no puede ser igual al de otro identificador de un elemento diferente a munición. Si ocurriese esta situación se llegaría a un estado de inconsistencia.

Dado que no existe un organismo central o herramienta global para monitorizar los identificadores se hace necesario disponer de algunas medidas para asegurar su unicidad.

En primer lugar, se ha de mantener un contador de identificadores de manera que se pueda asegurar que a un nuevo elemento no se le asigne un identificador repetido. En segundo lugar, hay que tomar las medidas necesarias si el identificador cambia o si la relación ya no existe.

Por esta razón se ha creado un sistema de referencias, el cual gestiona las relaciones que faltan entre identificadores y los objetos que identifican.

En conclusión, el modelo maneja los elementos del dominio gracias a la ayuda de los adaptadores y el control de referencias. La figura 7 muestra las relaciones de un elemento que el modelo necesita para trabajar con él.

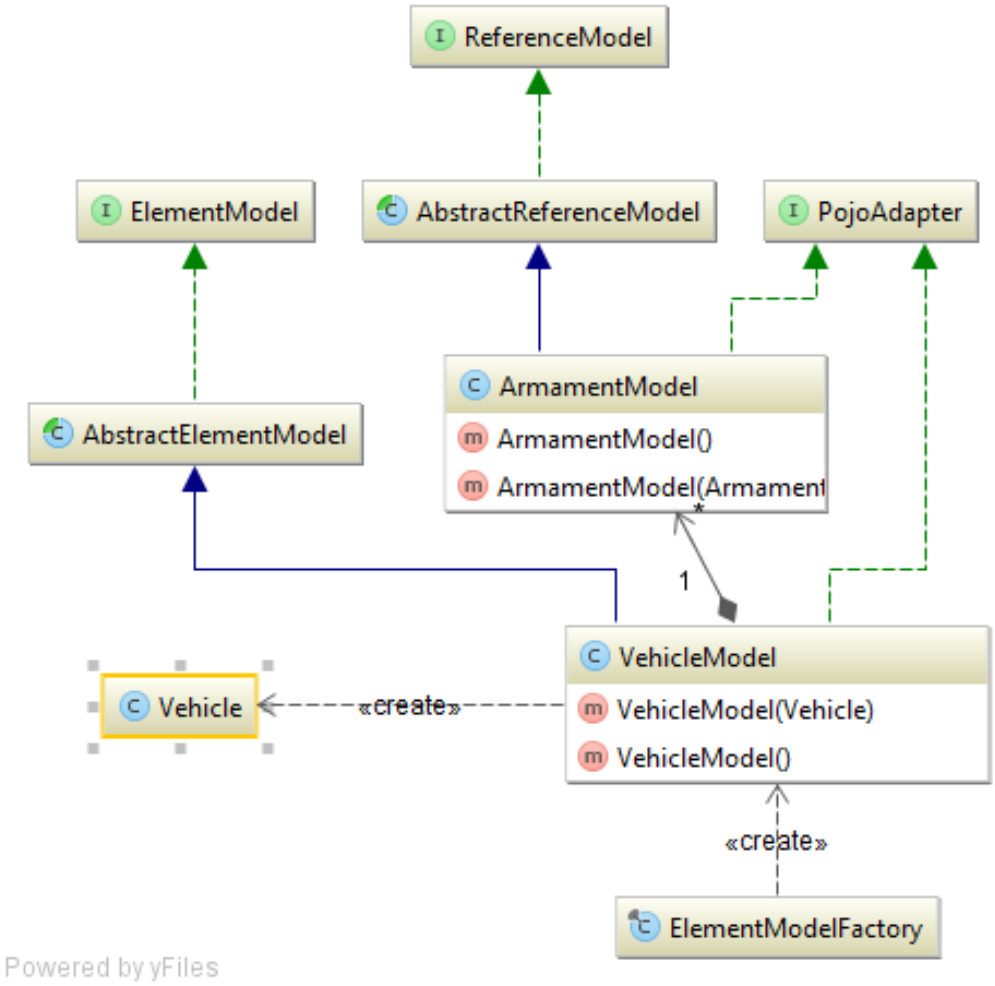

Figura 7. Relaciones de un elemento del estab *en la capa de modelo.*

3.4 Diseño del Controladores

La capa de controlador en realidad contiene a diversos controladores, los cuales se dividen en tres categorías:

- Controlador global: Encargado de la vista principal (es decir, de todo lo que el usuario ve de la aplicación al abrirla) y de realizar operaciones sobre el modelo de cada *estab* como puede ser por ejemplo copiar una serie de elementos desde un *estab* al otro.

- Controlador de *estab*: Maneja la vista de un único *estab*. Además se encarga de cargar los modelos de elementos

(objetos de adaptadores) en la lista y de abrir el editor de elemento correspondiente.

- Controlador de editor de elemento: Este controlador es el encargado de implementar el patrón Observador. Para ello enlaza las propiedades que proporciona el modelo del elemento a las propiedades de la vista del editor correspondiente.

La relación estre los diferentes controladores se puede ver con el diagrama siemple de la Figura 8:

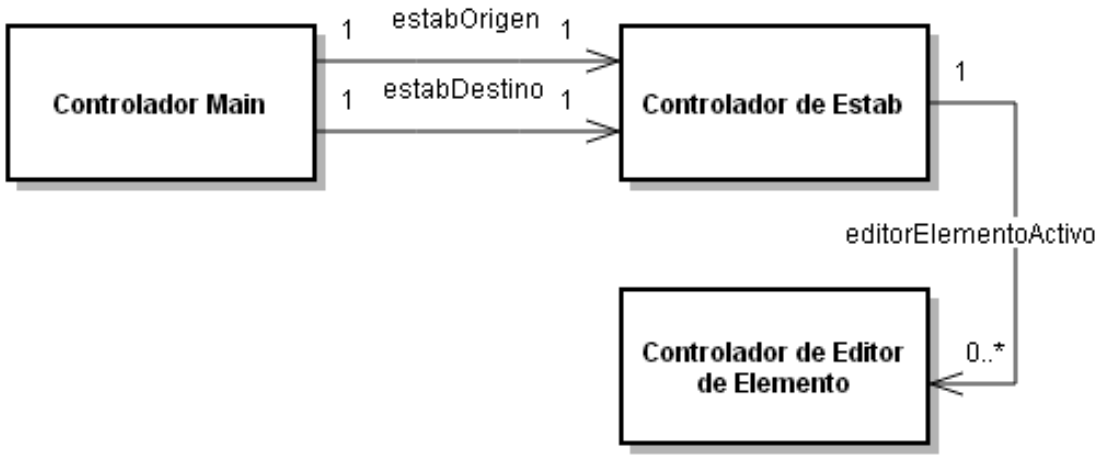

Figura 8. Relación entre controladores.

3.5 Diseño de la interfaz gráfica de usuario

Puesto que el objetivo es crear una aplicación de escritorio uno de los puntos más importantes a resolver es la elección de un *framework* para desarrollar la interfaz gráfica. Lo ideal es que la alternativa escogida disponga de todas las características necesarias sin añadir complejidad innecesaria al desarrollo.

Las opciones consideradas son aquellas con las cuales ya se ha trabajado previamente en otros proyectos. En este caso: Qt (C++), PyQt (Python) y Swing/JavaFX (Java).

Todas comparten la propiedad de ser multiplataforma, el cual es un requisito indispensable dado la necesidad de desarrollar la aplicación en diferentes sistemas operativos.

Finalmente se redujeron las opciones a las que están basadas en el lenguaje **Java** por las siguientes razones:

- Las clases iniciales sobre las que se basó este proyecto estaban programadas en Java. Usar Qt o PyQt supondría portar el código a otro lenguaje.

- Java es un lenguaje con el que el equipo ha trabajado mucho, lo cual permite centrarse más en el desarrollo de la aplicación y reducir el tiempo dedicado al estudio del lenguaje.

- Existe mucha documentación de los *frameworks* Swing y JavaFX, lo que permite agilizar el desarrollo del proyecto. Sin embargo, PyQt tiene una comunidad pequeña y no dispone de tanta documentación en comparación.

3.5.1 Elección del entorno gráfico: JavaFX

Existen varias plataformas para desarrollar IGUs en Java, las más comunes: AWT, Swing y JavaFX.

AWT (*Abstract Window Toolkit*) es la más antigua y la primera en ser incorporada a la API estándar de Java. Realmente es una interfaz al sistema nativo dependiente del SO, el comportamiento de los componentes varía según la máquina en la que se ejecuten.

Swing está basado en AWT mejorando en gran medida el desarrollo de interfaces. Además de añadir más componentes, más atractivos y mejor diseñados, estos componentes son controlados por la propia plataforma y son independientes del SO. Aunque es posible mezclar AWT y Swing no suele ser recomendable.

JavaFX rompe con los esquemas de las dos anteriores. Está diseñada para crear aplicaciones ligeras con aceleración hardware de manera sencilla, totalmente independiente del SO y con fácil portabilidad.

Al final se opta por utilizar JavaFX debido a:

- AWT está obsoleto y prácticamente sustituido por Swing. Así mismo, Swing será reemplazado por JavaFX [8].

- Aunque Swing sea un avance respecto de AWT, los ejecutables generados son más pesados que JavaFX.

- Swing es menos consistente a la hora de diseñar los componentes en la interfaz, los *layouts* suelen dar problemas y no es tan rico en características como JavaFX.

- JavaFX es mucho más moderno y sigue prácticas recomendadas, como la separación entre vista y controlador siguiendo el patrón MVC, o el uso de hojas de estilo CSS separadas de la vista.

- Scene Builder, el editor de interfaces de JavaFX, es simple e intuitivo y facilita la edición de archivos FXML, que es el lenguaje usado por JavaFX para definir interfaces.

- Puede portarse fácilmente a otras plataformas como la web y dispositivos móviles, en caso de ser necesario.

3.5.2 Interfaz principal

Se compone de tres áreas (Figura 10):

1. Barra de menús:
   - Opciones para los paneles de *estab* origen y *estab* destino: abrir, cerrar y guardar.
   - Edición de elementos: copiar, duplicar y eliminar.
   - Creación de elementos (vehículos, armas, fuerzas, servicios, etc.).
   - Opciones de interfaz: visibilidad de componentes y geometría.
   - Ayuda y créditos.

2. Barra de herramientas: Acceso rápido a operaciones como abrir o guardar *estabs*, elegir entre modo de edición simple o en paralelo, y creación y comparación elementos.

3. Paneles de *estabs*: Esta sección ocupa la mayor parte de la interfaz. De hecho se ha optado por mostrar las imágenes en

edición simple ya que las imágenes de edición en paralelo sobre pasan el ancho de la memoria.
Es en esta sección donde se cargan los editores y listas de elementos. Cada tipo de elemento dispone de un editor con el que leer y modificar su información. Las listas sirven para poder escoger el elemento con el que operar.

Ambos paneles son prácticamente iguales excepto por algunas diferencias:

3.1 Panel *estab* **origen**: los editores se abren en modo lectura y la única operación disponible sobre elementos es la de copiar. Se asegura que ningún *estab* pueda ser modificado desde este panel.

3.2 Panel *estab* **destino**: los editores se abren en modo escritura. Se permite duplicar y eliminar los elementos existentes además de pegar los copiados desde el estab origen.

3.5.3 Sección de estabs

La interfaz de los *estabs* se comparte tanto para el de origen como para el de destino con pequeñas diferencias. Esta interfaz se puede dividir en cuatro partes (Figura 9):

1. Estadísticas y características del archivo. En esta barra se muestra el modo en el que está abierto el *estab* con icono de un ojo para el *estab* origen y un lápiz para el *estab* destino. Además, la barra contiene un campo para cada tipo de elemento indicando el número de ocurrencias.

2. Componentes de gestión sobre la lista de elementos: botones para filtrar por tipo de elemento, barra de búsqueda para filtrar por nombre, botones para seleccionar determinados elementos en la lista, botones para eliminar y copiar/duplicar elementos seleccionados.

   El botón de eliminar está oculto en el *estab* origen para prevenir la pérdida de datos. En el *estab* destino el botón de copiar actúa como botón de duplicar, que significa que se creará una nueva copia de los elementos seleccionados en el mismo *estab* destino.

3. La lista de elementos. Cada elemento en la lista dispone de un checkbox con el que marcarlo para eliminar, copiar o duplicar. En el caso de las fuerzas la lista es de tipo árbol donde se puede ver toda la jerarquía de elementos complejos, la selección en este caso se hace en cascada (seleccionar un servicio automáticamente seleccionará todas sus fuerzas).

4. Sección de editor de elementos. Una vez seleccionado un elemento en la lista el programa cargará el editor correspondiente en la sección de editores. En el caso de que se seleccione otro elemento del mismo tipo, el editor permanecerá abierto y mostrará la información del nuevo elemento. Si por el contrario el nuevo elemento seleccionado es de tipo diferente al ya abierto, se cerrará el editor del tipo anterior y se cargará el del nuevo tipo.

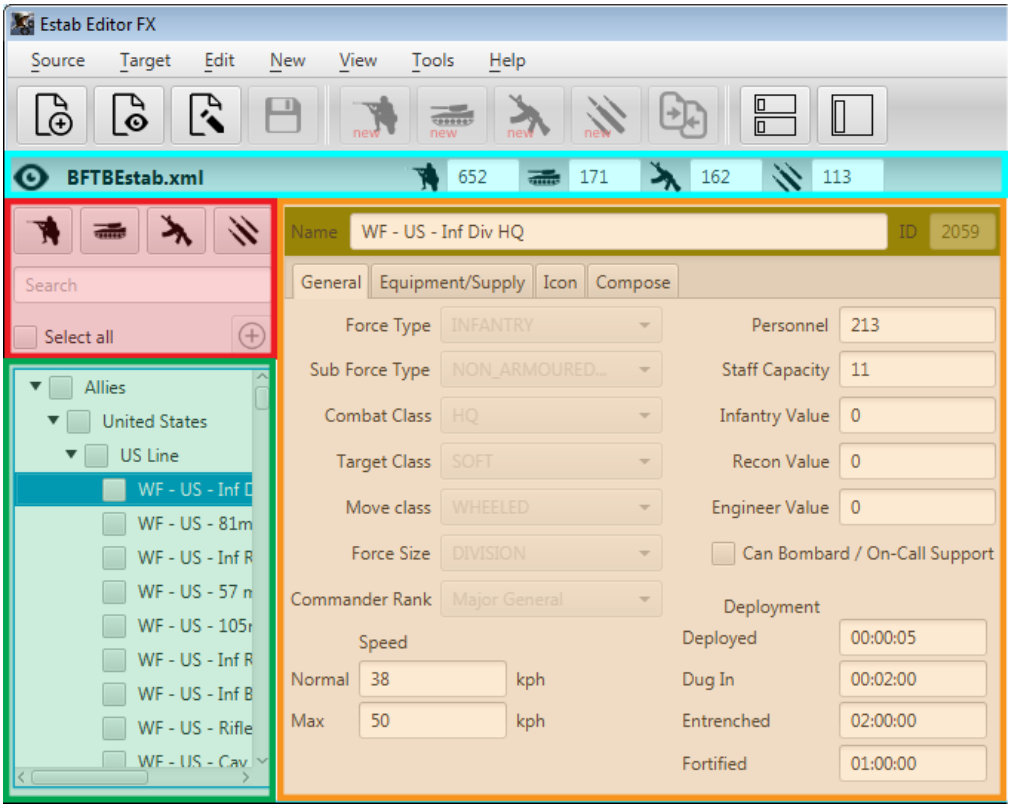

Figura 9. Partes de la interfaz de estab. Estadísticas del archivo en azul, lista de elementos en verde, filtros de la lista en rojo y sección de editores en naranja.

Figura 10. Partes de la interfaz principal resaltadas en colores: en rojo la barra de menús, en verde la barra de herramientas y en naranja la sección de estabs.

