UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Diseño de planos indoor para aplicaciones móviles
Autores: Nicolás Romagnoli
Director: Dra. Silvia E. Gordillo
Codirector: Dra. Cecilia Challiol
Asesor profesional:
Carrera: Licenciatura en Informática – Plan 1990

Resumen

Investigando el contexto de las aplicaciones de navegación indoor, encontramos que uno de los problemas en este dominio, es la generación de planos navegables. Este trabajo busca desarrollar una herramienta que permita la creación de planos de interiores, como así también la definición de marcadores posicionados espacialmente que permitan posteriormente brindar servicios de navegación para recorrerlo, a través de la definición de "caminos" entre el usuario y el destino seleccionado. El objetivo principal es poder definir el plano con una herramienta visual, así como los marcadores y proveer una forma de exportar estos datos en un archivo XML para que puedan ser usados por una aplicación móvil.

Palabras Claves

Posicionamiento, tags, navegación indoor, plano navegable.

Trabajos Realizados

Se eligió una herramienta dedicada al diseño de planos de interiores (SH3D), y se extendió de manera tal que permita la creación de planos navegables, a partir de nuevas estructuras que permiten definir puntos de interés dentro del plano, para ubicar tanto objetos como al usuario.

Conclusiones

A partir del problema planteado en la introducción, tomamos una herramienta basada en software libre que nos permita adaptar el plano resultante a nuestras necesidades con el menor costo posible, utilizando el código generado por la misma. De esta forma, para dibujar un plano navegable, se puede partir de dibujar un plano convencional con la herramienta, y, posteriormente, adicionar a partir del mismo,las nuevas estructuras de navegación.

Trabajos Futuros

Guardar planos parciales, deshacer y rehacer identificaciones de estructuras, trabajar con estructuras de muchos niveles, agregar un sistema de referencia al plano navegable, dividir paredes que se cruzan entre sí, ampliar datos de las estructuras identificadas.

Fecha de la presentación: DICIEMBRE 2013

AGRADECIMIENTOS

Quiero agradecer a la Dra. Silvia Gordillo y a la Dra. Cecilia Challiol, quienes en todo momento se prestaron a mis consultas y me guiaron durante todo el trabajo con mucha paciencia y sin ningún interés de por medio.

A Cristina, quien día a día me apoya y me acompaña sin condiciones.

A mis amigos, que están desde el inicio de la facultad conmigo y con los que hoy en día aún nos entendemos como la primera vez.

A mi hermana María José, a mis primos Julián, Floreana, Gabriel y José Manuel, a mis tías Miriam, Marta y Ana Julia, a mis tíos Carlitos, Jorge y Sergio, a mis abuelas Tina y Elsa, y a mis abuelos José y Pedro, que están y estuvieron desde SIEMPRE.

Y por último, además de agradecer, le quiero dedicar este trabajo a mis papás, José y Ana. Sin ellos nada hubiera sido posible. Gracias por tenerme paciencia y guiarme de la mejor manera en todo momento. Esto y todo es tanto de ustedes como mío.

Contenido

CAPITULO 1 - INTRODUCCIÓN 7

1.1. MOTIVACIÓN 7

1.2. OBJETIVO 9

1.3. ESTRUCTURA 9

CAPITULO 2 - SWEET HOME 3D 11

2.1. MODELO DE CLASES DE SWEET HOME 3D 11

Clase Wall 13

Clase Room 14

Clase HomePieceOfFurniture 15

Clase HomeDoorOrWindow 15

Interfaz Selectable 16

2.2. COMO USAR SWEET HOME 3D 19

Creación de Paredes 21

Creación de Habitaciones 23

Inserción de elementos al plano 26

Importando elementos personalizados 28

2.3. CÓMO EXTENDER SWEET HOME 3D 29

CAPITULO 3 – DECISIONES DE DISEÑO DE NUESTRA APLICACIÓN 31

3.1. INTRODUCCIÓN 31

3.2. MODELO EXTENDIDO 32

3.3. MODELO FINAL 36

3.4. REQUISITOS PARA LA INSTANCIACIÓN DEL MODELO 38

CAPITULO 4 - IMPLEMENTACIÓN DEL PROTOTIPO 45

4.1. ACCIONES RELACIONADAS A LA IDENTIFICACION DEL TERRENO 47

Acción AsociarTerreno 47

Acción VerTerreno 48

Acción DesasociarTerreno 49

4.2. ACCIONES RELACIONADAS A LA IDENTIFICACIÓN DE ESPACIOS 50

Acción GenerarEspacio 50

Acción VerEspacioPadre 51

Acción EliminarEspacio 53

4.3. ACCIONES RELACIONADAS A LA IDENTIFICACION DE ACCESOS 55

Acción AsociarAccesos 55

Acción VerAccesos ... 56
Acción VerParedDeAcceso ... 57
Acción DesasociarAccesos ... 59
Acción VaciarAccesos ... 60
4.4. ACCIONES RELACIONADAS A LA IDENTIFICACION DE TAGS ... 61
Acción AsociarTags ... 61
Acción VerTags ... 63
Acción VerEspacioDeTag ... 64
Acción DesasociarTags ... 66
Acción VaciarTags ... 67
4.5. ACCIONES RELACIONADAS A LA EXPORTACIÓN DE LAS IDENTIFICACIONES ... 68
Acción XMLExport ... 68
Clase Exporter y su método estático *Export()* ... 69
CAPITULO 5 - USO DE LA HERRAMIENTA DESARROLLADA ... 77
CAPITULO 6 - CONCLUSIONES Y TRABAJOS A FUTURO ... 89
BIBLIOGRAFÍA ... 91
ANEXO A – PROTOTIPO ... 93
ANEXO A .1 - EXPLOTACIÓN DE MÉTODOS ... 93
Explotación del método *asociarTerreno(Room habitacion)* de la clase TerrenoContainer ... 93
Explotación del método *asociarEspacio(List<Wall> paredes)* de la clase EspaciosContainer ... 94
Explotación del método *desasociarEspacio(List<Wall> paredes)* de la clase EspaciosContainer ... 98
Explotación del método *asociarAcceso(Object acceso)* de la clase AccesosContainer ... 100
Explotación del método *desasociarAcceso(Object acceso)* de la clase AccesosContainer ... 103
Explotación del método *asociarTag(Object tag)* de la clase TagsContainer ... 104
Explotación del método *desasociarTag(Object tag)* de la clase TagsContainer ... 106
Explotación del método *asociarObstaculosAutomaticamente()* de la clase ObstaculosContainer ... 107
Explotación del método estático *generarParedes()* de la clase Exporter ... 109
Explotación del método estático *generarObstaculos()* de la clase Exporter ... 111
Explotación del método estático *generarTags()* de la clase Exporter ... 113
Explotación del método estático *generarAccesos()* de la clase Exporter ... 115
ANEXO A.2. ALGUNOS COMPORTAMIENTOS EXTRAS ... 117
Almacenamiento de las instancias del modelo extendido ... 117
Control de cambios ... 117

Listeners.TagListener 119
Listeners.AccesoListener 120
Listeners.EspacioListener 120
Listeners.TerrenoListener 121
Listeners.HomeListener 122
ANEXO B - EJEMPLO 123
ANEXO B.1. CÓDIGO XML RESULTANTE DE LA EXPORTACIÓN DE NUESTRO MODELO 123

CAPITULO 1 - INTRODUCCIÓN

1.1. MOTIVACIÓN

Se conoce como aplicación móvil al tipo de aplicación que correrá en dispositivos móviles, tales como teléfonos celulares o en tablets. En nuestro caso en particular, llamaremos a una aplicación móvil, cuando cumple esta condición, pero además, la aplicación es capaz de brindar servicios dependientes de la ubicación del usuario (Location-Aware application) [Mobile Application]

En un principio, debido a las limitaciones de dichos dispositivos, se comenzaron desarrollando pequeñas aplicaciones individuales con funciones limitadas y aisladas, tales como juegos y calculadoras, las cuales no eran más que aplicaciones de PC's transportadas a los nuevos dispositivos. Con el correr del tiempo, a medida que estas aplicaciones fueron tornándose mas robustas y complejas, y sumado al avance tecnológico en el área, se comenzó a desarrollar software específico para estos dispositivos, buscando hacer hincapié en las ventajas y desventajas de los mismos.