3.5.4 Editores de elementos

La aplicación está diseñada de manera que cada elemento tiene un editor específico en el que se muestran y editan sus datos. Estos editores son cargados dinámicamente en la sección de editores dependiendo del tipo del elemento seleccionado en la lista.

Los editores muestran detalladamente la información de cada elemento. Por lo general los valores están representados en varios componentes gráficos como desplegables, tablas, casillas de selección o áreas de texto. Cada componente está etiquetado con el nombre del atributo que le identifica.

Debido a la gran cantidad de datos, los editores contienen varias pestañas con el objetivo de no sobrecargar visualmente al usuario.

(i) Editor de munición

El editor de munición es el más simple del conjunto y únicamente contiene una descripción, la cantidad de cargas y el peso total.

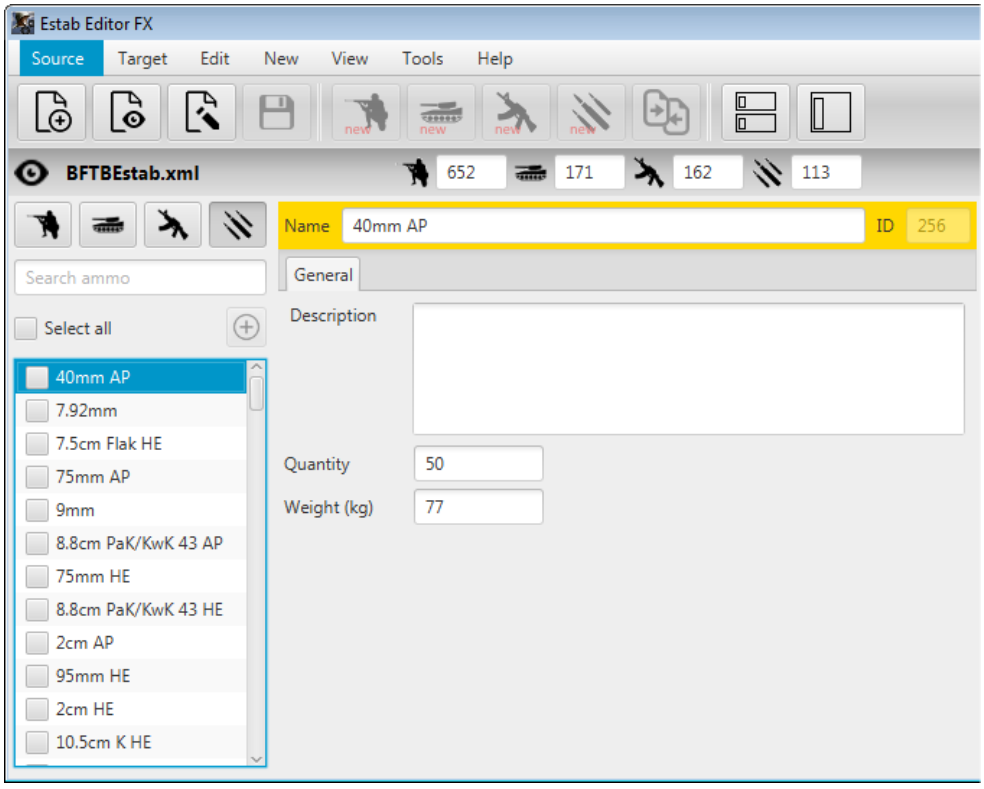

Figura 11. Pestaña general del editor de munición.

(ii) Editor de armas

El editor de armas se abre en la sección de editores del estab seleccionando un arma en la lista de elementos. Contiene las pestañas:

- General (Figura 12): En esta pestaña se muestran la información general de las armas. Se compone de los siguientes campos: descripción con una breve explicación del arma, una imagen opcionalmente asignada por el usuario, rol y tipo del arma.

- Performance (Figura 13): Es posible crear una serie de perfiles en cada arma en dependencia del tipo de combate para la que esté destinada. El perfil permite definir el peso de arma, el calibre, la velocidad de disparos y también asignar valores de puntería y penetración en función del rango al que se encuentre el objetivo.

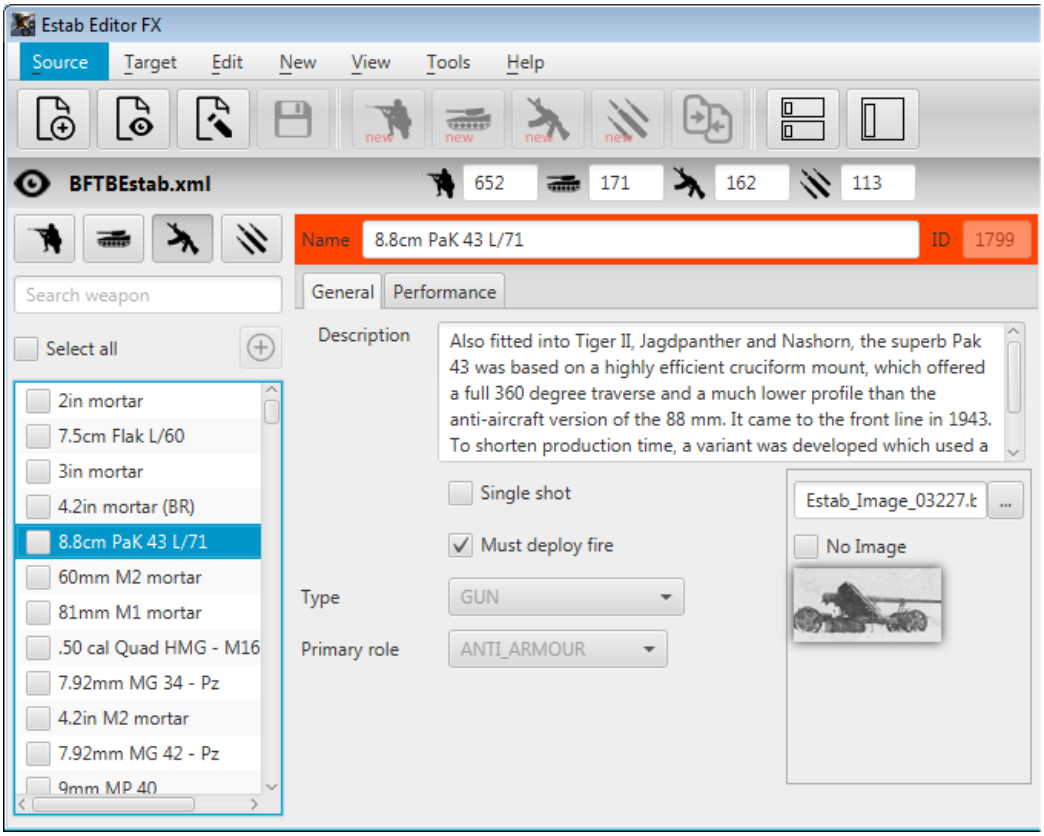

Figura 12. Pestaña general del editor de armas.

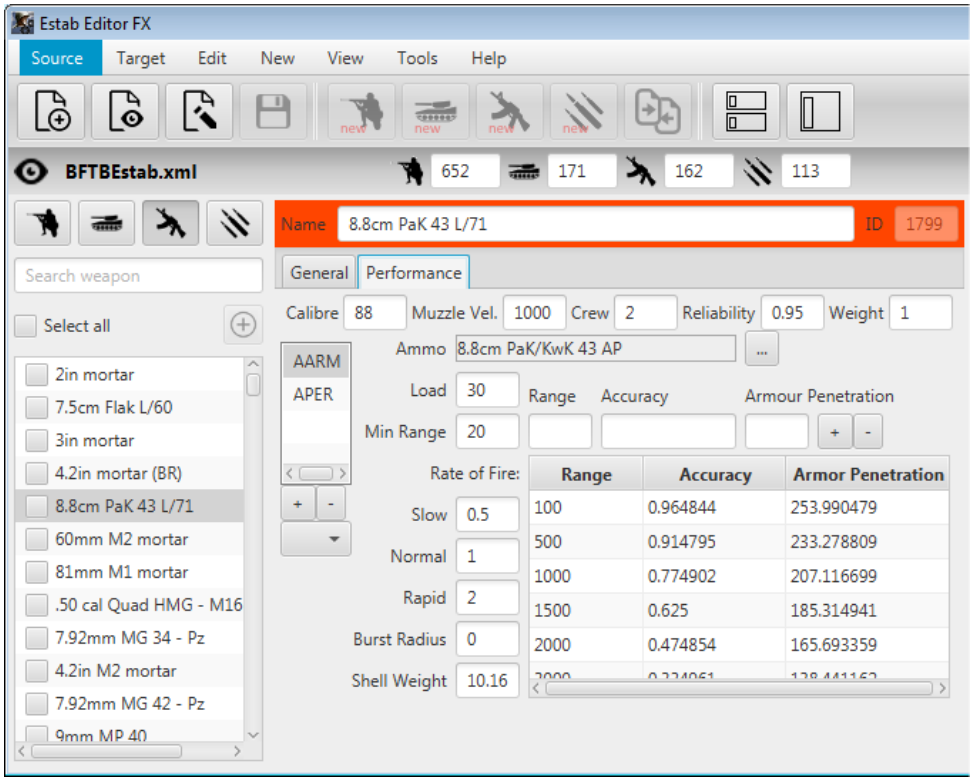

Figura 13. Pestaña rendimiento del editor de armas.

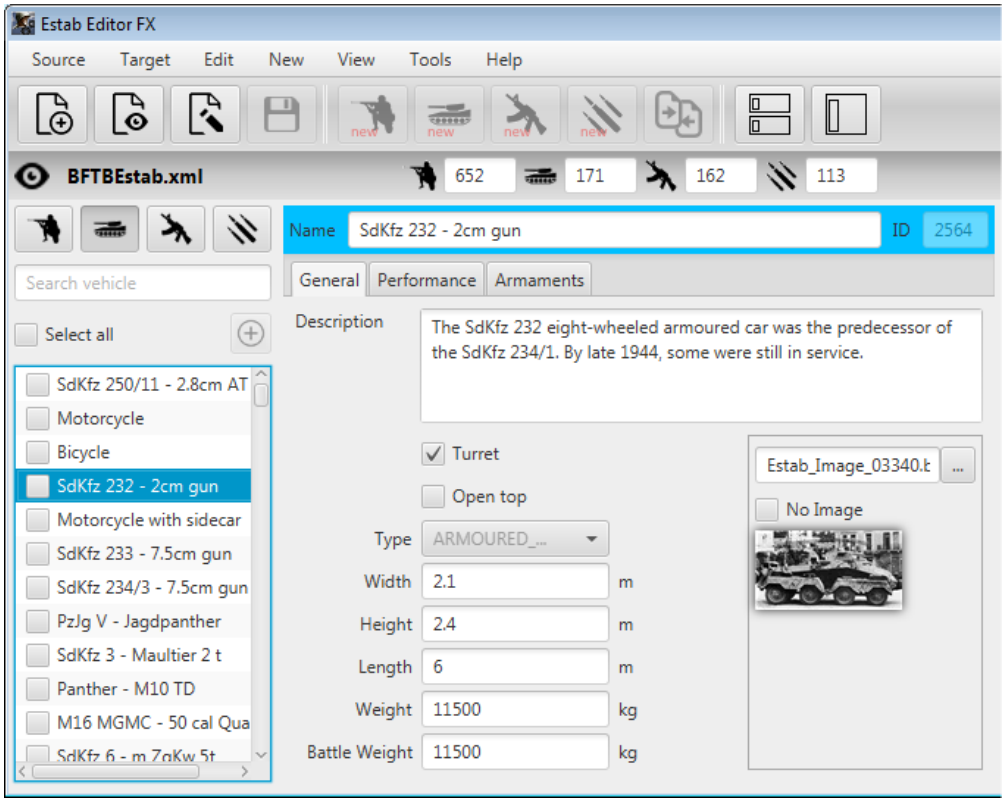

Figura 14. Pestaña general del editor de vehículos.

(iii) Editor de vehículos

Seleccionando un vehículo en la lista de elementos se abrirá el editor de vehículos con las siguientes pestañas:

- General (Figura 14): Contiene información básica de los vehículos con una pequeña descripción, las medidas geométricas, el peso y el tipo.

- Rendimiento (Figura 15): El rendimiento del vehículo es determinado por el tamaño del equipo que transporta, la capacidad del depósito de gasolina, la velocidad media y máxima, etc.

- Armamento (Figura 16): Un vehículo puede transportar un arsenal de armas para las unidades en combate. En esta pestaña se puede editar la lista con las armas deseadas.

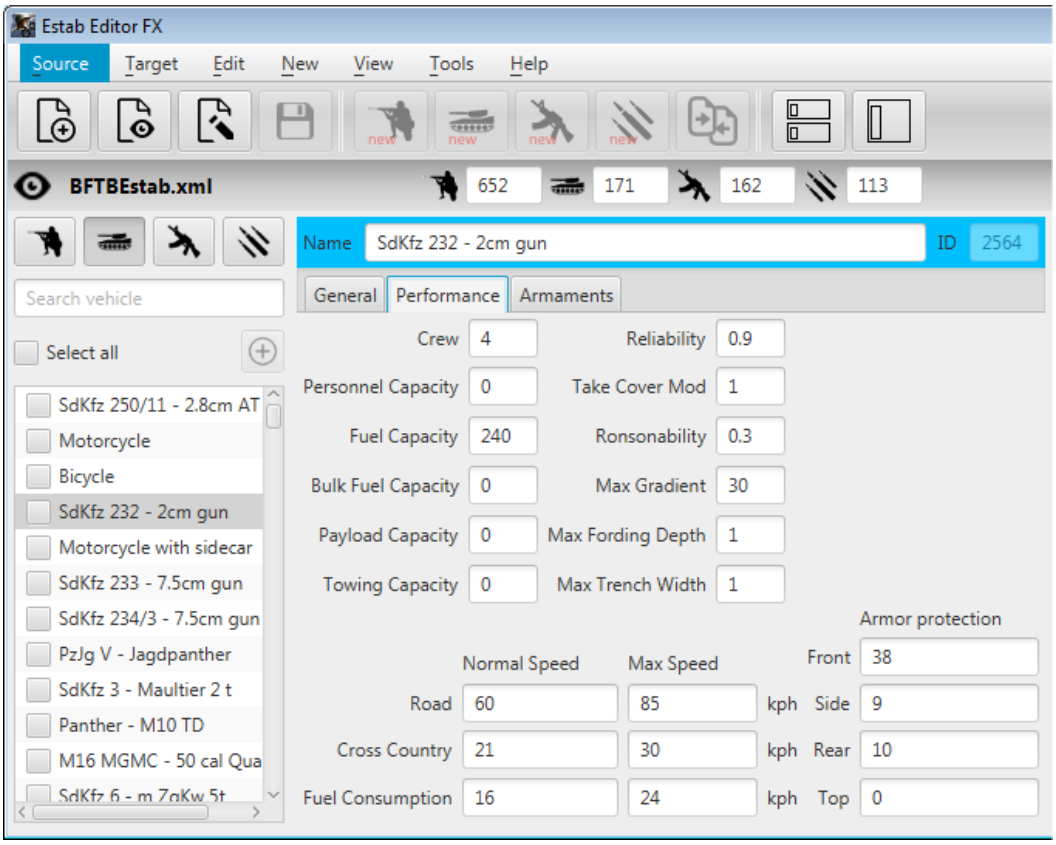

Figura 15. Pestaña rendimiento del editor de vehículos.

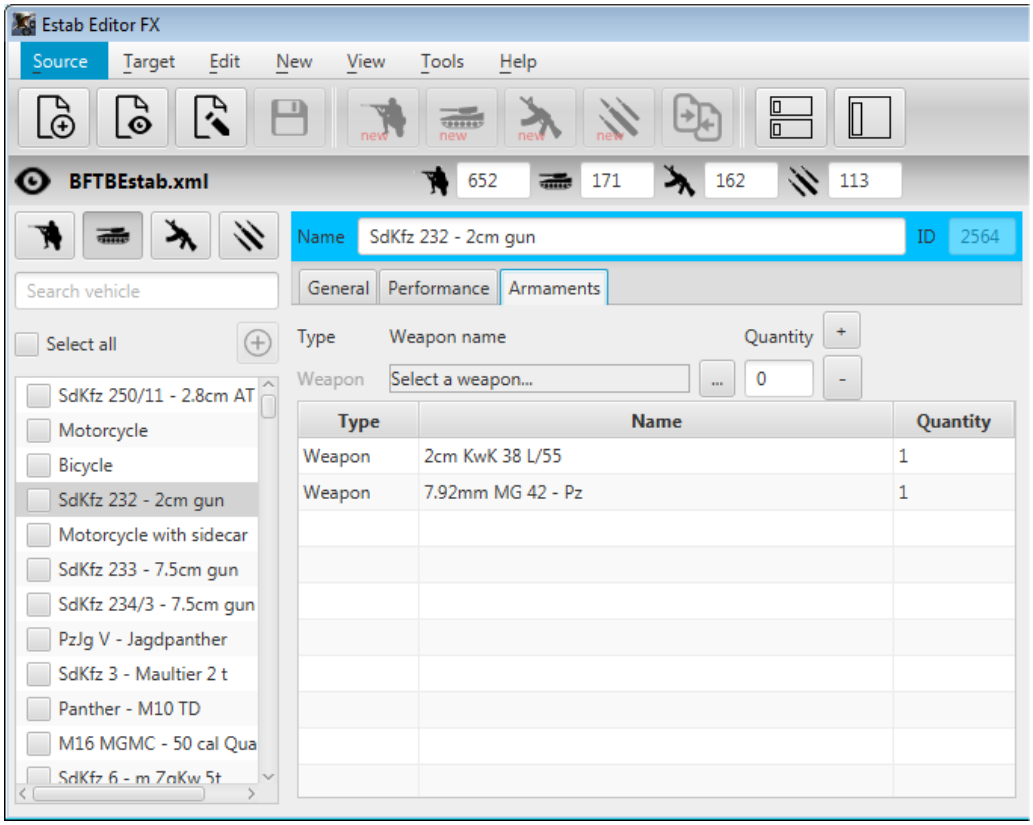

Figura 16. Pestaña armamento del editor de vehículos.

(iv) Editor de fuerzas

El editor de fuerzas se abrirá en la sección de editores al seleccionar una fuerza en la lista de elementos. Tiene las siguientes pestañas:

- General (Figura 17): En esta pestaña se muestra información general sobre las fuerzas. Los campos que se incluyen son la clase de combate, el tipo de fuerza (infantería, artillería, blindado o logística), la velocidad y el tipo de movimiento (por aire, agua o tierra, en vehículos o a pie, etc.), el tamaño (si se considera platón, regimiento, división…), el rango de la persona a cargo, el equipo de trabajo repartido en las diferentes clases de personal (ingeniero, infantería, reconocimiento…), el tiempo necesario para desplegarse (lo que tarda en atrincherarse, fortificarse, etc.).

- Equipamiento (Figura 18): En esta pestaña se gestionan los suministros de armas de cada fuerza. En la tabla se ven las armas seleccionadas y la cantidad deseada correspondiente.

- Icono (Figura 19): Opciones para el tipo de imagen que identifica la fuerza y los colores deseados para pintarla.

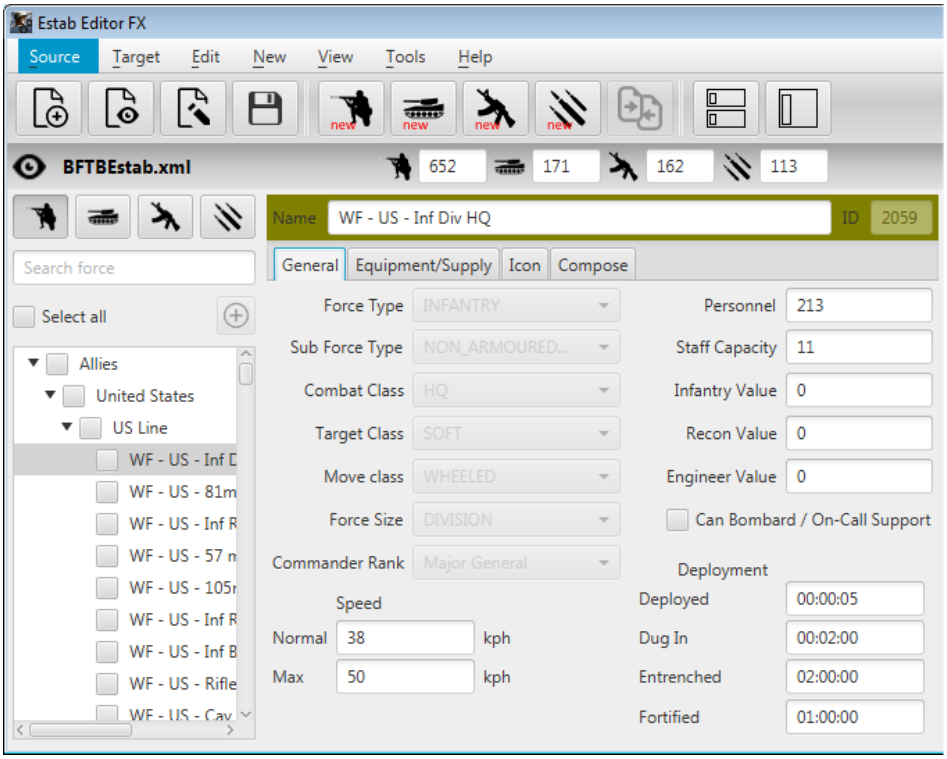

Figura 17. Pestaña general del editor de fuerzas.

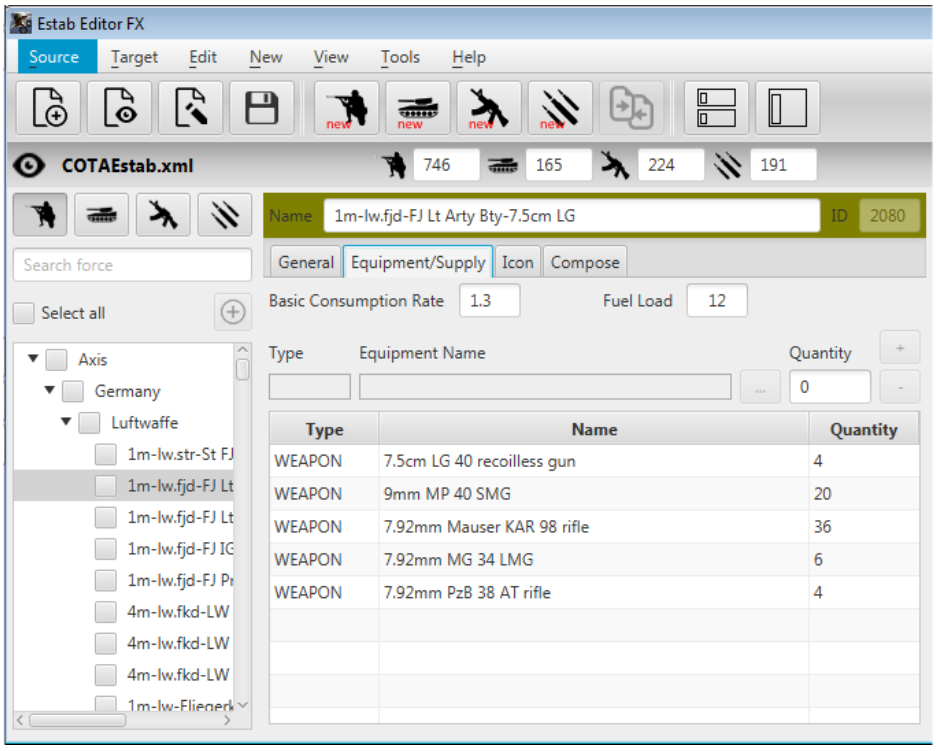

Figura 18. Pestaña de equipamiento del editor de fuerzas.

- Composición (Figura 20): Como ya se ha comentado en el apartado del modelo de dominio, una fuerza puede contener a su vez un conjunto de fuerzas. Por ejemplo, una compañía que esté formada por varios pelotones.
  Esta pestaña permite al usuario gestionar las composiciones seleccionando fuerzas ya existentes e incluyéndolas en la tabla de subfuerzas.

(v) Editor de servicios

Este editor se abrirá en la sección de editores al seleccionar un servicio en la lista de elementos. Se compone de dos pestañas:

- General (Figura 21): Se muestra una breve descripción del servicio y una lista para gestionar los rangos militares que pueden asumir las fuerzas dentro del servicio.

- Gráficos (Figura 22): Permite cambiar los colores y las insignias del servicio, así como aplicarle la misma configuración a las fuerzas que contiene.

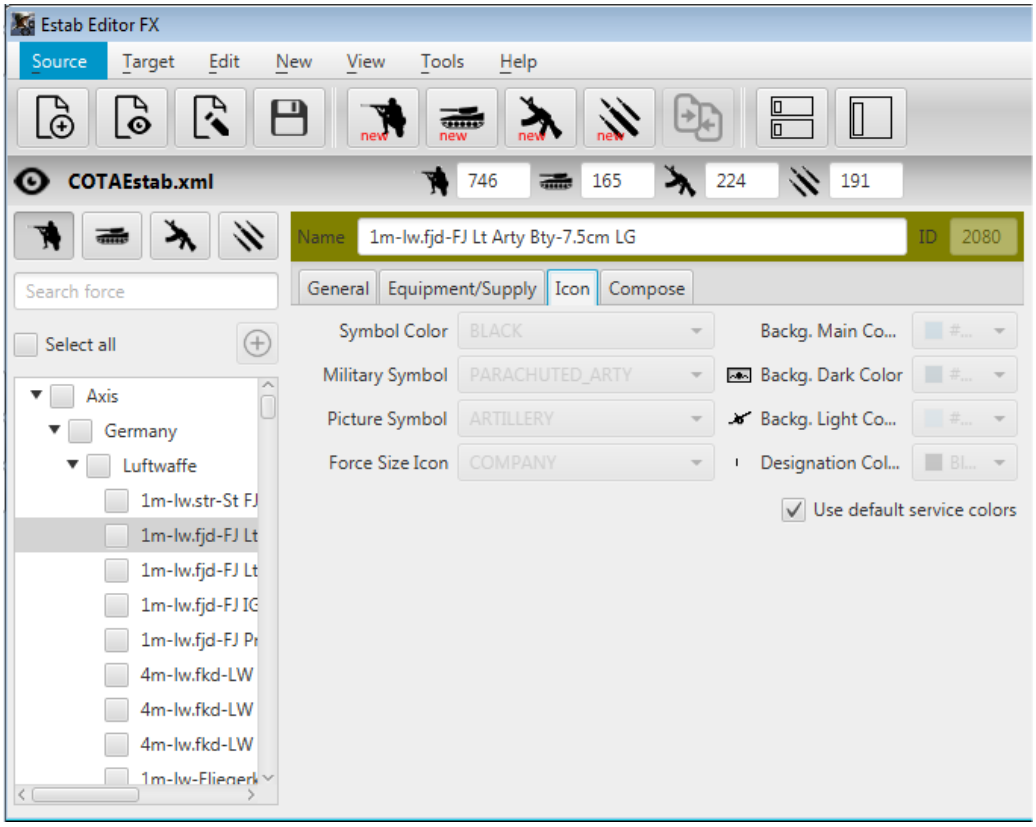

Figura 19. Pestaña de icono del editor de fuerzas.

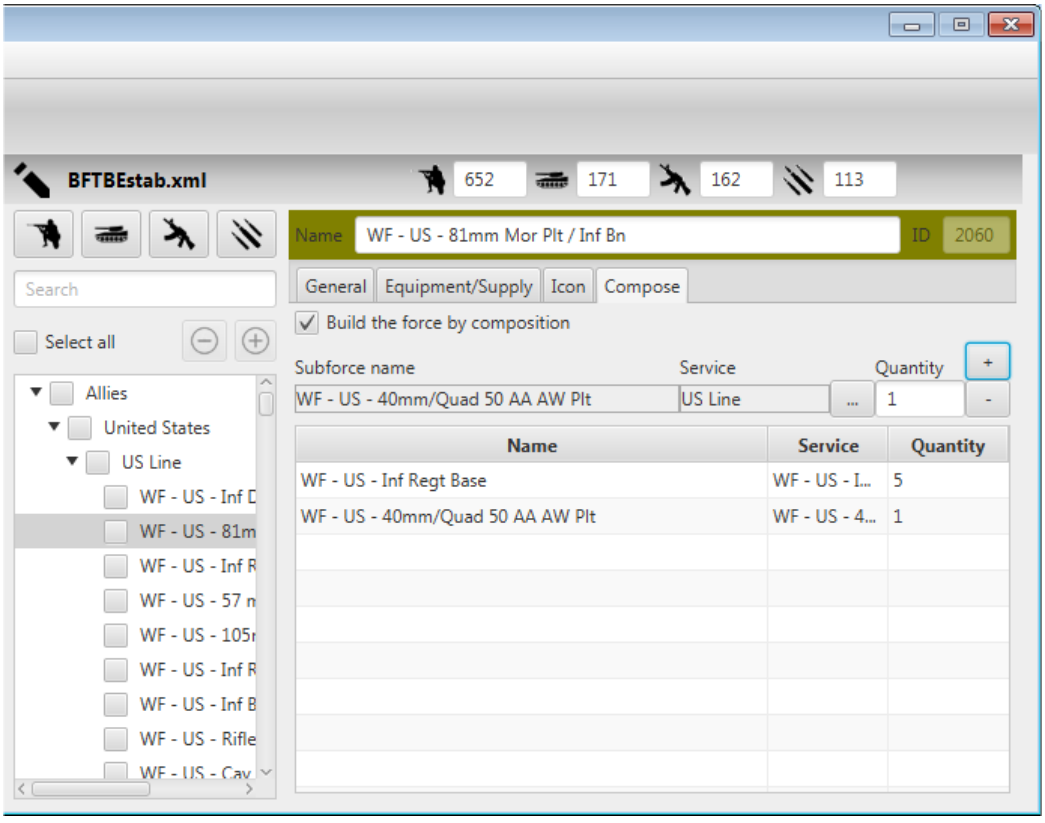

Figura 20. Pestaña de composición del editor de fuerzas.