Haciendo foco en la característica de movilidad y conectividad de los nuevos dispositivos, se ha centrado el interés en aplicaciones de posicionamiento y direccionamiento. Es decir, aplicaciones que puedan ubicar al usuario en un sitio determinado y que además puedan servir de guías en el movimiento del usuario. Para esto, se comienzan a utilizar los conceptos de posicionamiento outdoor e indoor en el desarrollo del software.

Si bien ambos conceptos parten de la misma idea, la cual es la de auto referenciar un objeto en una ubicación geográfica, Rainer Mautz [Mautz, 2008] define que el entorno y el contexto de cada uno de ellos provocan condiciones particulares al momento de su implementación. Según el autor, mientras que los ambientes indoor están acotados, en cuanto a tamaño, a edificios y habitaciones, las capacidades para el posicionamiento outdoor requieren cobertura regional, y en algunos casos hasta global. Por otro lado, las señales utilizadas para el posicionamiento en exteriores (satélites) no son suficientes o no llegan al contexto de interiores, donde la demanda de las mismas es incluso mayor.

Un claro ejemplo de aplicaciones outdoor son las aplicaciones GPS (siglas en inglés para Global Positioning System), las cuales se pueden utilizar en forma eficiente en espacios exteriores, para determinar, tanto la posición de un usuario con un dispositivo móvil, como de diferentes objetos de interés, como por ejemplo un museo o un hotel.

Sin embargo, el sistema GPS no resulta útil cuando se trata de posicionar a un usuario u objetos de interés en espacios interiores (indoor), como por ejemplo dentro de un edificio. Este problema llevó a los investigadores y desarrolladores a crear lo que se conoce como IPS (siglas en inglés para Indoor Positioning System), el cual permite a un usuario direccionarse y posicionarse en, y a través de estos espacios. Dempsey [Dempsey, 2003] define un IPS "como un sistema que continuamente y, en tiempo real, puede determinar la posición de alguien o algo en un espacio físico como un hospital, un gimnasio, etc."

Ambos sistemas (outdoor e indoor) necesitan información de contexto para que la información que obtienen pueda ser visualizada por el usuario de forma clara y útil, en el caso del GPS concretamente la información recabada es en unidades de latitud, longitud, esta información, en general, es procesada de manera de mostrar al usuario un mapa de la zona en que se encuentra (hoy en día los mapas de Google [Google Maps] son los más usados para esto) y en donde se muestra su posición concreta; el mapa constituye, de esta forma, información de contexto que se utiliza para la visualización de la posición.

En el caso de los sistemas indoor, un IPS puede proveer distintos tipos de información requerida por el usuario en una aplicación dependiente de la posición. La posición absoluta del mismo es calculada por algún IPS y, antes de que la misma pueda ser estimada, hace falta que el plano del área en donde se encuentra el usuario esté disponible [Gu et al., 2009].

Sin embargo, el posicionamiento indoor, comparado con el outdoor es más complejo, tanto en términos de los mecanismos de sensado posibles, como desde el punto de vista de la movilidad del usuario. En el primer caso, hay que tener en cuenta que hay paredes, equipamiento, obstáculos que podrían estar interfiriendo en cualquier sistema que trabaje con la propagación de ondas que podrían ser desvirtuadas por estos elementos; en el segundo caso, hay que tener en cuenta la geometría del espacio considerado para la movilidad del usuario.

Así, la información de contexto mínima necesaria es la representación del lugar en donde se quiere posicionar al usuario, de manera de posibilitar la visualización de la información en forma clara. Esto implica, la mayoría de las veces, crear esta representación ad-hoc para cada espacio particular que la aplicación considere.

En términos de representación, la primera idea que aparece es la definición del plano del sitio concreto; sin embargo, el plano por sí solo no alcanza para poder guiar al usuario dentro de un edificio: también se deben definir cuáles son los espacios por los que el usuario puede caminar dentro del edificio, cómo se pueden recorrer esos espacios, establecer la manera de posicionar al usuario, es decir, establecer qué formas de sensar su posición se van a utilizar, y, establecer la manera en que se van a posicionar los objetos de interés. En este caso, el concepto de guía que existe para el posicionamiento outdoor a través de la utilización de mapas, debe ser reconsiderado por varias razones: la definición del espacio específico por donde el usuario se desplazará, diferentes estilos de desplazamiento y restricciones que podrían considerarse en cada caso particular (todos los espacios se pueden recorrer, consideración de obstáculos etc.) y diferentes necesidades de los distintos usuarios. Dependiendo del objetivo, se podrían definir diferentes estrategias para navegar el espacio.

Muchas veces el proceso de representación del espacio es realizado “a mano” para cada lugar de interés, esto hace que las aplicaciones que asisten al usuario en espacios indoor, sean menos frecuentes y más acotadas. Desde este punto de vista, el proceso de creación de la representación es más costosa que en el caso de las aplicaciones outdoor.

En este contexto, la falta de herramientas para el desarrollo y diseño de estos planos “navegables” que puedan ser utilizados por la aplicación, es un obstáculo grave para el desarrollo de las mismas.

Entendemos por plano navegable, en el contexto de una aplicación móvil, no solamente a la posibilidad de representar la geometría del espacio de interés, sino también especificar algún mecanismo de sensado que permita establecer la posición del usuario para poder brindar servicios de guía al usuario.

1.2. OBJETIVO

El objetivo de este trabajo es el desarrollo de una herramienta que facilite la especificación de espacios indoor. Con el fin de cumplir dicho objetivo, se trabajará sobre una solución que permita la creación de planos "navegables" de interiores.

Esta herramienta, utilizará como base un software que permite dibujar un plano y que se extenderá para permitir las siguientes funcionalidades adicionales:

- La referenciación espacial del espacio
- La especificación de los espacios navegables, es decir los lugares por los que el usuario podrá desplazarse.
- La definición de marcadores posicionados espacialmente (en nuestro caso QR) que permitan tanto establecer a posteriori la posición del usuario, como la de diferentes objetos de interés. Estos marcadores constituirán la base para que luego, la aplicación móvil pueda brindar servicios de navegación a través de la definición de "caminos" entre el usuario y el destino seleccionado.
- La herramienta tendrá características visuales por los que podrá ser utilizada por usuarios no expertos.
- Un mecanismo que permita exportar el plano navegable a algún formato standard que pueda ser interpretado por una aplicación para su uso (en nuestro caso a un archivo XML).

1.3. ESTRUCTURA

La tesis está organizada de la siguiente manera:

En el capítulo 2 se presenta el software que utilizaremos como base para nuestra aplicación así también como las herramientas que dispondremos para extender al mismo.

El capítulo 3 describe las decisiones de diseño que tuvimos que tomar para adaptar el contexto y el modelo de datos del software base a las estructuras necesarias para cumplir con nuestro objetivo.

En el capítulo 4 explicamos y detallamos como llevamos a cabo nuestra herramienta y en el capítulo 5 mostramos el funcionamiento de la misma a través de un ejemplo.

CAPITULO 2 - SWEET HOME 3D

Sweet Home 3D (a partir de ahora lo identificaremos como SH3D) [Sweet Home 3D] es un programa de diseño de planos estructurales e interiores, Open Source, el cual permite dibujar planos 2D, con una vista previa 3D, permitiendo agregar en su interior, elementos comúnmente encontrados en casas, edificios y demás estructuras edilicias. SH3D puede ejecutarse en Windows, Mac OS X 10.4/10.8, Linux y Solaris, y está traducido a 22 idiomas diferentes. Al ser un software Open Source, el código del mismo se encuentra disponible en la Web y puede ser extendido para agregarle nueva funcionalidad.

SH3D es un programa que está dirigido a personas que quieren diseñar interiores fácil y rápido, permitiendo dibujar el plano del lugar, para luego arrastrar y colocar elementos en el mismo a partir de un catálogo organizado por categorías. Las principales estructuras que se pueden dibujar en un plano son paredes, habitaciones, puertas, ventanas y elementos (que van desde un aplique de pared a un placard, pasando por diferentes tipos y categorías, e incluso se pueden agregar nuevos modelos diseñados por un usuario).

2.1. MODELO DE CLASES DE SWEET HOME 3D

SH3D se encuentra programado en lenguaje JAVA, por lo que utiliza la clase MAIN **SweetHome3D** para iniciar el software, la cual mantiene información común para todos los planos que se están dibujando a través de una instancia de la clase **UserPreferences**. La clase **SweetHome3D** extiende a la clase abstracta **HomeApplication**, la cual almacena todos los planos abiertos en un momento dado a partir de un conjunto de instancias de la clase **Home**. Las clases mencionadas se pueden apreciar en la Figura 2-1.

Debido a que SH3D permite dibujar varios planos a la vez, hay cierta información que se comparte entre todos estos planos, y que solo es necesaria especificar una vez. Esta información común está representada, como ya se mencionó anteriormente, por la clase **UserPreferences**, la cual mantiene, entre otros datos, la unidad métrica a usar, el catálogo de elementos que podrán ser agregados al plano (agrupados en categorías) así como también los valores que tomarán ciertos atributos de estos al momento de ser agregados al plano. Es importante notar la importancia del catálogo de elementos, debido a que a partir de este, se obtienen todos los posibles elementos que podrán ser agregados a un plano (no incluyen las paredes ni las habitaciones).

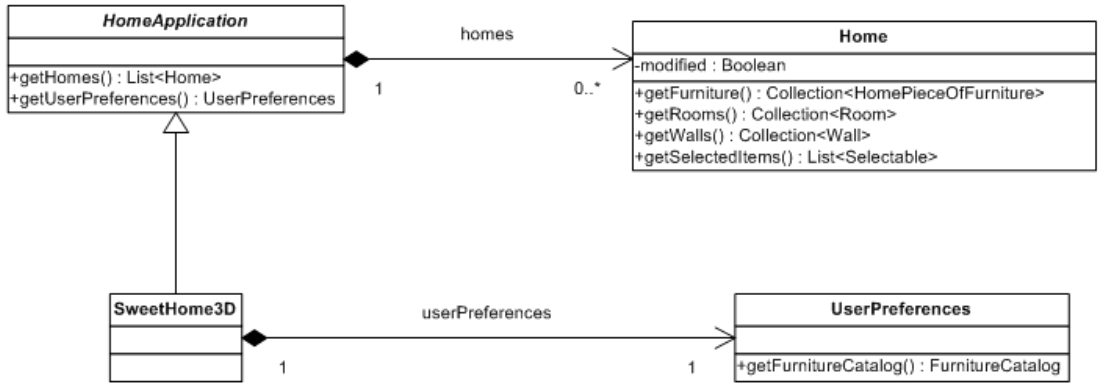


Figura 2-1: Clases principales de SH3D

Veamos en más detalle la clase **Home**. Esta clase se encuentra en el paquete **com.eteks.sweetHome3D.model** y es la más importante de todas, debido a que actúa como un contenedor de todas las características que hacen único a un plano. Las instancias de esta clase almacenan toda la información particular de un y solo un plano, teniendo como principal objetivo guardar las paredes, habitaciones, y elementos (puertas y ventanas, muebles, accesorios) que en él se dibujan. Estos objetos están relacionados con la clase Home, como lo muestra la Figura 2-2*,* y están representados con clases particulares:

- Las paredes están representadas por la clase **Wall,** la cual se encuentra en el paquete **com.eteks.sweetHome3D.model,** y son almacenadas por la clase Home en el atributo *walls*.
- Las habitaciones están representadas por la clase **Room (**la cual se encuentra en el paquete **com.eteks.sweetHome3D.model)** y son almacenadas por la clase Home en el atributo *rooms*.
- Los elementos que representan muebles y accesorios están representados por la clase **HomePieceOfFurniture,** que se encuentra en el paquete **com.eteks.sweetHome3D.model**, y son almacenados por la clase Home en el atributo *furnitures* .
- Los elementos que representan puertas y ventanas están representados por la clase **HomeDoorOrWindow**, la cual extiende de la clase **HomePieceOfFurniture, y** son almacenados por la clase Home en el atributo *furnitures* .

Cada vez que se agrega una instancia de **HomePieceOfFurniture**, **HomeDoorOrWindow, Wall** o **Room** a la clase **Home** (mediante el método *add* correspondiente) se le agrega a la instancia un mecanismo de listener, para que cuando haya alguna modificación en la interfaz de usuario, la instancia se modifique. Al sacar/borrar (mediante el método *delete* correspondiente) alguna de estas instancias, agregadas a la clase **Home,** finaliza el mecanismo de listener asociado a la misma.

El atributo *selectedItems* de la clase **Home** representa todos los elementos seleccionados en el plano en un momento dado. Más adelante se detalla la interface Selectable. A partir del getter de este atributo podemos identificar todos los objetos seleccionados en el plano así como

también a través de su setter seleccionar los objetos que quisiéramos por medio de código y no solamente visual.

Cabe aclarar que el atributo *modified* de la clase Home permite identificar si la instancia fue modificada o no a través de la interfaz de usuario. Cada modificación actualiza este atributo para luego saber si es necesario guardar el proyecto o no.

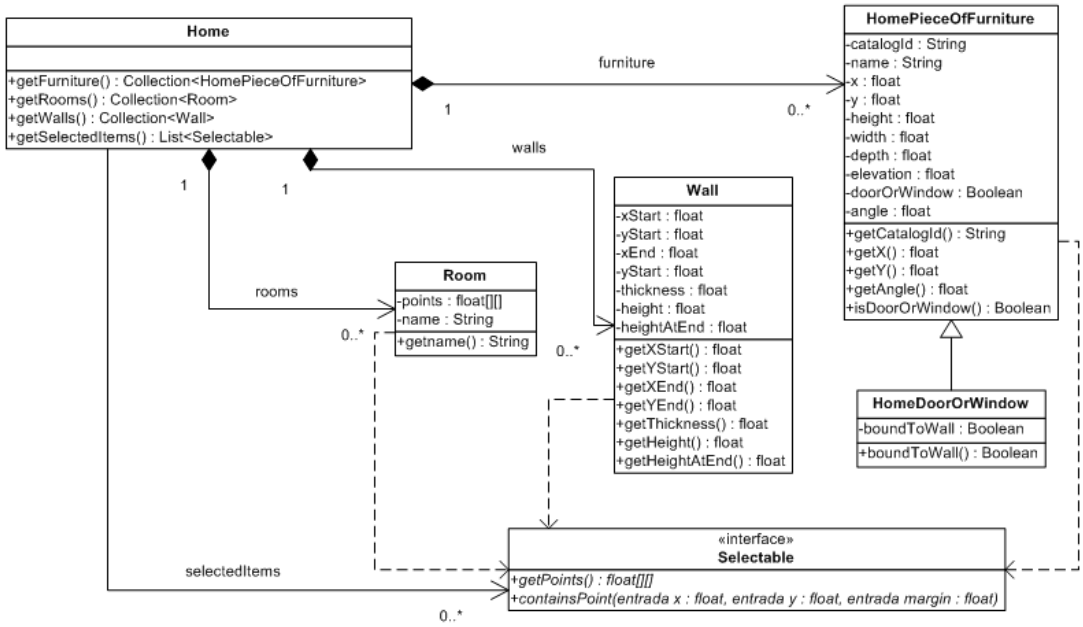


Figura 2-2: Clase Home y sus relaciones

A continuación se especificará con más detalle las Clases e Interfaz presentadas en la Figura 2-2, en el siguiente orden:

- Clase **Wall**
- Clase **Room**
- Clase **HomePieceOfFurniture**
- Clase **HomeDoorOrWindow**
- Interfaz **Selectable**

Clase Wall

Como ya se mencionó anteriormente, esta clase representa las paredes. Cada pared tiene una coordenada inicial y una final, la cual representa el punto inicial y el punto final de la pared, además de una altura inicial y una final. Cada una de las coordenadas están separadas en dos atributos: un X y un Y.
Los principales atributos de esta clase se pueden observar en la Tabla 1.

Nombre	Tipo	Descripción
xStart	float	Representa el punto del eje X de la coordenada donde se encuentra ubicado el inicio de la pared en el plano.
yStart	float	Representa el punto del eje Y de la coordenada donde se encuentra ubicado el inicio de la pared en el plano.
xEnd	float	Representa el punto del eje X de la coordenada donde se encuentra ubicado el fin de la pared en el plano.
yEnd	float	Representa el punto del eje Y de la coordenada donde se encuentra ubicado el fin de la pared en el plano.
height	float	Representa la altura de la pared en el punto inicial.
heightAtEnd	float	Representa la altura de la pared en el punto final.
thickness	float	Representa el espesor de la pared.
wallAtStart	Wall	Referencia a la pared (en caso de haber una) a la cual se une la instancia en su punto inicial.
wallAtEnd	Wall	Referencia a la pared (en caso de haber una) a la cual se une la instancia en su punto final.