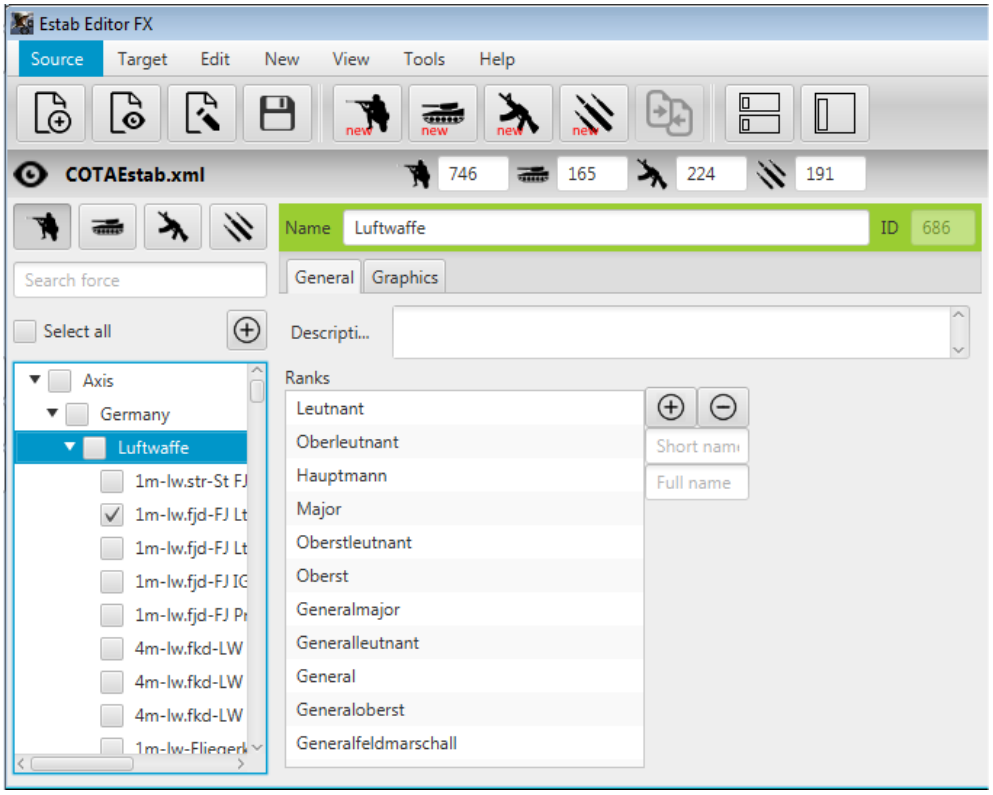

Figura 21. Pestaña general del editor de servicios.

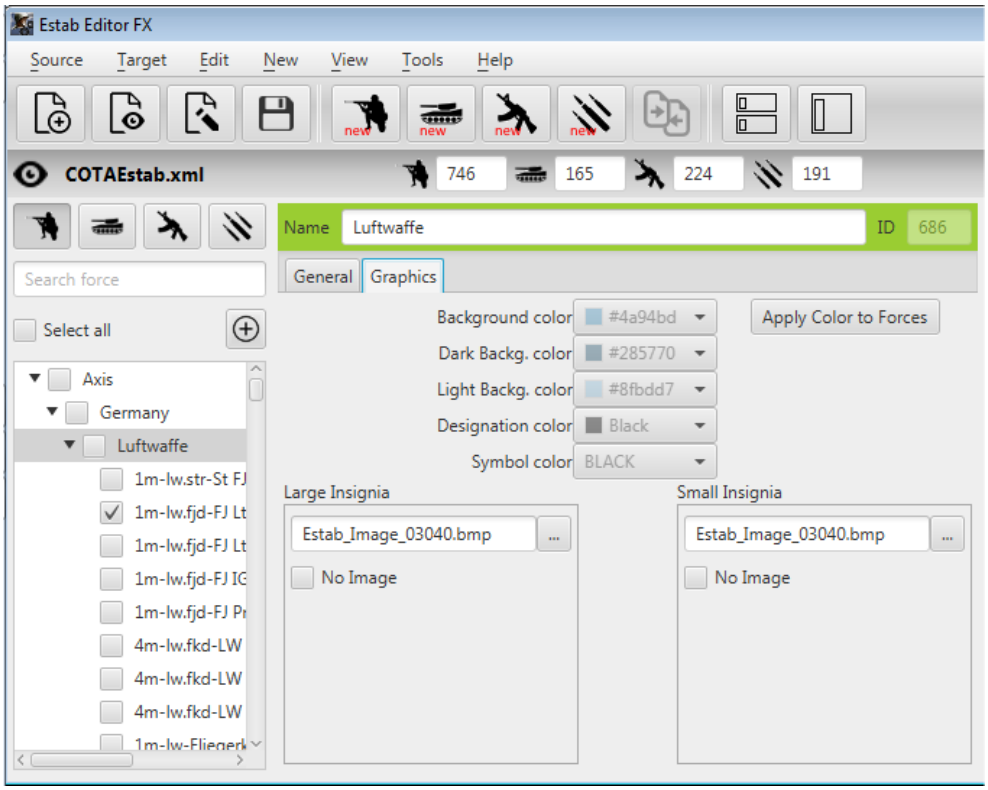

Figura 22. Pestaña de gráficos del editor de servicios.

(vi) Editor de naciones

En este editor se modifica la información referente a las diferentes naciones. Como se puede observar en la Figura 19 las naciones, al contrario que el resto de elementos, son bastante simples y carecen de gran contenido editable. Solo se puede cambiar la descripción y las insignias pequeña y grande.

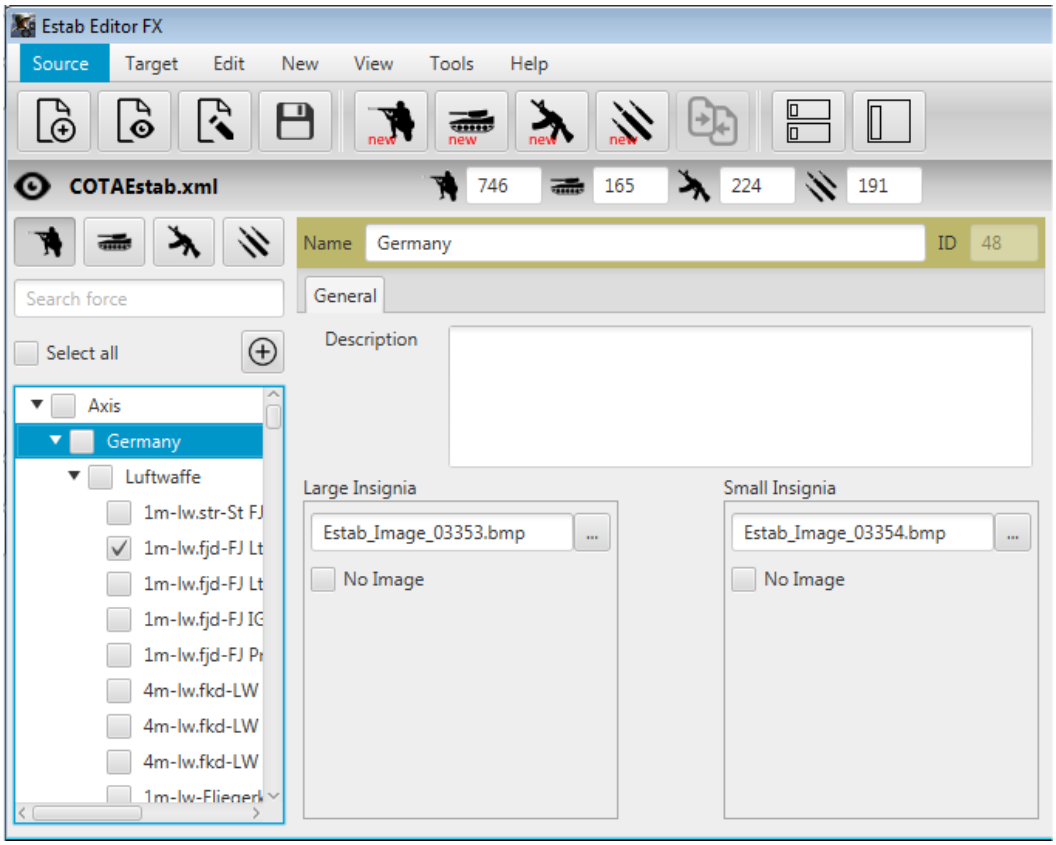

Figura 23. Pestaña general del editor de naciones.

(vii) Editor de bandos

En esta pestaña se incluyen el conjunto de herramientas necesarias para modificar los diferentes campos que forman un bando. De la misma manera que en el caso de naciones, estas entidades disponen de pocos elementos editables. En concreto estos campos son: la descripción del bando, el ratio de consumo básico.

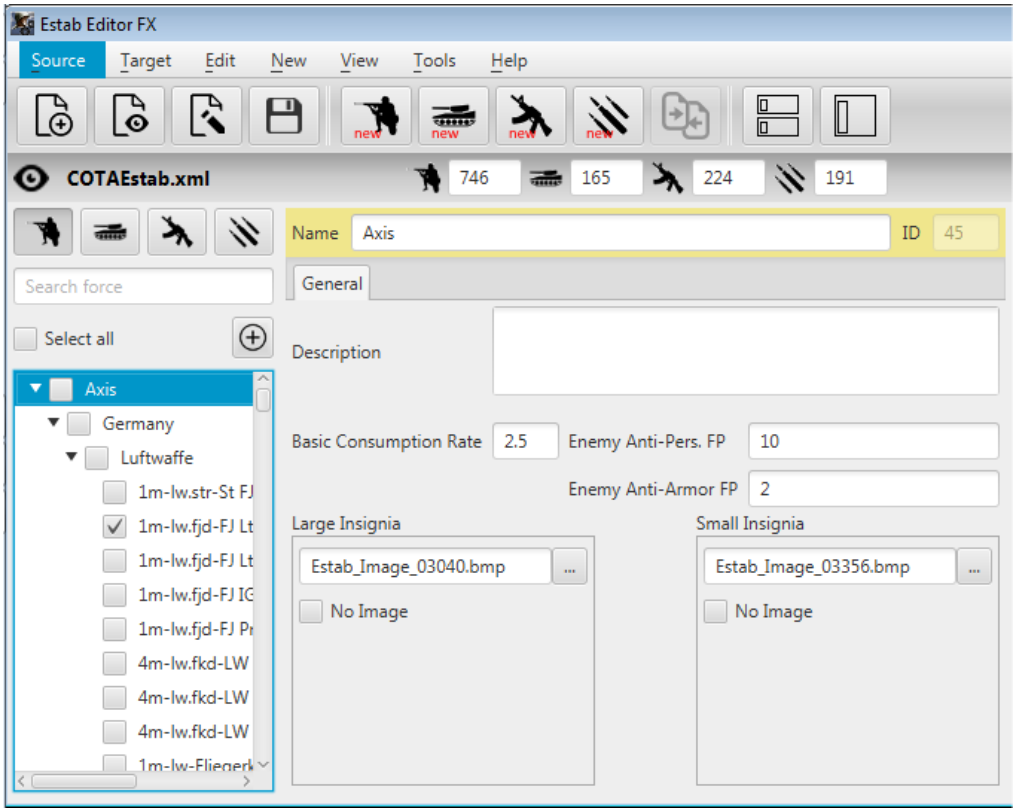

Figura 24.Pestaña general del editor de bandos.

Implementación y pruebas

Ahora que ya se han visto los conceptos básicos sobre los que se fundamenta la aplicación como el patrón de implementación, las propiedades y enlaces, y la serialización, este capítulo explica con más detalle el desarrollo y la organización interna del código.

4.1 Entorno de trabajo

El desarrollo del proyecto ha estado muy influenciado por tres herramientas que han facilitado la creación y organización del código.

4.1.1 IntelliJ IDEA

La elección del IDE suele ser una decisión personal. Hoy en día todos ofrecen tantas funcionalidades además de las básicas que la razón de elegir uno en concreto tiene es más subjetivo que objetivo.

De entre los IDEs más comunes, como Eclipse o Netbeans, es IDEA el que más ha destacado con su sistema de navegación avanzado, la increíble ayuda al generar código automáticamente de forma inteligente, la perfecta integración de otras herramientas como JAXB o el sistema de control de versiones nativo y externo (Git, Mercury, SVN, etc.).

Además, IDEA tiene soporte para JavaFX y Scene Builder, lo cual permite que operaciones como refactorizar se puedan automatizar reduciendo el esfuerzo y el posible error humano.

4.1.2 Gradle

Gradle es una herramienta de automatización para la compilación de aplicaciones. Da la posibilidad de definir secuencias de operaciones para la limpieza, compilación y depuración de código. EN cierto modo se podría comparar con el `Makefile` de un proyecto en C++.

Sin embargo, la razón primordial por la que se usa esta herramienta es porque permite la gestión de librerías externas, resolviendo cualquier dependencia sin necesidad de controlarlas manualmente.

4.1.3 Git

Los sistemas de control de versiones (VCS en inglés) facilitan en gran medida el desarrollo de un código. La idea principal es tener diferentes versiones del mismo proyecto según la funcionalidad que se está implementando y la evolución en el tiempo.

Lo que se pretende es aislar el desarrollo de un módulo de código sin afectar a la totalidad del proyecto. Una vez se ha obtenido el resultado esperado se aceptan los cambios realizados, en caso contrario se descartan.

A diferencia de otros sistemas, Git no funciona con repositorios centrales sino con locales. Esto significa que no es necesario estar conectado a otra máquina para poder utilizarlo, una vez el código sea el deseado se procede a la operación de fusión que crea una nueva versión del código mezclando la nueva y la anterior. Posteriormente se puede compartir el histórico de cambios a un repositorio global en la nube como el de nuestro proyecto [13].

4.1.4 JUnit

Las pruebas unitarias consisten en demostrar que un módulo de código funciona de la manera deseada sin tener que involucrar al resto del proyecto. El objetivo es crear una serie de pruebas que la que el código debe superar en todos los estados del desarrollo para justificar que cumple con los requisitos propuestos.

La manera más cómoda de generar estas pruebas es con JUnit, que es un conjunto de bibliotecas para Java diseñadas con este motivo. Además de estar en el repositorio de Gradle, IDEA proporciona una interfaz nativa con la que trabajar con JUnit y crear casos de prueba sencillamente.

Aunque no se haya usado durante todo el desarrollo como el resto de de herramientas comentadas se le hace una especial mención por su gran utilidad.

4.2 Estructura del proyecto

La mayoría de las clases del proyecto se han dividido en tres paquetes principales cada uno representando a la capa en la que se ubica reflejando el patrón MVC:

- *data:* Clases de elementos del *estab* con los datos puros.
- *controller:* Clases que coordinan las capas de modelo y vista.
- *views:* Interfaces definidas de forma declarativa con FXML.
- *model:* Modelos de las clases de la capa de persistencia y lógica para gestionar elementos.

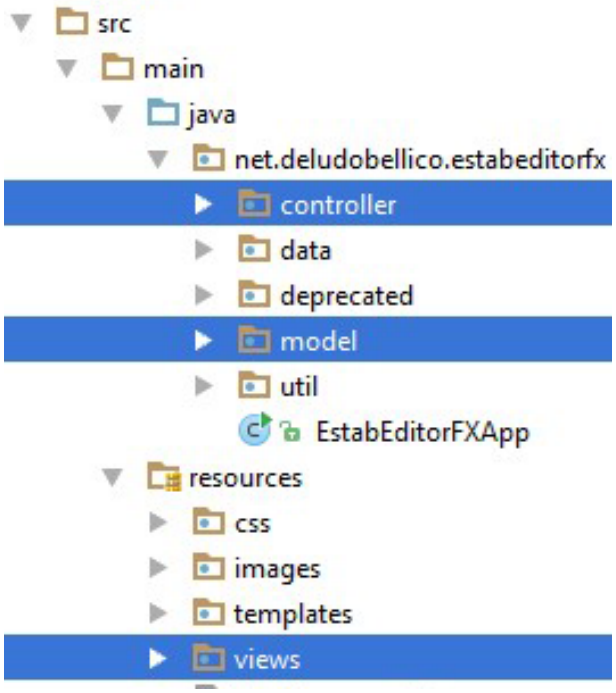

Figura 25. Estructura del proyecto.

4.3 Controladores

Existen tres tipos diferentes de controlador: el principal, el de estab, y el de los editores.

El controlador principal proporciona la lógica necesaria para gestionar la interfaz principal y operar con *estabs,* lo cual incluye: abrir los ficheros en disco, guardarlos, cerrarlos, mantener el estado en el que se encuentran y modificar los elementos visuales en concordancia (por ejemplo, desactivar los paneles de *estabs* si no hay ningún archivo abierto).