Tabla 1: Principales atributos de la clase Wall.

La clase Wall implementa la interfaz **Selectable** (la cual se especificará más adelante), esto genera que la clase deba implementar los métodos asociados a esta interfaz, permitiendo que las paredes puedan ser seleccionadas por el usuario.

Clase Room

Esta clase representa las habitaciones, como ya se mencionó anteriormente. Una habitación contiene el conjunto de vértices (coordenadas X e Y) del polígono que la representa. Los principales atributos de esta clase se pueden observar en la Tablas 2.

Nombre	Tipo	Descripción
name	String	Representa el nombre de la instancia que elija el usuario para el mismo.
points	Float[][]	Conjunto de coordenadas (x, y) que representan los vértices que forman la habitación

Tabla 2: Principales atributos de la clase Room.

La clase Room, al igual que la clase Wall, implementa la interfaz **Selectable,** esto permite que las habitaciones puedan ser seleccionadas por el usuario. Esta clase deberá implementar los métodos especificados por dicha interfaz.

Clase HomePieceOfFurniture

Esta clase, como ya se mencionó anteriormente, representa muebles y accesorios. La Tabla 3 muestra los principales atributos de esta clase.

Nombre	Tipo	Descripción
catalogId	String	Representa el id del tipo de elemento.
name	String	Representa el nombre de la instancia que elija el usuario para el mismo.
x	Float	Representa el punto del eje X de la coordenada donde se encuentra ubicado el centro del elemento en el plano.
y	Float	Representa el punto del eje Y de la coordenada donde se encuentra ubicado el centro del elemento en el plano.
height	Float	Representa la altura del elemento.
width	Float	Representa el ancho del elemento.
depth	Float	Representa la profundidad del elemento.
elevation	Float	Representa la elevación del elemento. Se calcula desde el piso de la Home hasta la base del elemento.
angle	Float	Representa la inclinación del ángulo que forma la base del elemento con el piso de la Home.
doorOrWindow	Boolean	Identifica si el elemento en cuestión es una puerta o ventana. A partir de este atributo podemos definir si se puede castear a la clase específica que representa puertas y ventanas para utilizar sus características especiales.

Tabla 3: Principales atributos de la clase HomePieceOfFurniture.

En particular, el atributo *doorOrWindow*, en esta clase siempre toma el valor FALSE. Este atributo se obtiene al invocar el método *isDoorOrWindow()*.

Esta clase también implementa la interfaz **Selectable**, permitiendo que se puedan seleccionar tanto muebles como accesorios. Para esto, dicha clase implementa los métodos de la interfaz mencionada.

Clase HomeDoorOrWindow

Permite representar puertas y ventanas, como ya se mencionó anteriormente. Es una subclase de **HomePieceOfFurniture**, por lo tanto también es **Selectable**, es decir, las puertas y ventanas se pueden seleccionar. Cabe aclarar, el atributo *doorOrWindoow* que hereda de **HomePieceOfFurniture**, toma valor TRUE para todas las instancias de **HomeDoorOrWindow.**

En la Tabla 4 se pueden apreciar los principales atributos de la clase **HomeDoorOrWindow**.

Nombre	Tipo	Descripción
boundToWall	boolean	Representa si el elemento está sobre una pared.
wallThickness	float	Representa el espesor que debe tener la pared a la cual está pegado el elemento. Esta expresado en un porcentaje a la profundidad del elemento.
wallDistance	float	Representa la distancia a la que debe estar el elemento de la pared. Esta expresado en un porcentaje a la profundidad del mueble.

Tabla 4: principales atributos de la clase HomeDoorOrWindow.

Interfaz Selectable

Esta interfaz define el protocolo de métodos necesario para que un objeto pueda ser seleccionado y manipulado dentro del plano. Las clases que implementen esta interfaz deben definir los métodos especificados por la interfaz.

Como ya se mencionó anteriormente, las clases que representan paredes, elementos y habitaciones implementan esta interfaz, esto se puede apreciar en la Figura 2-3.

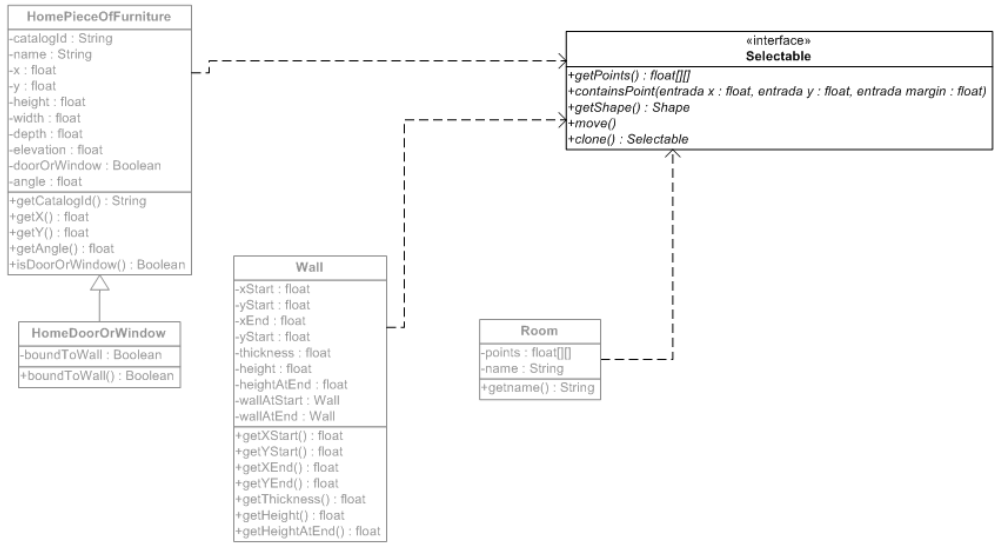


Figura 2-3: Clases que implementan la interface Selectable.

Los métodos más importantes definidos por la interfaz **Selectable** se pueden observar en la Tabla 5. Luego, cada clase que implemente la interfaz, deberá definir el comportamiento para cada método.

Nombre	Parámetros	Tipo Retorno	Descripción
getPoints		Float[][]	Retorna un arreglo con las coordenadas (x, y) que forman al objeto en cuestión.
containsPoint	float x, float y, float margin	boolean	Retorna si el punto (x, y) pasado como parámetro se encuentra dentro de la forma y la posición del objeto
getShape		Shape	Retorna una instancia de la clase Shape (perteneciente a la API de Java) que se corresponde con la forma del objeto
move	Float dx, Float dy		Modifica la posición del objeto la diferencia pasada como parámetro (dx y dy).
clone		Selectable	Duplica el objeto en cuestión.

Tabla 5: principales métodos de la interface Selectable.

De esta manera, quedaron detalladas las principales características de las clases **Wall**, **Room**, **HomePieceofFurniture** y **HomeDoorOrWindow**, como así también la interfaz **Selectable**.

Como se explicó previamente al inicio del capítulo, SH3D permite agregar muchos tipos de elementos, e incluso permite la importación de tipos desde la interfaz de usuario. Para lograr esto, el software provee un catálogo de elementos, el cual, se obtiene a partir de la clase **UserPreferences** (presentada previamente en la Figura 2-1). Cabe aclarar que todos los planos comparten el mismo catálogo de elementos.

El catálogo está representado por la clase **FurnitureCatalog**, que como podemos ver en la Figura 2-4, mantiene un listado de categorías de tipos de elementos, implementadas en la clase **FurnitureCategory.** Esta clase es la encargada de almacenar todos los tipos de elementos organizados en categorías. Para esto, cada categoría tiene su nombre, que será visible en la interfaz de usuario, y una colección de instancias de la clase **CatalogPieceOfFurniture** que representa al tipo de elemento, manteniendo un id que identifica al tipo de elemento unívocamente, así como también todos los valores por defecto que tomará un elemento de este tipo cuando sea agregado al plano.