El controlador de cada *estab* gestiona el estado de la interfaz de *estab* y comunica al modelo cualquier cambio de información realizado por el usuario. Es aquí donde se especifican las acciones sobre elementos seleccionados en la lista como puede ser cargar el editor correspondiente en la vista o avisar al modelo para crear, copiar o eliminar determinados elementos.

Por último, los controladores de editores de elementos manejan el enlace entre las propiedades de la vista y las propiedades del modelo con los métodos `bindProperties()` y `unbindProperties()` como se puede ver en la

A continuación se muestra un extracto del controlador del editor de munición. Se utiliza el método `bindBidirectional()` para enlazar los campos de la interfaz (*`name`, `id`, `description`, `quantity`* y *`weight`)* con las propiedades del modelo del elemento (variable *`element`*).

```
@Override
public void bindProperties() {
    AmmoModel element = getActiveElement();
    name.textProperty()
            .bindBidirectional(element.nameProperty());
    id.textProperty()
            .bindBidirectional(element.idProperty(),
                                            NUMBER_STRING_CONVERTER);
    description.textProperty()
            .bindBidirectional(element.descriptionProperty());
    quantity.textProperty()
            .bindBidirectional(element.minOrderQtyProperty(),
                                            NUMBER_STRING_CONVERTER);
    weight.textProperty()
            .bindBidirectional(element.minOrderWeightProperty(),
                                            NUMBER_STRING_CONVERTER);
}
```

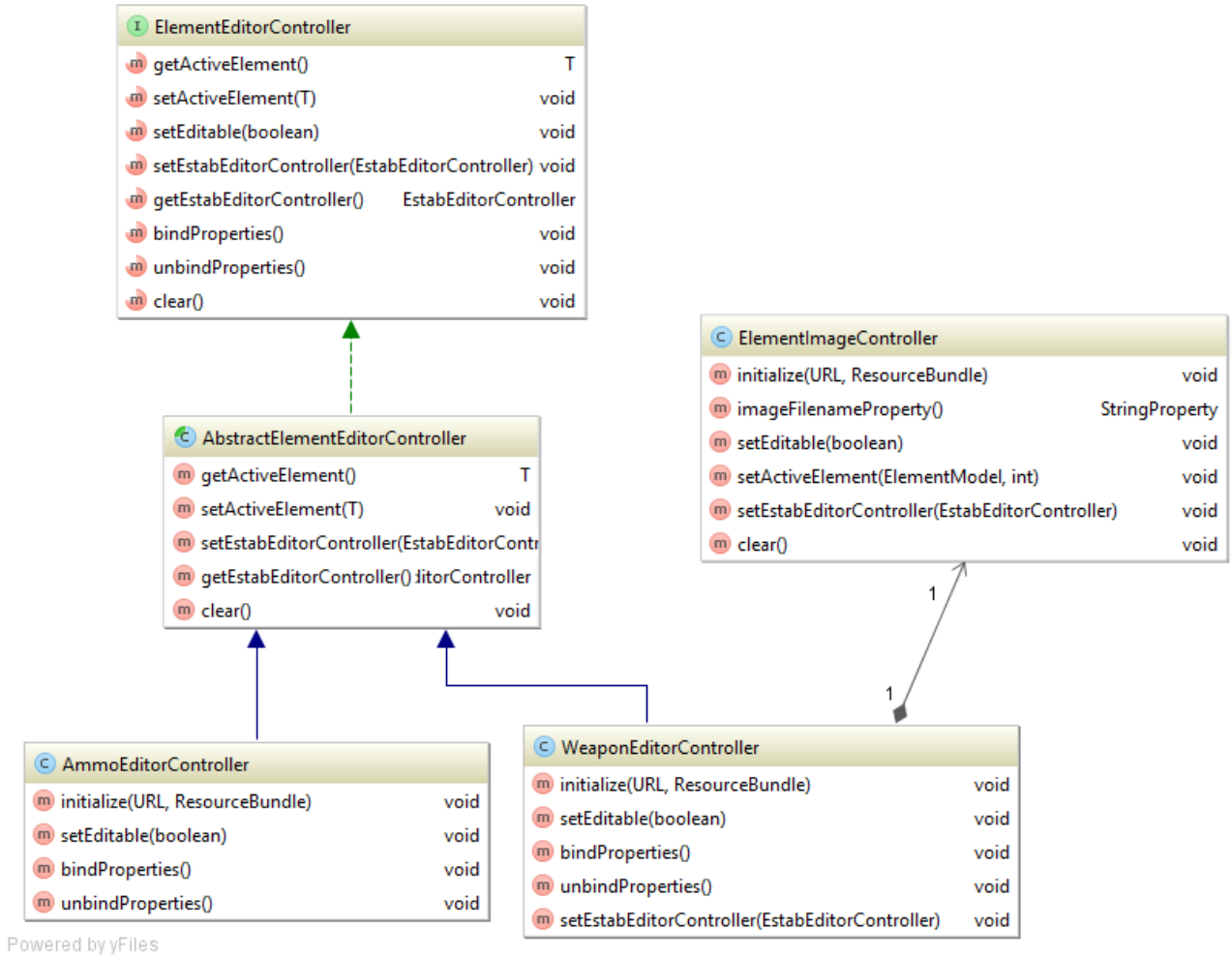

Figura 26. Controladores del editor de munición y de armas, este último asociado al controlador de imágenes.

Hemos enlazado las propiedades de forma bidireccional. Con esto se asegura que cualquier cambio realizado en la interfaz sea automáticamente modificado en el modelo y si cambia el modelo del objeto se verá reflejado en la interfaz.

Dado que se utiliza la misma interfaz de editor para varios objetos del mismo tipo de elemento, es necesario deshacer los enlaces una vez se ha terminado de editar el elemento. Para ello se utiliza el método `unbindBidirectional()` en cada propiedad que hayamos hecho el *binding*.

Todos los controladores siguen el mismo patrón cada vez que se carga un elemento: deshacer los *bindings* si fuese necesario y crearlos para el nuevo elemento. Esto permite crear una clase abstracta para los editores con el siguiente método:

```
@Override
public void setActiveElement(T element) {
    if (activeElement != null) unbindProperties();
    activeElement = element;
    bindProperties();
}
```

4.4 Serialización con JAXB

Toda la información que el usuario genere o modifique en un *estab* tiene que poder ser cargada y además ser guardada de forma persistente. Dado que los *estab* son ficheros hemos de emplear un procedimiento que permita guardar instancias de las clases en estos ficheros en disco y otro procedimiento para realizar el paso inverso.

JAXB cumple con los requisitos que se exigen. Esta herramienta dispone de dos procesos para la serialización:

- *Unmarshall:* el cual carga la información de archivos XML en objetos POJOs de las clases correspondientes.
- *Marshall*: realiza el paso inverso de manera que almacena en disco la información del *estab*.

No obstante, antes de emplear estos dos procesos se deben generar previamente las clases correspondientes a cada tipo de elemento contenido en el XML. Aunque se podría hacer manualmente este paso es fácilmente automatizable con la herramienta XJC. Para ello creamos un archivo XSD (Anexo A) en el que definimos el esquema de los ficheros *estab*.

Así pues la preparación para la serialización se resume en:

1. Analizar los campos de los elementos y generar el esquema XSD.
2. Ejecutar XJC y pasarle el esquema XSD para crear las clases Java.

Y la propia serialización que se lleva a cabo con JAXB:

- Utilizar el proceso de *unmarshall* para cargar la información guardada en el archivo XML a objetos de las clases creadas con XJC.

- Utilizar el proceso de *Marshall* para guardar los objetos en el archivo XML.

Como ejemplo tomaremos la clase `Nation` (Nación). Se obtiene una nación del archivo *estab* y vemos que esta contiene los campos `name`, `description`, `nationality`, `large-insignia`, y `small-insignia` seguidos de una lista de sus servicios.

```
<nation id="1472">
  <name>Poland</name>
  <description>
  </description>
  <nationality>Polish</nationality>
  <large-insignia id="3585" />
  <small-insignia id="3586" />
  <service id="1475">
      <!—service data-->
  </service>
</nation>
```

A partir de estos datos se crea un tipo complejo en el esquema XSD que define una nación de la siguiente manera:

```
<xs:complexType name="Nation">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element name="nationality" type="xs:string"/>
        <xs:element name="large-insignia" type="Insignia"/>
        <xs:element name="small-insignia" type="Insignia"/>
        <xs:element name="service" type="Service" maxOccurs="unbounded"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
</xs:complexType>
```

Es ahora cuando utilizamos XJC para generar la clase Java con los atributos arriba definidos y sus métodos de acceso. La clase resultado se puede ver a continuación.

```java
public class Nation {
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected String description;
    @XmlElement(required = true)
    protected String nationality;
    @XmlElement(name = "large-insignia", required = true)
    protected Insignia largeInsignia;
    @XmlElement(name = "small-insignia", required = true)
    protected Insignia smallInsignia;
    protected List<Service> service;
    @XmlAttribute(name = "id", required = true)
    protected int id;

public String getName() {
    return name;
}

public void setName(String value) {
    this.name = value;
}

public String getDescription() {
    return description;
}

public void setDescription(String value) {
    this.description = value;
}

public String getNationality() {
    return nationality;
}
```

```
public void setNationality(String value) {
    this.nationality = value;
}

public Insignia getLargeInsignia() {
    return largeInsignia;
}

public void setLargeInsignia(Insignia value) {
    this.largeInsignia = value;
}

public Insignia getSmallInsignia() {
    return smallInsignia;
}

public void setSmallInsignia(Insignia value) {
    this.smallInsignia = value;
}

public List<Service> getService() {
    if (service == null) {
        service = new ArrayList<Service>();
    }
    return this.service;
}

public int getId() {
    return id;
}

public void setId(int value) {
    this.id = value;
}
```

En el momento de abrir un *estab* en la aplicación, JAXB generará una instancia de `Nation` por cada elemento nación que se encuentre en el archivo e inicializará los atributos con los campos correspondientes del archivo XML.

4.5 Capa de Persistencia y modelos

El objetivo de utilizar las propiedades es siempre mantener sincronizados la capa de presentación con la capa de datos automáticamente.

Como hemos comentado en el apartado de serialización se generan las clases de cada elemento del *estab* con XJC a partir del esquema XSD que se ha definido.

En este paso nos encontramos con el inconveniente de que ninguna clase generada dispone de propiedades y por lo tanto los objetos no se pueden enlazar a la interfaz.

Un intento de solucionar este problema fue con el *plugin* JAXBFX [4] para JAXB, el cual añade una propiedad por atributo además de los métodos necesarios para acceder y modificar las respectivas propiedades. Sin embargo, este *plugin* creó una serie de errores mayores a la hora de trabajar con propiedades de colección e introducía una dependencia adicional que preferíamos evitar.

Al final se optó por crear una capa de modelos separada de la capa de persistencia. La capa de persistencia son los datos puros (*POJOs),* mientras que los modelos proporcionan propiedades y métodos necesarios para poder ser enlazadas a la interfaz.

Así pues el proceso básico que se sigue para trabajar con los elementos del *estab* es:

1. *Unmarshall.* Pasar la información guardada en el fichero a objetos de elementos.
2. Inicializar el modelo correspondiente de cada objeto cargado.
3. Enlazar en el controlador las propiedades de estos adaptadores con las propiedades del editor con el que se mostrará la información del elemento.
4. Si se va a mostrar un nuevo elemento en el editor, antes de volver a enlazar las propiedades, desenlazar las del elemento anterior y repetir el paso 3 con el nuevo elemento.

Los elementos definidos en el diseño original del estab tienen una clase Java correspondiente en la capa de persistencia, y una clase asociada en la capa de modelo. La convención utilizada en el proyecto es añadir “Model” al nombre del elemento, de manera que, por ejemplo, al adaptador de la clase `Vehicle` se le llamará `VehicleModel`.

El modelo añade código necesario para encapsular los atributos en propiedades y darles lógica para trabajar con ellos. Siguiendo con la clase `Vehicle`, el atributo `crew` de tipo `int` será representado en su adaptador por la propiedad de tipo `IntegerPropierty` y añadirá los métodos necesarios para acceder a dicha propiedad como se puede ver a continuación:

```
// POJO Vehicle Class
public class Vehicle {

    protected int crew;

    public int getCrew() {
        return crew;
    }

    public void setCrew(int value) {
        this.crew = value;
    }
}


// Vehicle Adapter Class
public class VehicleModel
       extends AbstractElementModel<VehicleModel>
       implements PojoAdapter<Vehicle> {

    private final IntegerProperty crew = new
SimpleIntegerProperty();

    public int getCrew() {
        return crew.get();
    }

    public void setCrew(int crew) {
        this.crew.set(crew);
    }

    public IntegerProperty crewProperty() {
        return crew;
    }
}
```

El modelo mantiene los métodos de acceso de la variable y añade el de la propiedad.

Todo modelo extiende la clase abstracta `AbstractElementModel` y la interfaz genérica `PojoAdapter` para asegurarse de que cumplen con ciertos requisitos.

`PojoAdapter`. Los modelos han de tener un método con el que pueda se obtenga el POJO que encapsula. Es la interfaz `PojoAdapter` la que define dicho método.

`AbstractElementModel`. Cada uno de los elementos (y por lo tanto, también todos los modelos) tienen atributos en común, en concreto el nombre, el identificador, una descripción y una colección de banderas que determina el estado del elemento (si es nuevo, si ha sido editado desde que se ha abierto el *estab*, o si se quiere eliminar).

Para facilitar el procesado de elementos sin depender del tipo específico se ha creado una interfaz genérica llamada `ElementModel` que los engloba. `AbstractElementModel` implementa esta interfaz y declara los atributos mencionados además de sus métodos de acceso correspondientes.

Al abrir un *estab*, sea en modo lectura o de edición, se crea una instancia de `EstabModel` en la que los elementos del fichero son guardados en colecciones. Esta clase se encarga de mantener y gestionar los elementos. Aquí se ve claramente la utilidad de crear la interfaz `ElementModel`.

Por ejemplo, el siguiente código muestra el momento en el que `EstabModel` crea los modelos de los elementos para guardarlos en las respectivas colecciones y actualiza el identificador máximo del *estab* si fuese necesario.

```
// Wrap all the elements to their element model and saves them
to their corresponding map

// maxID is an array because we need an effective final variable
to access it within the lambda expression
final int[] maxId = {0};
for (List<? extends ElementModel> elements : estabLists)
    elements.stream().forEach(element -> {
        if (element.getId() > maxId[0])
                    maxId[0] = element.getId();
           element.shallowCopyToMap(
                           allElements.get(element.getClass()));
         }
     );
```

```
/**
 * Duplicates elements into the model.
 * For every element the max id will increment and
 * a soft copy (only references) will be added to the model.
 *
 * @param elements collection of elements to duplicate
 */

@SuppressWarnings("unchecked")
public void duplicate(Collection<ElementModel> elements) {
    for (ElementModel selectedItem : elements)
        selectedItem.cloneToMap(
             ElementModelFactory.incrementMaxId(),
             allElements.get(selectedItem.getClass()));
}
/**
 * Return a list of elements of the given class ({@code
elementClass}) whose name matches exactly the provided {@code
query}.
 *
 * @param query        text to search
 * @param elementClass used to load the corresponding map
 * @return collection with all matching elements
 */

public List<ElementModel> searchExactElement(String query,
Class<? extends   ElementModel> elementClass) {
    return
allElements.get(elementClass).values().parallelStream()
            .filter(element ->

      element.getName().toLowerCase().equals(query.toLowerCase()
))
            .collect(Collectors.toCollection(ArrayList::new));
}
```

En el extracto de código del método `duplícate` se itera sobre la colección que se pasa como parámetro para duplicar cada elemento en el modelo. Como se puede observar la operación no es dependiente de un elemento.