A partir del catálogo se selecciona un tipo de elemento y se agrega un nuevo objeto de este tipo al plano. Cada vez que esto ocurre, SH3D utiliza los atributos del tipo de elemento seleccionado para instanciar la clase **HomePieceOfFurniture** o su subclase **HomeDoorOrWindow,** los cuales serán los objetos que finalmente se agreguen al plano. Esto es posible gracias a que tanto la clase **HomePieceOfFurniture** como la clase **CatalogPieceOfFurniture** implementan la interface **PieceOfFurniture**.

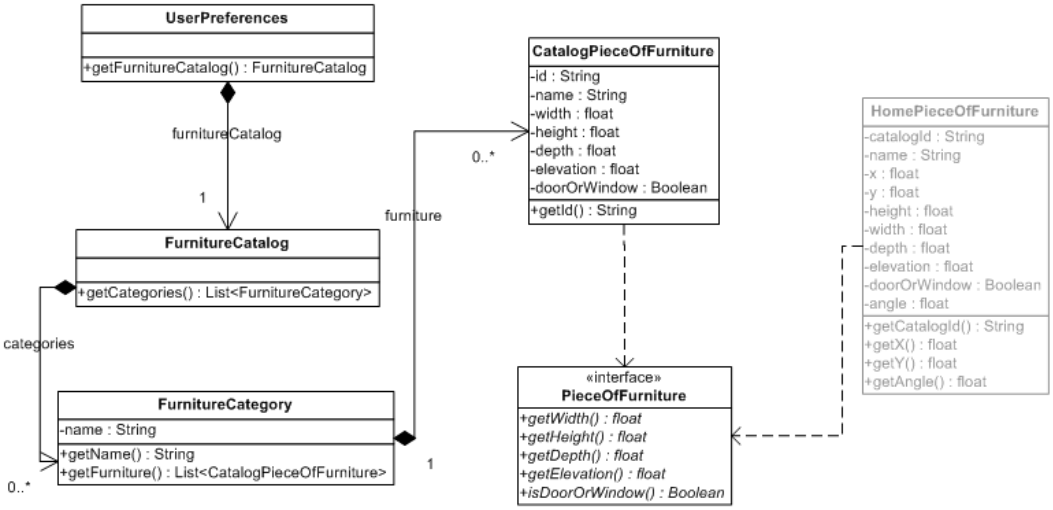


Figura 2-4: Clase relacionadas al catálogo de elementos.

La interfaz **PieceOfFurniture** representa el protocolo de métodos que debe cumplir cualquier clase que representa elementos que se deben agreguen al plano, como es el caso de **HomePieceOfFurniture y CatalogPieceOfFurniture.** Los principales métodos de esta interfaz se detallan en la Tabla 6, cabe aclarar que cada clase que implementa esta interfaz implementará el método acorde a sus características.

Nombre	**Tipo Retorno**	**Descripción**
getName	String	Representa el nombre que se le asigne al objeto de la clase que implementa esta interface.
getHeight	float	Representa la altura que se le asigne al objeto de la clase que implementa esta interface.
getWidth	float	Representa el ancho que se le asigne al objeto de la clase que implementa esta interface.
getDepth	float	Representa la profundidad que se le asigne al objeto de la clase que implementa esta interface.
getElevation	float	Representa la elevación del piso que se le asigne al objeto de la clase que implementa esta interface.
isDoorOrWindow	boolean	Diferencia en dos tipos a los objetos de las clases que implementan esta interface., indicándonos si es una puerta o ventana.

Tabla 6: Principales métodos definidos por la interface PieceOfFurniture.

2.2. COMO USAR SWEET HOME 3D

A continuación especificaremos y explicaremos las principales funcionalidades del software.

Al iniciar el software SH3D se puede ver la pantalla principal dividida en 5 secciones, como se muestra en la Figura 2-5.

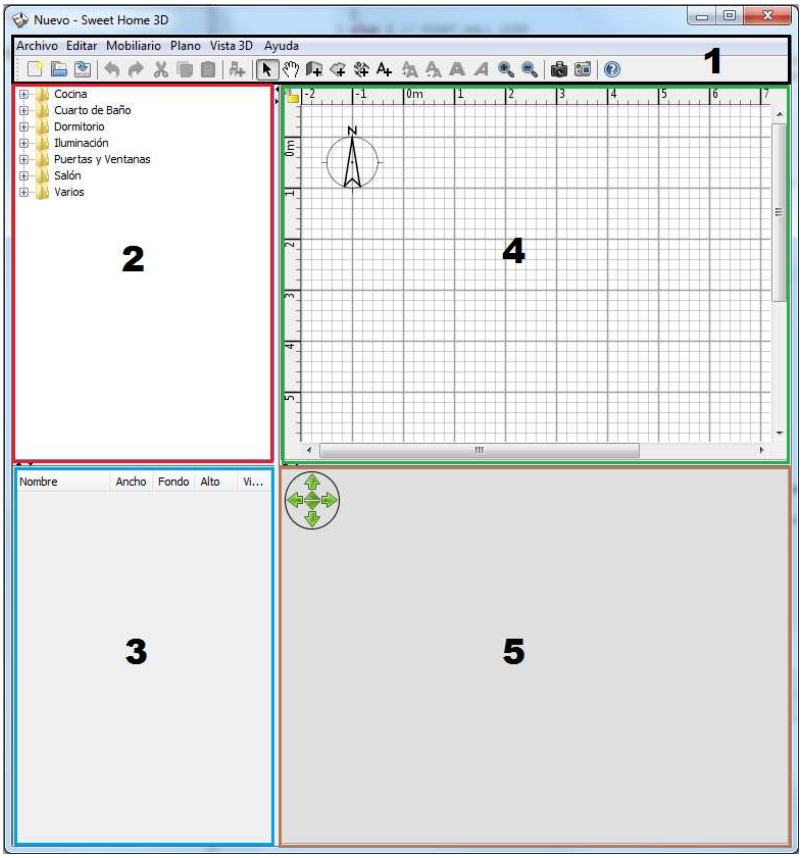


Figura 2-5: Interfaz de Usuario de SH3D.

A continuación se especifican cada una de las Secciones presentadas en la Figura 2-5.

- Sección 1: Un menú + barra de herramientas, el cual provee las distintas acciones que se pueden ejecutar, así como la opción de agregar paredes y habitaciones.
- Sección 2: Un catálogo de elementos, el cual identifica todos los elementos que podrán ser agregados al plano, organizado en categorías, para una mejor distribución.
- Sección 3: La lista de elementos insertados en el plano, muestra todos los elementos agregados en el plano hasta el momento, donde se muestra el nombre, tamaño y otras características.

- Sección 4: La vista 2D, muestra el plano hecho hasta el momento visto desde arriba. En este panel se agregan todas las componentes del plano.

- Sección 5: La vista 3D, muestra la vista 2D volcada en tres dimensiones. Permite observarla desde distintos ángulos.

Al iniciar la aplicación, debemos configurar las preferencias de SH3D. Estos valores quedarán preestablecidos para este y futuros uso de la aplicación hasta que se vuelvan a cambiar. Para esto se debe seleccionar del menú *Archivo->Preferencias*, y se abrirá una ventana como lo muestra la Figura 2-6, mediante la cual podemos establecer los valores para ciertos atributos, tales como el idioma, la unidad métrica, la vista de catálogos de muebles, la forma de mostrar paredes y sus alturas por defecto, entre otras.

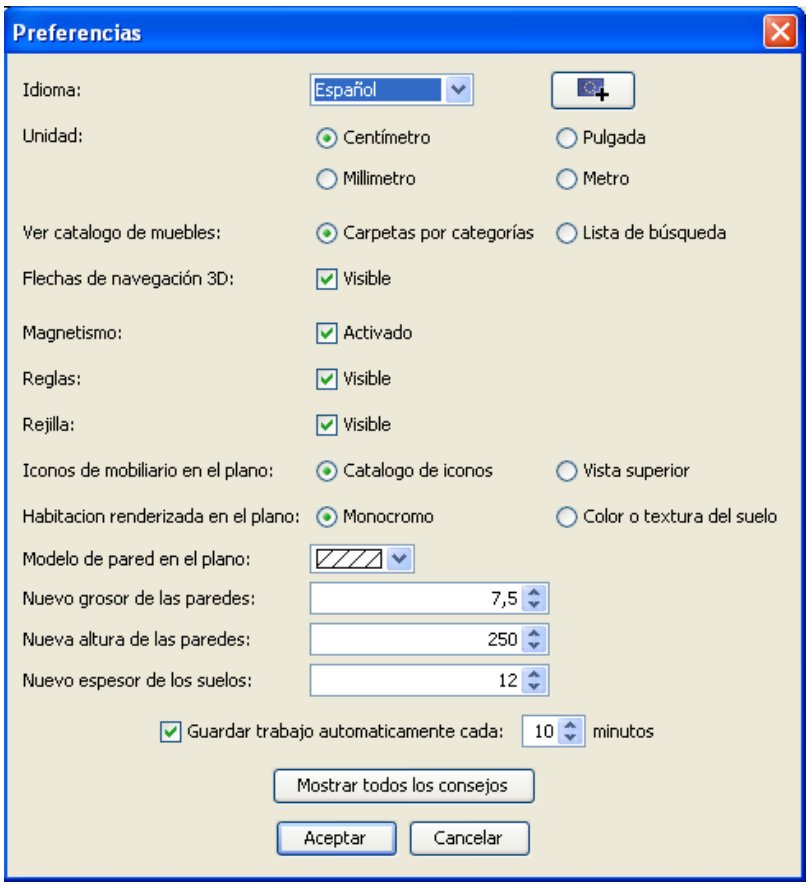


Figura 2-6: Ventana de edición de preferencias.

Una vez establecidos estos atributos se procede a crear un plano nuevo. Para esto se debe seleccionar del menú *Archivo->Nuevo* o bien usar el botón  de la barra de herramientas. Una vez creado el plano se pueden agregar objetos al mismo.

Debido a que existen muchas opciones dentro de SH3D y no todas son relevantes en el transcurso de este documento, sólo nos centraremos en explicar cómo agregar habitaciones, paredes, muebles, puertas y ventanas, que es lo que en el transcurso del mismo utilizaremos.

Creación de Paredes

Para dibujar las paredes se utiliza el botón  que se encuentra en la barra de herramientas. Una vez seleccionado este botón, cada vez que se haga un click sobre el panel de la vista 2D se iniciará o finalizará una nueva pared de la siguiente manera:

- Se hace un click sobre la vista 2D dando inicio a la pared en el punto deseado del plano.
- Se dirige el mouse hacia el punto final de la pared iniciada.
- Una vez determinado el punto final, se opta entre 2 alternativas:
  - haciendo un click sobre la coordenada, se finaliza la pared y se inicia una nueva en el punto final de esta, dejándolas unidas en este punto (Figura 2-7*, en la cual se puede ver cómo fue unida la segunda pared al punto final de la primera).* De esta opción se desprende, a partir del uso reiterado de la misma, la creación de varias paredes unidas entre sí, permitiéndonos entre otras cosas cerrar el final de la última pared al inicio de la primera (Figura 2-8). Vale aclarar que el uso de esta alternativa dejará relacionadas todas las paredes involucradas, por lo que cualquier modificación que involucre la modificación de los extremos de alguna de ellas, afectará a la otra.

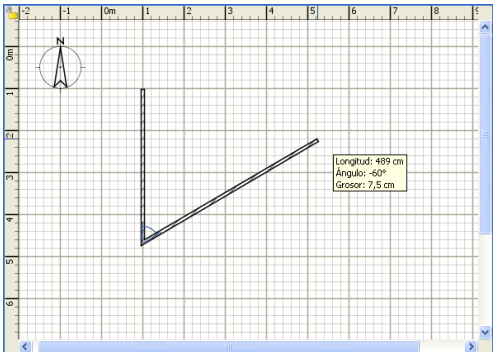


Figura 2-7: Dibujo dos paredes unidas y relacionadas.

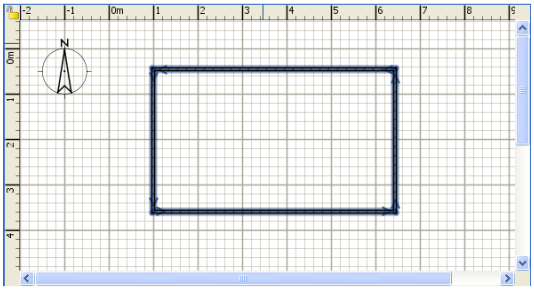


Figura 2-8: Conjunto de paredes unidas y relacionadas.

- haciendo doble click sobre la coordenada, se finaliza la edición de paredes, dejando el mouse libre para ubicarlo en cualquier punto del plano e iniciar una nueva pared donde se lo desee (Figura 2-9). Cabe aclarar que en caso de que se opte por esta alternativa y por el contrario, se quiera unir otra pared a algún extremo, solo bastará con posicionar el mouse en el extremo deseado hasta que aparezca una cruz referenciando el punto de unión e iniciar la nueva pared. En caso de que el punto de unión ya tenga dos paredes relacionadas, la pared se creará desde el extremo seleccionado pero no relacionando las paredes. Esto se debe a que cada pared sólo podrá tener relacionada una y solo una pared en cada extremo, por lo que paredes como lo muestra la Figura 2-10 no están todas relacionadas, sino sólo a lo sumo dos de ellas.

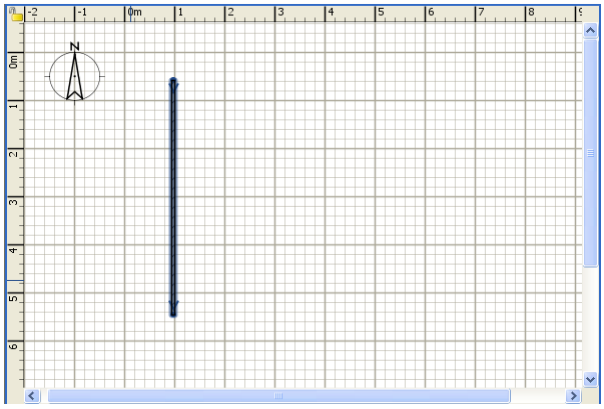


Figura 2-9: Dibujo una pared sola.

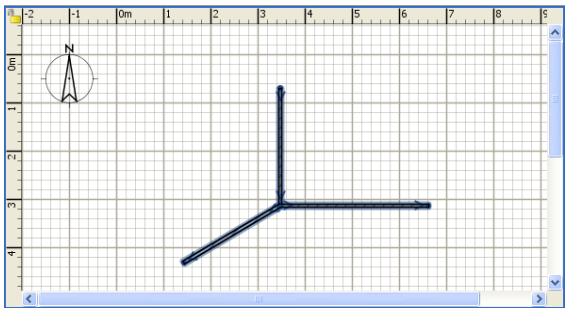


Figura 2-10: Dibujo de paredes unidas pero no asociadas.

Una vez dibujada al menos una pared, existe la posibilidad de editar sus atributos. Para esto se selecciona el botón de la barra de herramientas, y haciendo doble click sobre la pared a editar se nos abre una ventana como lo muestra la Figura 2-11. A partir de la misma podrán ser editados todos sus atributos. Vale aclarar que SH3D no controla que las paredes se superpongan, por lo que paredes superpuestas, cruzadas o desunidas están permitidas por el software.

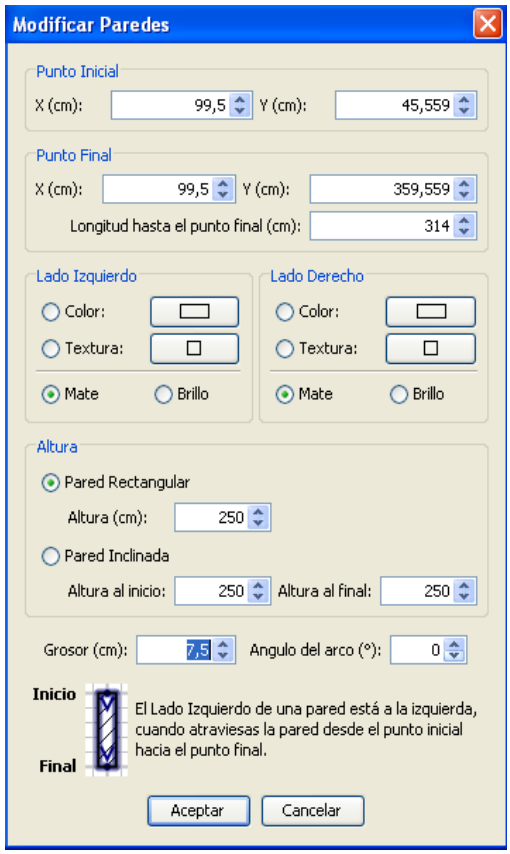


Figura 2-11: Edición de las propiedades de una pared.