Para la búsqueda de elementos en el método `searchExactElement` se ha aprovechado la actualización 8 de Java con el sistema de lambdas [7] que optimiza las operaciones sobre colecciones. Una vez más el diseño con herencia y polimorfismo abstrae la implementción y ahorra código.

4.6 Interfaces en FXML

El primer paso para diseñar la interfaz es aprender a utilizar FXML, un lenguaje declarativo basado en XML creado específicamente para definir las interfaces de usuario en JavaFX. Es importante saber manejar los contenedores para tener control sobre la posición y la organización de los componentes en los que se visualizará la información.

La interfaz principal está basada en un `BorderPane` que es un tipo de contenedor dividido en cinco regiones: los cuatro puntos cardinales y el centro. La parte superior contiene la barra de menús y la barra de herramientas, mientras que en la central se hace una referencia doble al mismo archivo en el que se ha diseñado la interfaz del *estab* (una para el *estab* de destino y otra para el de origen)

```
<BorderPane fx:id="mainPane" stylesheets="@../css/style.css"
xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="net.deludobellico.estabeditorfx.controller.MainController">
    <top>
        <VBox>
            <children>
                <MenuBar BorderPane.alignment="CENTER">
                <!-- MenuBar Components-->  </MenuBar>
                <ToolBar fx:id="toolBar">
                <!-- ToolBar Components--> </ToolBar>
            </children>
        </VBox>
    </top>
    <center>
        <FlowPane BorderPane.alignment="CENTER">
            <children>
                 <fx:include fx:id="sourcePane" source="estab-editor.fxml" />
                 <fx:include fx:id="targetPane" source="estab-editor.fxml" />
            </children>
        </FlowPane>
    </center>
</BorderPane>
```

En el código se muestra un extracto del código de la interfaz donde se puede apreciar la declaración del panel principal, con la clase de su controlador `MainController` y los componentes contenidos en las dos regiones utilizadas.

En el caso de la interfaz de *estabs* se ha optado por utilizar contenedores simples como `VBox` y `HBox` que añaden nuevos componentes en un flujo vertical y horizontal respectivamente. Para poder reutilizar el mismo archivo para ambos tipos de *estab*, se han definido todos los componentes que pueden aparecer en ambos,

controlando la visibilidad mediante lógica en la capa de controladores. Un ejemplo de esto es el botón de eliminar un elemento seleccionado en la lista, el cual aparece invisible en el *estab* origen para evitar suprimir información.

Dentro de la interfaz de *estab* se ha definido un panel destinado a la carga dinámica de editores de elementos. Estos editores están diseñados de manera tal que toda la información mostrada se adapte al espacio reservado para este panel. Debido a la cantidad de información la mayoría de editores organizan los componentes en pestañas (`TabPane`) como se muestra en el siguiente código:

```
<VBox fx:id="editorPane" styleClass="element-editor"
stylesheets="@../css/style.css" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="net.deludobellico.estabeditorfx.controller.editors.ForceEditor
Controller">
     <children>
        <!-- Other components -->
         <TabPane minWidth="470.0" tabClosingPolicy="UNAVAILABLE">
             <tabs>
                 <Tab text="General"> <!-- Tab content --> </Tab>
                 <Tab text="Equipment/Supply"> <!-- Tab content --> </Tab>
                 <Tab text="Icon">
                 <Tab text="Compose"> <!-- Tab content --> </Tab>
             </tabs>
         </TabPane>
     </children>
 </VBox>
```

A continuación podemos ver un ejemplo simple del botón de crear munición en la barra de herramientas de la aplicación.

La parte importante del código del botón en FXML es el atributo `onAction`, el cual enlaza el método definido en el controlador a los eventos del botón de manera automática.

```
<Button fx:id="createNewAmmoButton"
        disable="true" maxHeight="Infinity"
        mnemonicParsing="true" onAction="#createNewAmmoAction">
    <graphic>
        <ImageView pickOnBounds="true" preserveRatio="true">
            <image>
                <Image url="@../images/new_ammo.png" />
            </image>
        </ImageView>
    </graphic>
    <tooltip>
        <Tooltip text="New ammunition" />
    </tooltip>
</Button>
```

Una vez la aplicación se esté ejecutando y el usuario presione el botón de crear munición, un disparador generará una señal que invocará el método `createNewAmmoAction()`. En ese momento el controlador mandará al modelo a crear un nuevo elemento de munición y una vez finalizado actualizará la interfaz para mostrarlo en su editor correspondiente.

```
@FXML
private void createNewAmmoAction() {
    targetPaneController.createNewElement(new AmmoModel());
}
```

4.7 Pruebas unitarias

Una vez se han implementado las funcionalidades del programa se debe verificar su correcto funcionamiento con las pruebas adecuadas.

El primer paso es esencial y consiste en diseñar las pruebas unitarias. Es decir, determinar cómo se ejecutarán, qué datos de entrada hay que proporcionar y cuál es la salida esperada del módulo de código que estamos estudiando. Este será el marco de referencia con el que compararemos los resultados del código implementado, de aquí la importancia que tiene crear un diseño sólido. Una buena táctica es generar pruebas lo más simples y triviales posibles, y solo si es necesario ir aumentando la complejidad con los casos posteriores.

Cuando se haya terminado la etapa del diseño y las pruebas estén programadas, queda pasar a la propia ejecución. Si el código

examinado las ha superado, se prosigue a las pruebas de integración que comprueba cómo se comporta el código en el ambiente en el que se va a desenvolver. Si por el contrario el código no las ha superado se procederá a la resolución de los fallos y se comprobará una vez más.

En ningún momento una prueba se eliminará haya sido exitosa o no siempre y cuando siga siendo un requisito del análisis la funcionalidad que examina. Cada caso de prueba que haya sido utilizado deberá ejecutarse siempre junto al resto de casos. Con esto se asegurar que las funcionalidades ya implementadas no tengan problemas con el código nuevo.

Los casos de ejemplo que se muestran más adelante ponen a prueba la funcionalidad de comparación de elementos (método `compareTo`). El resultado será positivo cuando dos elementos iguales los identifique como iguales, y negativo en caso contrario.

```
@Test
public void testCompareToEqualsTrue() throws Exception {
	SideModel ours = new SideModel();
	SideModel other = new SideModel();
	initToDefaultValues(ours);
	initToDefaultValues(other);
	assertEquals(true, other.compareTo(ours));
}
```

```
@Test
public void testDifferentIdEqualsFalse() throws Exception {
	SideModel ours = new SideModel();
	SideModel other = new SideModel();
	initToDefaultValues(ours);
	initToDefaultValues(other);
	ours.setID(other.getID+1);
	assertEquals(false, other.compareTo(ours));
}
```

La primera prueba inicializa dos elementos con los mismos datos y comprueba si el resultado de `compareTo` es igual a `true`. El método `assertEquals` devuelve positivo si se cumple la condición de igualdad, proporcionándole el resultado esperado como primer parámetro y el resultado actual como Segundo. Es este caso la prueba es satisfactoria siempre y cuando el método que hemos implementado devuelva `true`.

La segunda prueba realiza los mismos pasos que la primera con la excepción de que el identificador de uno de los elementos se ha cambiado. Ahora el resultado que se espera es `false`.

Como se puede ver las pruebas tienen nombres que describen lo que ocurre en ellas y son muy simples, casi triviales. Es precisamente la simplicidad lo que las hace fácil de depurar y detectar fallos en el código desarrollado.

Cada caso de prueba ha de generarse por separado, o lo que es lo mismo en métodos diferentes sin más de una aserción. De la misma manera, cada conjunto de pruebas ha de generarse por cada elemento.

Se incluye un ejemplo de resultados durante el desarrollo del proyecto (Figura 27) en el que algunas pruebas esperaban unos resultados diferentes a los obtenidos.

En la figura 28 se ven las clases implementadas en el proyecto y las clases correspondientes en el paquete de pruebas.

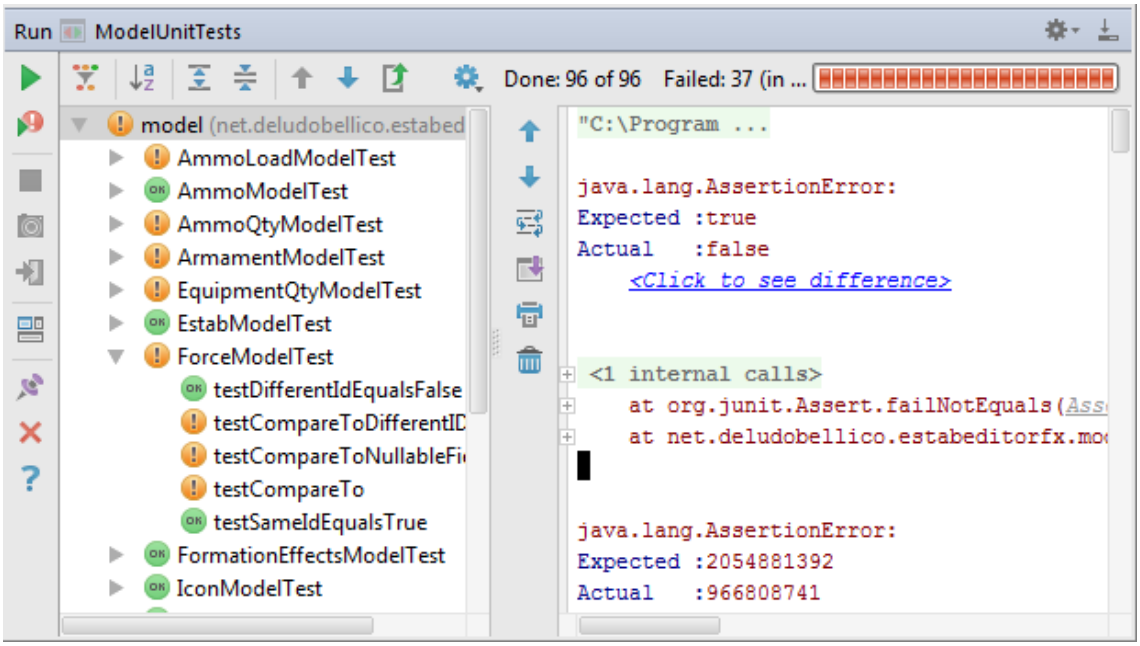

Figura 27. Resultado de pruebas unitarias en el paquete model.

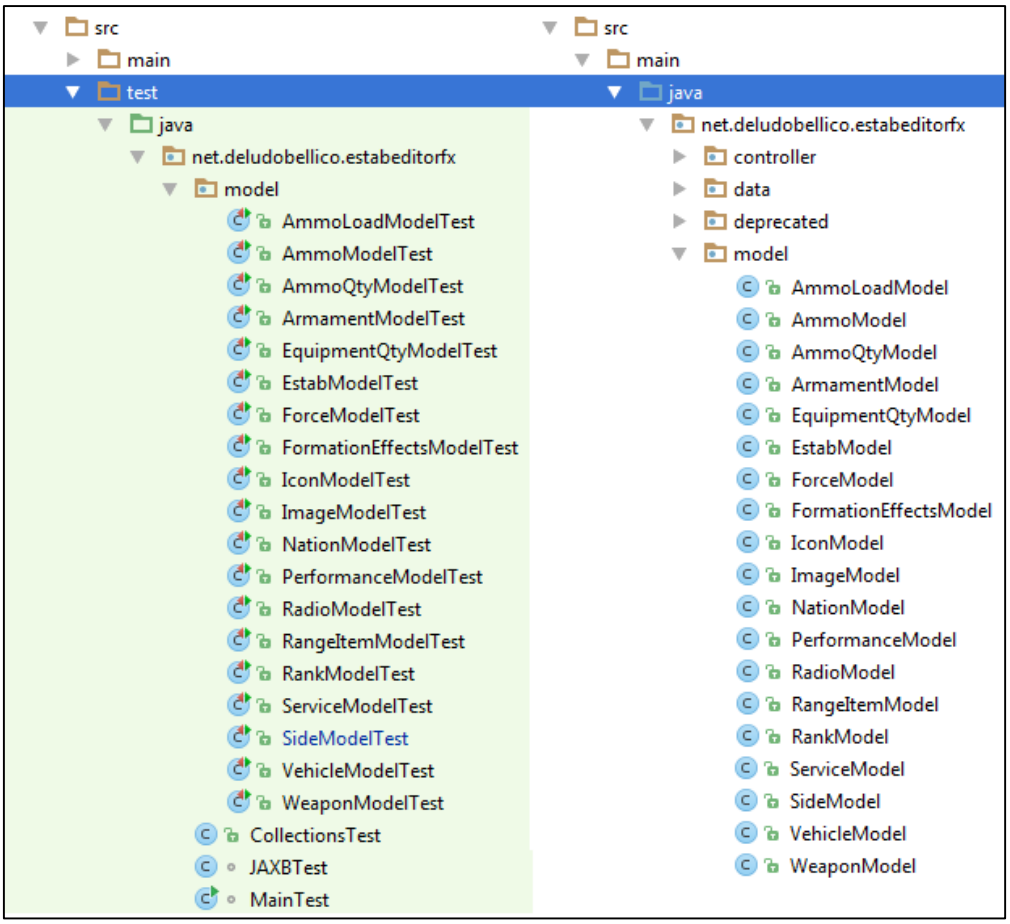

Figura 28. Clases de pruebas correspondientes a las clases en el modelo. Se han omitido algunas clases para simplificar la imagen.

Conclusiones

En todo momento del desarrollo el objetivo primordial era cumplir con los requisitos definidos al comienzo del proyecto y se puede afirmar que se han superado.

Dicho esto también es cierto que la aplicación no tiene porqué quedarse limitada a esos requisitos, tiene potencial para que se siga trabajando en ella. La forma en la que se ha desarrollado con el patrón MVC favoreciendo la modularidad hace que sea sencillo la adición y modificación de funcionalidades. En el caso de que exista una base de usuarios que lo soliciten la aplicación podría seguir mejorándose, por ejemplo:

- Implementando las funcionalidades de las Radios y de los Efectos de Formaciones.
- Dando la posibilidad de abrir varios editores del mismo tipo.
- Permitiendo desacoplar los editores de la interfaz principal en ventanas externas.
- Portando la aplicación a otra plataforma aprovechando las facilidades que ofrece JavaFX en este aspecto.

En retrospectiva la decisión de haber escogido JavaFX como *framework* de la aplicación ha sido más que acertada. Esta herramienta ha demostrado ser sencilla de utilizar, ligera, muy bien documentada y soportada por una gran comunidad. Tal vez el único inconveniente a la hora de elaborar interfaces ha sido al usar Scene Builder[12], que se trata de una herramienta diseñada específicamente para el desarrollo de interfaces en JavaFX. Esta herramienta tiene algunos defectos, como el hecho de que no soporta bien trabajar con interfaces complejas cargadas de componentes, es más conveniente para crear diálogos o ventanas simples.

Ha sido una ayuda apoyarse en la interfaz del editor original, sobre todo a la hora de organizar toda la información que contienen los elementos en los editores. Los problemas de basarse en el paquete del juego se dieron más en la capa de modelo que en la de interfaz. Uno que ha estado presente a lo largo de todo el proyecto y que ha creado más dificultades de las previstas es el horrible diseño original del modelo de clases en los *estab* (apartado 3.2), es decir las que están contenidas en la capa de persistencia. Por ejemplo, en esta

capa existe una clase llamada `YesNo` que únicamente define un booleano y ni siquiera se usa siempre para declarar este tipo de datos. En general la utilidad de algunas de estas clases y sus relaciones generan más preguntas que respuestas. La creación de un modelo propio con el que poder trabajar era una necesidad primordial.

Por último, y de forma personal, mencionar que ha sido un placer formar parte de este equipo de desarrollo. Solo hay agradecimiento con todo el apoyo que he recibido.