Creación de Habitaciones

Para crear una habitación, el proceso es muy similar a la creación de paredes. Hacemos click en el botón  de la barra de herramientas. Veamos las dos formas que existen de crearlas:

- Haciendo click en orden (sin importar por cual se empieza) en cada vértice del polígono que forman la habitación, hasta llegar al último, en donde se procederá con un doble click o utilizando la tecla ESC del teclado (resultado final de la creación mostrado en la Figura 2-12). Esta habitación no tiene paredes.

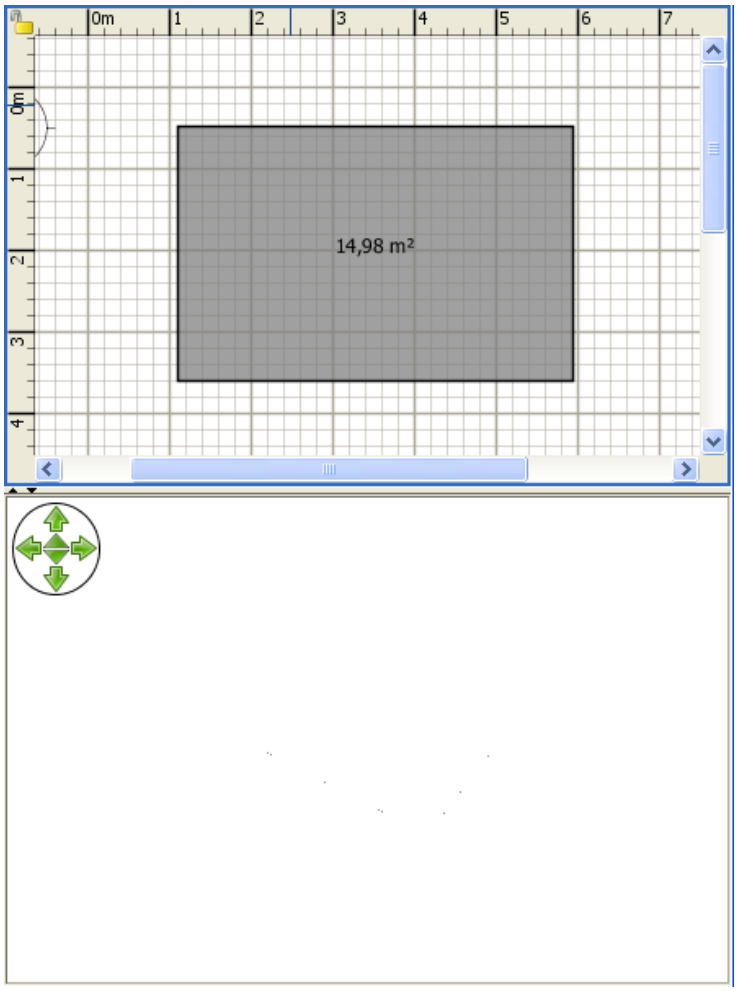


Figura 2-12: Dibujo una habitación sin paredes.

- Haciendo doble click dentro de una superficie cerrada existente. Es decir, dentro de un conjunto de paredes que formen un ciclo (resultado final de la creación mostrado en la Figura 2-13).

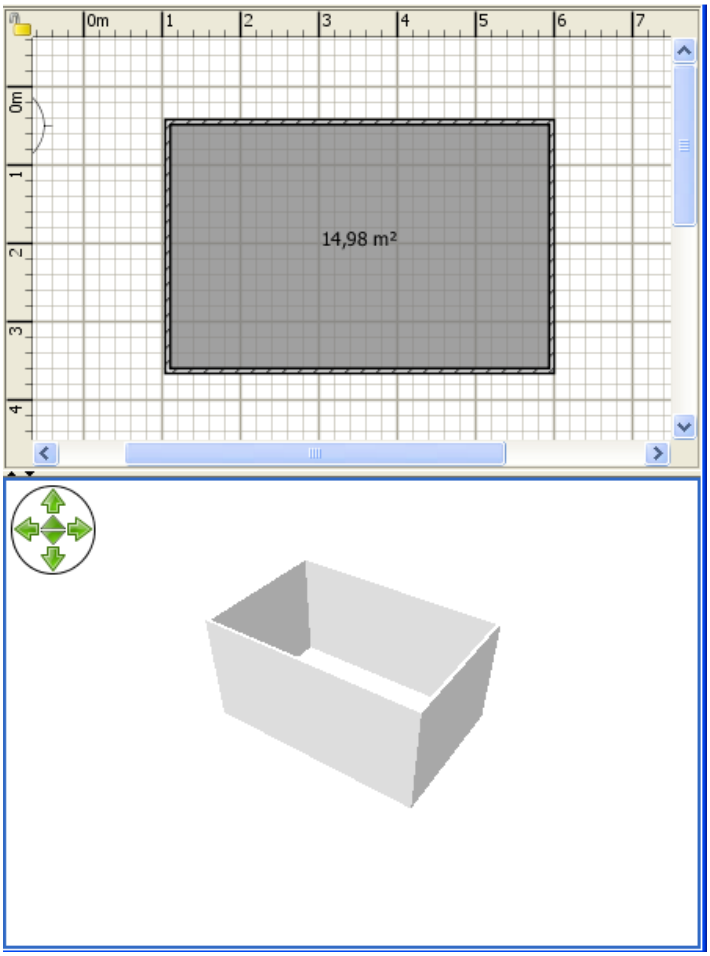


Figura 2-13: Dibujo una habitación que tiene paredes.

Una vez dibujada al menos una habitación, existe la posibilidad de editar sus atributos. Para esto se selecciona el botón de la barra de herramientas, y haciendo doble click sobre la habitación a editar se nos abre una ventana como lo muestra la Figura 2-14. A partir de la misma podrán ser editados todos sus atributos. En caso de querer modificar su forma o su posición, sólo es necesario hacer click sobre la esquina deseada y arrastrar el mouse hasta la nueva posición.

Figura 2-14: Edición de las propiedades de una habitación.

Vale aclarar que SH3D no controla que las habitaciones estén bien formadas, por lo que si las esquinas de la misma no son dibujadas en orden, el software permite dibujar habitaciones con aristas cruzadas, como se muestra en la Figura 2-15.

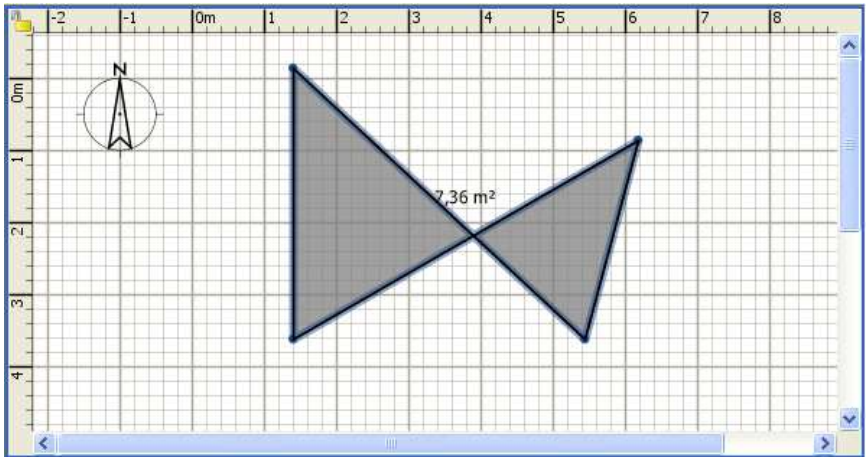


Figura 2-15: Dibujo de una habitación mal formada.

Inserción de elementos al plano

Para añadir elementos al plano se debe seleccionar el tipo de elemento deseado desde la vista del catálogo y arrastrarlo hasta la posición deseada en la vista 2D. Otra opción es seleccionar un elemento del catálogo para luego hacer click en el botón  de la barra de herramientas. Agregar cualquier tipo de mueble que no sea una puerta y ventana se verá reflejado de la misma forma sobre la vista 2D, diferenciándose únicamente en el logo mostrado (ver Figura 2-16, donde se agregaron un armario, una bañera y una cocina como parte de una habitación).