Bibliografía

[1] Matrix Games - Command Ops: Battle from the Bulge. [Consulta: 25 de agosto 2015]. Disponible en: http://www.matrixgames.com/products/377/details/Command.Ops:.Battles.from.the.Bulge.

[2] JavaFX Developer Home. [Consulta: 25 de agosto 2015]. Disponible en: http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html

[3] Introduction to FXML. [Consulta: 25 de agosto 2015]. Disponible en: https://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html

[4] Working With Layouts in JavaFX: Using Built-in Layout Panes. [Consulta: 25 de agosto 2015]. Disponible en: https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm

[5] Coffin, David, and CarlDea. "JavaFX 8: Introduction by Example". A Press, 2014.

[6] A JAXB2 XJC plugin to generate JavaFX properties. [Consulta: 25 de agosto 2015]. Disponible en: https://github.com/buschmais/jaxbfx

[7] Neward, Ted. "Java 8: Lambdas Part 1". Java Magazine [en línea], Julio/Agosto 2013. [Consulta: 25 agosto 2015]. Disponible en: http://www.oracle.com/technetwork/articles/java/architect-lambdas-part1-2080972.html

[8] JavaFX FAQ. [Consulta: 25 agosto 2015]. Disponible en: http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6

[9] Project JAXB. [Consulta: 25 agosto 2015]. Disponible en: https://jaxb.java.net/2.2.11/docs/ch01.html#documentation

[10] Java Architecture for XML Binding (JAXB) [Consulta: 25 agosto 2015]. Disponible en: http://www.oracle.com/technetwork/articles/javase/index-140168.html

[11] XSD How To? [Consulta: 25 de agosto 2015]. Disponible en: http://www.w3schools.com/schema/schema_howto.asp

[12] Scene Builder – Overview. [Consulta: 25 de agosto 2015]. Disponible en: https://docs.oracle.com/javase/8/scene-builder-2/get-started-tutorial/overview.htm#JSBGS164

[13] Repositorio del proyecto en GitHub: https://github.com/magomar/EstabEditorFX

Anexos

Anexo A. Glosario

Adaptador: Clase que encapsula una serie de atributos u otra clase para añadirle funcionalidad extra.

API: *Application programming interface.* Conjunto de rutinas, protocolos y herramientas para construir aplicaciones de software.

Estab: Abreviación de “establecimiento”. Este término se puede referir tanto a las unidades militares y su equipamiento como al propio archivo en disco donde estas se almacenan.

Evento: Ocurrencia o acción detectada que puede ser controlada por el programa.

Framework: Es un conjunto de aplicaciones y técnicas que sirven de soporte para construir un software.

FXML: Lenguaje declarativo basado en XML diseñado para definir interfaces de usuario en aplicaciones JavaFX.

Getter: Método que permiten obtener el valor de una variable.

Handler: Subrutinas que se ejecutan, si el evento ocurrido es el correcto.

IDE: *Integrated Development Environment* es una aplicación destinada al desarrollo de software.

IGU: Interfaz Gráfica de Usuario. Es la parte estética de la aplicaciñon.

Layout: Configuración de la posición y geometría de los contenedores en JavaFX.

Listener: Subrutinas que se ejecutan si el evento ocurrido es el correcto.

Marshall: Proceso de pasar de instancia a capa persostente de datos.

POJO: Objeto Java de una clase que no implementa ninguna interfaz especial ni extiende de otra clase.

Propiedad: Clase que encapsula a un atributo proporcionando getters y setters.

Repositorio: Lugar en el que se almacenan los datos actualizados e histórico de cambios de un código.

Serialización: Paso de información en la capa de modelo a datos persistentes.

Setter: Método que permite modificar el valor de una variable.

SO: Sistema Operativo. Sistema software esencial que gestiona recursos hardware y software en un ordenador.

Unmarshall: Paso de información de los elementos de archivo en disco a estancias.

XML: *Extensible Markup Language.* Es un lenguaje de marcado que define una serie de reglas para codificar documentos en un formato que es tanto legible para humanos como para máquinas.

Anexo B. Esquema XSD para la persistencia de datos.

```
<?xml version="2.0" encoding="UTF-8"?>
<xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        attributeFormDefault="unqualified" elementFormDefault="qualified">

    <!-- ESTAB data -->
    <xs:element name="estab-data">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="image" maxOccurs="unbounded" minOccurs="0"/>
                <xs:element ref="side" maxOccurs="unbounded" minOccurs="2"/>
                <xs:element ref="vehicle" maxOccurs="unbounded" minOccurs="0"/>
                <xs:element ref="weapon" maxOccurs="unbounded" minOccurs="1"/>
                <xs:element ref="radio" maxOccurs="unbounded" minOccurs="0"/>
                <xs:element ref="ammo" maxOccurs="unbounded" minOccurs="1"/>
                <xs:element ref="formation-effects" maxOccurs="unbounded"
minOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="version" type="xs:int" default="4"/>
            <xs:attribute name="dlb-version" type="xs:string"/>
            <xs:attribute name="edited" type="xs:boolean" default="false"/>
            <xs:attribute name="last-edit" type="xs:dateTime"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="image" type="Image"/>
    <xs:element name="side" type="Side"/>
    <xs:element name="vehicle" type="Vehicle"/>
    <xs:element name="weapon" type="Weapon"/>
    <xs:element name="radio" type="Radio"/>
    <xs:element name="ammo" type="Ammo"/>
    <xs:element name="formation-effects" type="FormationEffects"/>

    <!-- Images -->
    <xs:complexType name="Image">
        <xs:attribute name="id" type="xs:int" use="required"/>
        <xs:attribute name="file-id" type="xs:string" use="required"/>
        <xs:attribute name="flags" type="FlagList"/>
    </xs:complexType>

    <!-- Sides -->
    <xs:complexType name="Side">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="large-insignia" type="Insignia"/>
            <xs:element name="small-insignia" type="Insignia"/>
            <xs:element name="basics-consumption-rate" type="xs:double"/>
            <xs:element name="default-enemy-aper-fp" type="xs:int"/>
            <xs:element name="default-enemy-aarm-fp" type="xs:int"/>
            <xs:element name="nation" type="Nation" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:int" use="required"/>
        <xs:attribute name="flags" type="FlagList"/>
    </xs:complexType>
```

```
    <!-- Nations -->
    <xs:complexType name="Nation">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="nationality" type="xs:string"/>
            <xs:element name="large-insignia" type="Insignia"/>
            <xs:element name="small-insignia" type="Insignia"/>
            <xs:element name="service" type="Service" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:int" use="required"/>
        <xs:attribute name="flags" type="FlagList"/>
    </xs:complexType>

    <!-- Services -->
    <xs:complexType name="Service">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="large-insignia" type="Insignia"/>
            <xs:element name="small-insignia" type="Insignia"/>
            <xs:element name="rank-list" type="RankList"/>
            <xs:element name="default-icon-colors" type="DefaultIconColors"/>
            <xs:element name="force" type="Force" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:int" name="id" use="required"/>
        <xs:attribute name="flags" type="FlagList"/>
    </xs:complexType>


    <xs:complexType name="RankList">
        <xs:sequence>
            <xs:element name="rank" type="Rank" maxOccurs="unbounded" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="Rank">
        <xs:attribute name="short-name" type="xs:string" use="required"/>
        <xs:attribute name="full-name" type="xs:string" use="required"/>
    </xs:complexType>


    <xs:complexType name="DefaultIconColors">
        <xs:sequence>
            <xs:element name="background-color" type="RGBColor"/>
            <xs:element name="background-dark-color" type="RGBColor"/>
            <xs:element name="background-light-color" type="RGBColor"/>
            <xs:element name="designation-color" type="RGBColor"/>
            <xs:element name="symbol-color" type="SymbolColor"/>
        </xs:sequence>
    </xs:complexType>
```

```
    <!-- Forces -->
    <xs:complexType name="Force">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="icon" type="Icon"/>
            <xs:element name="type" type="ForceType"/>
            <xs:element name="sub-type" type="ForceSubtype"/>
            <xs:element name="size" type="ForceSize"/>
            <xs:element name="combat-class" type="CombatClass"/>
            <xs:element name="target-class" type="TargetClass"/>
            <xs:element name="infantry-value" type="xs:int"/>
            <xs:element name="recon-value" type="xs:int"/>
            <xs:element name="engineering-value" type="xs:int"/>
            <xs:element name="move-type" type="MoveType"/>
            <xs:element name="pers-qty" type="xs:int"/>
            <xs:element name="staff-capacity" type="xs:int"/>
            <xs:element name="basics-qty" type="xs:double"/>
            <xs:element name="basics-consumption-rate-modifier" type="xs:double"/>
            <xs:element name="commander-rank" type="xs:int"/>
            <xs:element name="fuel-qty" type="xs:double"/>
            <xs:element name="fuel-load" type="xs:double"/>
            <xs:element name="speed" type="Speed"/>
            <xs:element name="deployment-duration" type="DeploymentDuration"/>
            <xs:element name="ready-to-fire-duration" type="xs:string"/>
            <xs:element name="ready-to-bombard-duration" type="xs:string"/>
            <xs:element name="equipment-list" type="EquipmentList"/>
            <xs:element name="ammo-list" type="AmmoList"/>
            <xs:element name="can-bombard" type="YesNo"/>
            <xs:element name="force-composition" type="ForceComposition"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:int" use="required"/>
        <xs:attribute name="flags" type="FlagList"/>
    </xs:complexType>

    <xs:complexType name="Icon">
        <xs:sequence>
            <xs:element name="background-color" type="RGBColor"/>
            <xs:element name="background-dark-color" type="RGBColor"/>
            <xs:element name="background-light-color" type="RGBColor"/>
            <xs:element name="designation-color" type="RGBColor"/>
            <xs:element name="symbol-color" type="SymbolColor"/>
            <xs:element name="military-symbol" type="xs:int"/>
            <xs:element name="picture-symbol" type="PictureSymbol"/>
            <xs:element name="force-size-icon" type="ForceSize"/>
            <xs:element name="is-hq" type="YesNo"/>
        </xs:sequence>
    </xs:complexType>


    <xs:complexType name="Speed">
        <xs:sequence>
            <xs:element name="normal" type="xs:double"/>
            <xs:element name="max" type="xs:double"/>
        </xs:sequence>
    </xs:complexType>
```

```
    <xs:complexType name="DeploymentDuration">
        <xs:sequence>
            <xs:element name="deployed" type="xs:string"/>
            <xs:element name="dug-in" type="xs:string"/>
            <xs:element name="entrenched" type="xs:string"/>
            <xs:element name="fortified" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="EquipmentList">
        <xs:sequence>
            <xs:element name="equipment" type="Equipment" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="Equipment">
        <xs:attribute name="equipment-object-id" type="xs:int" use="required"/>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="qty" type="xs:int" use="required"/>
    </xs:complexType>

    <xs:complexType name="AmmoList">
        <xs:sequence>
            <xs:element name="ammo" type="AmmoQty" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="AmmoQty">
        <xs:attribute name="ammo-object-id" type="xs:int" use="required"/>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="qty" type="xs:int" use="required"/>
    </xs:complexType>

    <xs:complexType name="ForceComposition">
        <xs:sequence>
            <xs:element name="subforce" type="ForceQty" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ForceQty">
        <xs:attribute name="force-object-id" type="xs:int" use="required"/>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="qty" type="xs:int" use="required"/>
    </xs:complexType>
```

```
    <!-- Vehicles -->
    <xs:complexType name="Vehicle">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="picture" type="Picture"/>
            <xs:element name="picture-filename" type="xs:string"/>
            <xs:element name="size" type="VehicleSize"/>
            <xs:element name="crew" type="xs:int"/>
            <xs:element name="reliability" type="Proportion"/>
            <xs:element name="armaments" type="ArmamentList"/>
            <xs:element name="type" type="VehicleType"/>
            <xs:element name="fuel-capacity" type="xs:double"/>
            <xs:element name="speed" type="VehicleSpeeds"/>
            <xs:element name="fuel-consumption" type="FuelConsumption"/>
            <xs:element name="ronsonability" type="Proportion"/>
            <xs:element name="max-gradient" type="xs:int"/>
            <xs:element name="max-fording-depth" type="xs:int"/>
            <xs:element name="max-trench-width" type="xs:int"/>
            <xs:element name="towing-capacity" type="xs:double"/>
            <xs:element name="personnel-capacity" type="xs:int"/>
            <xs:element name="bulk-fuel-capacity" type="xs:double"/>
            <xs:element name="payload-capacity" type="xs:double"/>
            <xs:element name="take-cover-mod" type="xs:double"/>
            <xs:element name="has-turret" type="xs:string"/>
            <xs:element name="has-open-top" type="xs:string"/>
            <xs:element name="battle-weight" type="xs:double"/>
            <xs:element name="armour" type="Armor"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:int" use="required"/>
        <xs:attribute name="flags" type="FlagList"/>
    </xs:complexType>

    <xs:complexType name="Picture">
        <xs:attribute name="id" type="xs:int" use="required"/>
    </xs:complexType>

    <xs:complexType name="VehicleSize">
        <xs:attribute name="width" type="xs:double" use="required"/>
        <xs:attribute name="height" type="xs:double" use="required"/>
        <xs:attribute name="length" type="xs:double" use="required"/>
        <xs:attribute name="weight" type="xs:double" use="required"/>
    </xs:complexType>

    <xs:complexType name="ArmamentList">
        <xs:sequence>
            <xs:element name="armament" type="Armament" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="Armament">
        <xs:attribute type="xs:int" name="equipment-object-id" use="required"/>
        <xs:attribute type="xs:string" name="equipment-name" use="required"/>
        <xs:attribute type="xs:int" name="qty" use="required"/>
    </xs:complexType>

    <xs:complexType name="VehicleSpeeds">
        <xs:sequence>
            <xs:element name="road" type="SpeedData"/>
            <xs:element name="cross-country" type="SpeedData"/>
        </xs:sequence>
    </xs:complexType>
```

```
<xs:complexType name="Armor">
    <xs:attribute name="front" type="xs:double" use="required"/>
    <xs:attribute name="side" type="xs:double" use="required"/>
    <xs:attribute name="rear" type="xs:double" use="required"/>
    <xs:attribute name="top" type="xs:double" use="required"/>
</xs:complexType>

<!-- Weapons -->

<xs:complexType name="Weapon">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element name="picture" type="Picture"/>
        <xs:element name="picture-filename" type="xs:string"/>
        <xs:element name="size" type="WeaponSize"/>
        <xs:element name="crew" type="xs:int"/>
        <xs:element name="reliability" type="Proportion"/>
        <xs:element name="armaments" type="xs:string"/>
        <xs:element name="type" type="WeaponType"/>
        <xs:element name="single-shot" type="YesNo"/>
        <xs:element name="primary-role" type="PrimaryRole"/>
        <xs:element name="calibre" type="xs:double"/>
        <xs:element name="muzzle-velocity" type="xs:int"/>
        <xs:element name="must-deploy-to-fire" type="YesNo"/>
        <xs:element name="performance-list" type="PerformanceList"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
    <xs:attribute name="flags" type="FlagList"/>
</xs:complexType>

<xs:complexType name="WeaponSize">
    <xs:attribute name="width" type="xs:double" use="required"/>
    <xs:attribute name="height" type="xs:double" use="required"/>
    <xs:attribute name="length" type="xs:double" use="required"/>
    <xs:attribute name="weight" type="xs:double" use="required"/>
</xs:complexType>

<xs:complexType name="PerformanceList">
    <xs:sequence>
        <xs:element name="performance" type="Performance" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="Performance">
    <xs:sequence>
        <xs:element name="ammo" type="AmmoLoad"/>
        <xs:element name="min-range" type="xs:int"/>
        <xs:element name="rof" type="ROF"/>
        <xs:element name="burst-radius" type="xs:int"/>
        <xs:element name="shell-weight" type="xs:double"/>
        <xs:element name="range-table" type="RangeTable"/>
    </xs:sequence>
    <xs:attribute name="fire-type" type="FireType" use="required"/>
</xs:complexType>

<xs:complexType name="AmmoLoad">
    <xs:attribute name="object-id" type="xs:int" use="required"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="load" type="xs:int" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="ROF">
    <xs:attribute name="slow" type="xs:double" use="required"/>
    <xs:attribute name="normal" type="xs:double" use="required"/>
    <xs:attribute name="rapid" type="xs:double" use="required"/>
</xs:complexType>

<xs:complexType name="RangeTable">
    <xs:sequence>
        <xs:element name="range-item" type="RangeItem" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="RangeItem">
    <xs:attribute name="range" type="xs:int" use="required"/>
    <xs:attribute name="accuracy" type="Proportion" use="required"/>
    <xs:attribute name="armour-penetration" type="xs:double" use="required"/>
</xs:complexType>

<!-- Radios -->
<xs:complexType name="Radio">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element name="picture" type="xs:string"/>
        <xs:element name="picture-filename" type="xs:string"/>
        <xs:element name="size" type="RadioSize"/>
        <xs:element name="crew" type="xs:string"/>
        <xs:element name="reliability" type="Proportion"/>
        <xs:element name="armaments" type="xs:string"/>
        <!--why empty?-->
        <xs:element name="net-type" type="NetType"/>
        <xs:element name="freq-type" type="FreqType"/>
        <xs:element name="max-range" type="xs:int"/>
        <xs:element name="gain" type="xs:double"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
    <xs:attribute name="flags" type="FlagList"/>
</xs:complexType>

<xs:complexType name="RadioSize"> <!-- weight OK, but the rest?  -->
    <xs:attribute name="width" type="xs:double" use="required"/>
    <xs:attribute name="height" type="xs:double" use="required"/>
    <xs:attribute name="length" type="xs:double" use="required"/>
    <xs:attribute name="weight" type="xs:double" use="required"/>
</xs:complexType>

<!-- Ammo -->
<xs:complexType name="Ammo">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element name="min-order-qty" type="xs:int"/>
        <xs:element name="min-order-weight" type="xs:double"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
    <xs:attribute name="flags" type="FlagList"/>
</xs:complexType>
```

```
<!-- Formation effects -->

<xs:complexType name="FormationEffects">
    <xs:sequence>
        <xs:element name="moving-cohesion-level" type="xs:double"/>
        <xs:element name="frontage-per-man" type="xs:double"/>
        <xs:element name="depth-per-man" type="xs:double"/>
        <xs:element name="firing-percentages" type="FiringPercentages"/>
        <xs:element name="target-percentages" type="TargetPercentages"/>
        <xs:element name="security" type="FormationSecurity"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
    <xs:attribute name="type" type="FormationType" use="required"/>
    <xs:attribute name="flags" type="FlagList"/>
</xs:complexType>

<xs:complexType name="FiringPercentages">
    <xs:sequence>
        <xs:element name="front" type="Proportion"/>
        <xs:element name="left" type="Proportion"/>
        <xs:element name="right" type="Proportion"/>
        <xs:element name="rear" type="Proportion"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="TargetPercentages">
    <xs:sequence>
        <xs:element name="front" type="Proportion"/>
        <xs:element name="left" type="Proportion"/>
        <xs:element name="right" type="Proportion"/>
        <xs:element name="rear" type="Proportion"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="FormationSecurity">
    <xs:sequence>
        <xs:element name="front" type="Security"/>
        <xs:element name="left" type="Security"/>
        <xs:element name="right" type="Security"/>
        <xs:element name="rear" type="Security"/>
    </xs:sequence>
</xs:complexType>


<!-- ****************  GLOBAL TYPES  ******************** -->


<xs:simpleType name="Proportion">
    <xs:restriction base="xs:double">
        <xs:minInclusive value="0.0"/>
        <xs:maxInclusive value="1.0"/>
    </xs:restriction>
</xs:simpleType>

<xs:complexType name="Insignia">
    <xs:attribute name="id" type="xs:int" use="required"/>
</xs:complexType>

<xs:complexType name="SpeedData">
    <xs:attribute name="max" type="xs:double" use="required"/>
    <xs:attribute name="normal" type="xs:double" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="FuelConsumption">
    <xs:attribute name="max" type="xs:double" use="required"/>
    <xs:attribute name="normal" type="xs:double" use="required"/>
</xs:complexType>

<xs:complexType name="RGBColor">
    <xs:attribute name="red" type="xs:int" use="required"/>
    <xs:attribute name="green" type="xs:int" use="required"/>
    <xs:attribute name="blue" type="xs:int" use="required"/>
</xs:complexType>

<!-- ****************  Enumerations  ******************* -->

<xs:simpleType name="Security">
    <xs:restriction base="xs:string">
        <xs:enumeration value="min"/>
        <xs:enumeration value="normal"/>
        <xs:enumeration value="max"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="FormationType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="mob"/>
        <xs:enumeration value="road-column"/>
        <xs:enumeration value="line"/>
        <xs:enumeration value="successive-lines"/>
        <xs:enumeration value="square"/>
        <xs:enumeration value="arrow-head"/>
        <xs:enumeration value="left-echelon"/>
        <xs:enumeration value="right-echelon"/>
        <xs:enumeration value="vee"/>
        <xs:enumeration value="all-round-defence"/>
        <xs:enumeration value="in-situ"/>
        <xs:enumeration value="unspecified"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="FireType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="aper"/>
        <xs:enumeration value="aarm"/>
        <xs:enumeration value="aair"/>
        <xs:enumeration value="bombard"/>
        <xs:enumeration value="smoke"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="FreqType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="hf"/>
        <xs:enumeration value="vhf"/>
        <xs:enumeration value="uhf"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="NetType"> <!-- only one value?  -->
    <xs:restriction base="xs:string">
        <xs:enumeration value="land"/>
    </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="SymbolColor">
    <xs:restriction base="xs:string">
        <xs:enumeration value="black"/>
        <xs:enumeration value="white"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="YesNo">
    <xs:restriction base="xs:string">
        <xs:enumeration value="yes"/>
        <xs:enumeration value="no"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ForceSize">
    <xs:restriction base="xs:string">
        <xs:enumeration value="section"/>
        <xs:enumeration value="platoon"/>
        <xs:enumeration value="company"/>
        <xs:enumeration value="battalion"/>
        <xs:enumeration value="brigade"/>
        <xs:enumeration value="regiment"/>
        <xs:enumeration value="division"/>
        <xs:enumeration value="corps"/>
        <xs:enumeration value="army"/>
        <xs:enumeration value="army-group"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="MoveType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="foot"/>
        <xs:enumeration value="wheeled"/>
        <xs:enumeration value="tracked"/>
        <xs:enumeration value="half-track"/>
        <xs:enumeration value="horse"/>
        <xs:enumeration value="bicycle"/>
        <xs:enumeration value="air"/>
        <xs:enumeration value="sea"/>
        <xs:enumeration value="sub"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="CombatClass">
    <xs:restriction base="xs:string">
        <xs:enumeration value="line"/>
        <xs:enumeration value="line-support"/>
        <xs:enumeration value="support"/>
        <xs:enumeration value="hq"/>
        <xs:enumeration value="base"/>
        <xs:enumeration value="static"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TargetClass">
    <xs:restriction base="xs:string">
        <xs:enumeration value="soft"/>
        <xs:enumeration value="hard"/>
        <xs:enumeration value="mixed"/>
    </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="PictureSymbol">
    <xs:restriction base="xs:string">
        <xs:enumeration value="tank"/>
        <xs:enumeration value="assault-gun"/>
        <xs:enumeration value="light-tank"/>
        <xs:enumeration value="armoured-car"/>
        <xs:enumeration value="artillery"/>
        <xs:enumeration value="at-gun"/>
        <xs:enumeration value="heavy-flak"/>
        <xs:enumeration value="light-flak"/>
        <xs:enumeration value="rockets"/>
        <xs:enumeration value="sp-artillery"/>
        <xs:enumeration value="engineer"/>
        <xs:enumeration value="bridging"/>
        <xs:enumeration value="mech-inf"/>
        <xs:enumeration value="motor-inf"/>
        <xs:enumeration value="infantry"/>
        <xs:enumeration value="para"/>
        <xs:enumeration value="hq"/>
        <xs:enumeration value="hmg"/>
        <xs:enumeration value="mortar"/>
        <xs:enumeration value="base"/>
        <xs:enumeration value="ammo"/>
        <xs:enumeration value="fuel"/>
        <xs:enumeration value="bridge"/>
        <xs:enumeration value="heavy-tank"/>
        <xs:enumeration value="inf-gun"/>
        <xs:enumeration value="manpack"/>
        <xs:enumeration value="motorcycle"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ForceType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="infantry"/>
        <xs:enumeration value="armour"/>
        <xs:enumeration value="artillery"/>
        <xs:enumeration value="logistics"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ForceSubtype">
    <xs:restriction base="xs:string">
        <!-- Infantry subtypes -->
        <xs:enumeration value="Infantry"/>
        <xs:enumeration value="Leg Infantry"/>
        <xs:enumeration value="Mot Infantry"/>
        <xs:enumeration value="Mech Infantry"/>
        <xs:enumeration value="Para Infantry"/>
        <xs:enumeration value="Glider Infantry"/>
        <xs:enumeration value="Combat Engineer"/>
        <xs:enumeration value="AT Infantry"/>
        <xs:enumeration value="SMG Infantry"/>
        <xs:enumeration value="MG Infantry"/>
        <xs:enumeration value="Hvy Wpns Infantry"/>
        <xs:enumeration value="Special Forces"/>
        <xs:enumeration value="Mountain Infantry"/>
        <xs:enumeration value="Bicycle Infantry"/>
        <xs:enumeration value="Cavalry"/>
        <xs:enumeration value="Leg Recon"/>
        <xs:enumeration value="Mot Recon"/>
        <xs:enumeration value="Mech Recon"/>
        <xs:enumeration value="Non-Armoured HQ"/>
```

```
            <!-- Armor subtypes -->
            <xs:enumeration value="Armour"/>
            <xs:enumeration value="Tank"/>
            <xs:enumeration value="Assault Gun"/>
            <xs:enumeration value="Inf Support AG"/>
            <xs:enumeration value="Tank Destroyer"/>
            <xs:enumeration value="Armoured Car"/>
            <xs:enumeration value="Armoured Recon"/>
            <xs:enumeration value="Flame-Thrower AFV"/>
            <xs:enumeration value="Armoured HQ"/>
            <!-- Artillery -->
            <xs:enumeration value="Artillery"/>
            <xs:enumeration value="Anti-Tank"/>
            <xs:enumeration value="SP AT"/>
            <xs:enumeration value="Mortar"/>
            <xs:enumeration value="SP Mortar"/>
            <xs:enumeration value="Howitzer"/>
            <xs:enumeration value="SP Howitzer"/>
            <xs:enumeration value="Gun"/>
            <xs:enumeration value="SP Gun"/>
            <xs:enumeration value="Inf Gun"/>
            <xs:enumeration value="SP Inf Gun"/>
            <xs:enumeration value="Rocket Launcher"/>
            <xs:enumeration value="SP RL"/>
            <xs:enumeration value="Flak"/>
            <xs:enumeration value="SP Flak"/>
            <xs:enumeration value="Mot Howitzer"/>
            <xs:enumeration value="Mot Mortar"/>
            <xs:enumeration value="Mot Gun"/>
            <xs:enumeration value="Mot Inf Gun"/>
            <xs:enumeration value="Mot RL"/>
            <xs:enumeration value="Mot Flak"/>
            <xs:enumeration value="Mot AT"/>
            <xs:enumeration value="Abn Arty"/>
            <xs:enumeration value="Para Arty"/>
            <!-- Logistics -->
            <xs:enumeration value="Logistics"/>
            <xs:enumeration value="Base"/>
            <xs:enumeration value="Bridge Engineer"/>
            <xs:enumeration value="Const Engineer"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="VehicleType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="tank"/>
            <xs:enumeration value="tank-destroyer"/>
            <xs:enumeration value="assault-gun"/>
            <xs:enumeration value="self-propelled-artillery"/>
            <xs:enumeration value="armoured-car"/>
            <xs:enumeration value="other-afv"/>
            <xs:enumeration value="truck"/>
            <xs:enumeration value="car"/>
            <xs:enumeration value="motor-cycle"/>
            <xs:enumeration value="construction-vehicle"/>
        </xs:restriction>
    </xs:simpleType>
```

```
            <!-- Armor subtypes -->
            <xs:enumeration value="Armour"/>
            <xs:enumeration value="Tank"/>
            <xs:enumeration value="Assault Gun"/>
            <xs:enumeration value="Inf Support AG"/>
            <xs:enumeration value="Tank Destroyer"/>
            <xs:enumeration value="Armoured Car"/>
            <xs:enumeration value="Armoured Recon"/>
            <xs:enumeration value="Flame-Thrower AFV"/>
            <xs:enumeration value="Armoured HQ"/>
            <!-- Artillery -->
            <xs:enumeration value="Artillery"/>
            <xs:enumeration value="Anti-Tank"/>
            <xs:enumeration value="SP AT"/>
            <xs:enumeration value="Mortar"/>
            <xs:enumeration value="SP Mortar"/>
            <xs:enumeration value="Howitzer"/>
            <xs:enumeration value="SP Howitzer"/>
            <xs:enumeration value="Gun"/>
            <xs:enumeration value="SP Gun"/>
            <xs:enumeration value="Inf Gun"/>
            <xs:enumeration value="SP Inf Gun"/>
            <xs:enumeration value="Rocket Launcher"/>
            <xs:enumeration value="SP RL"/>
            <xs:enumeration value="Flak"/>
            <xs:enumeration value="SP Flak"/>
            <xs:enumeration value="Mot Howitzer"/>
            <xs:enumeration value="Mot Mortar"/>
            <xs:enumeration value="Mot Gun"/>
            <xs:enumeration value="Mot Inf Gun"/>
            <xs:enumeration value="Mot RL"/>
            <xs:enumeration value="Mot Flak"/>
            <xs:enumeration value="Mot AT"/>
            <xs:enumeration value="Abn Arty"/>
            <xs:enumeration value="Para Arty"/>
            <!-- Logistics -->
            <xs:enumeration value="Logistics"/>
            <xs:enumeration value="Base"/>
            <xs:enumeration value="Bridge Engineer"/>
            <xs:enumeration value="Const Engineer"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="VehicleType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="tank"/>
            <xs:enumeration value="tank-destroyer"/>
            <xs:enumeration value="assault-gun"/>
            <xs:enumeration value="self-propelled-artillery"/>
            <xs:enumeration value="armoured-car"/>
            <xs:enumeration value="other-afv"/>
            <xs:enumeration value="truck"/>
            <xs:enumeration value="car"/>
            <xs:enumeration value="motor-cycle"/>
            <xs:enumeration value="construction-vehicle"/>
        </xs:restriction>
    </xs:simpleType>
```

```
    <xs:simpleType name="WeaponType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="small-arm"/>
            <xs:enumeration value="rpg"/>
            <xs:enumeration value="mortar"/>
            <xs:enumeration value="gun"/>
            <xs:enumeration value="rocket-launcher"/>
            <xs:enumeration value="other"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="PrimaryRole">
        <xs:restriction base="xs:string">
            <xs:enumeration value="anti-personnel"/>
            <xs:enumeration value="anti-armour"/>
            <xs:enumeration value="anti-air"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="FlagList">
        <xs:list itemType="Flag"/>
    </xs:simpleType>

    <xs:simpleType name="Flag">
        <xs:restriction base="xs:string">
            <xs:enumeration value="new"/>
            <xs:enumeration value="copy"/>
            <xs:enumeration value="remove"/>
            <xs:enumeration value="user"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```