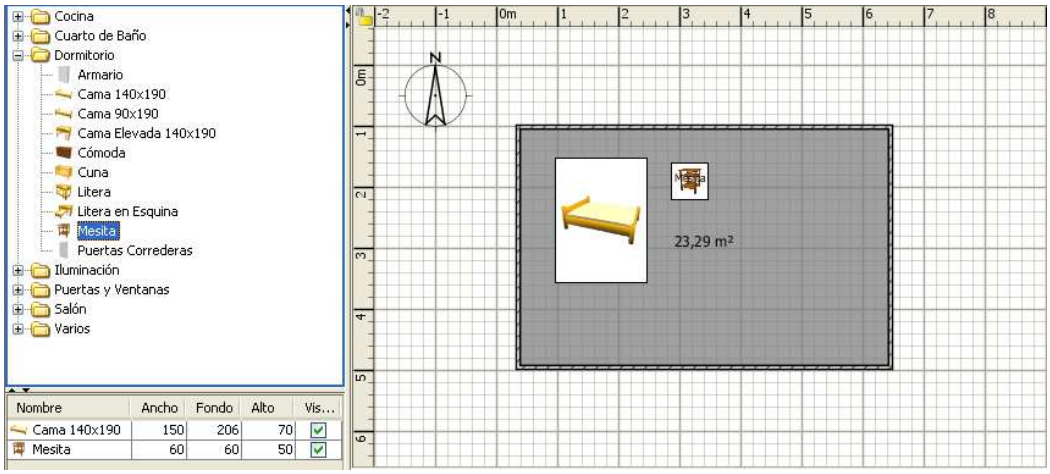


Figura 2-16: Agregar elementos al plano.

Vale aclarar que SH3D no controla que los muebles estén bien ubicados, por lo que un mueble superpuesto a otro o por sobre una pared están permitidos por el software.

Por otro lado, agregar una puerta o ventana al plano, se refleja de manera totalmente diferente, ya que las puertas y ventanas tienen su propio dibujo. La Figura 2-17 muestra que se agregó a la habitación una puerta, una ventana y una ventana corrediza (la cual no tiene abertura como se aprecia en la Figura 2-17).

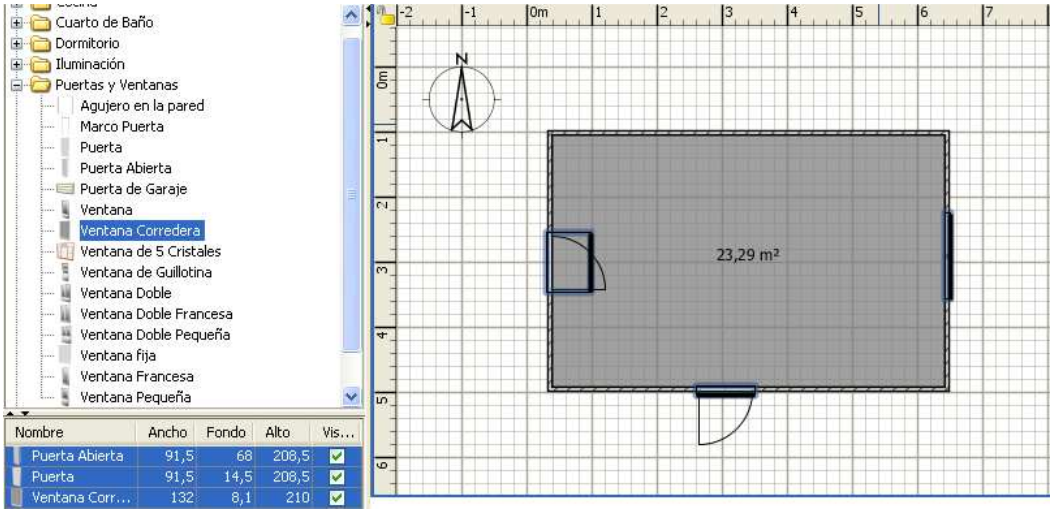


Figura 2-17: Agregar elementos del tipo puerta y ventana al plano

Es importante aclarar que aunque lo correcto es agregar cualquier tipo de abertura sobre una pared, SH3D no controla estas restricciones, por lo que una abertura puede NO tener una pared asociada.

Tanto a puertas como ventanas se les puede modificar sus atributos. Para esto solo hace falta hacer doble click sobre el objeto a editar y se abrirá una ventana como lo muestra la Figura 2-18, mediante la cual se obtiene acceso a los datos y se nos permite modificarlos.

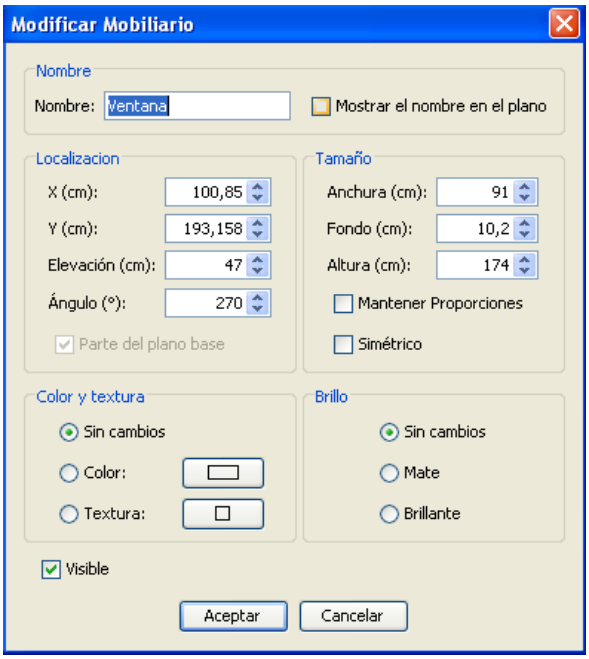


Figura 2-18: Editar las propiedades de los elementos.

Importando elementos personalizados

SH3D da la facilidad de, en el caso de querer agregar un mobiliario que no se encuentre en el catálogo, importarlo. Para esto existen muchos modelos diseñados por otros usuarios de SH3D en la web, los cuales pueden descargarse e importarse. En caso de no existir el modelo deseado, el usuario puede crear el suyo con programas externos y luego utilizarlos en SH3D. Los modelos 3D soportados por SH3D son archivos con formato OBJ, DAE, 3DS, LWS o en un archivo ZIP que contenga un archivo de este tipo. Seleccionando el botón  de la barra de herramienta, se abrirá el asistente para la importación de un nuevo modelo. En Windows y MAC OS también se podrá abrir este asistente arrastrando y soltando el archivo del modelo al SH3D.

El primer paso para importar un nuevo elemento es elegir el archivo con el nuevo modelo (en caso de no haber usado la opción de arrastre, Figura 2-19). Luego se debe orientar el modelo con las flechas del teclado de manera que la vista frontal muestre la cara anterior del modelo 3D (ver Figura 2-20). Luego, puede cambiar, de ser necesario, el nombre, el tamaño, la elevación, el color del modelo importado, y si este modelo es móvil y es una puerta o una ventana (como se muestra en la Figura 2-21). Y por último tiene la opción de girar el modelo con el Mouse para obtener el mejor punto de vista que aparecerá luego en la vista del catálogo (ver Figura 2-22). Una vez finalizada la importación el objeto estará disponible en la vista del catálogo y podrá ser usado en adelante como cualquier otro objeto que esté por defecto.

Figura 2-19: Seleccionar un elemento a importar

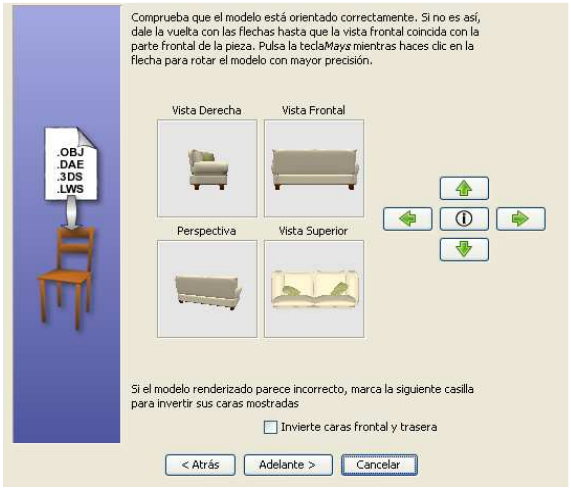


Figura 2-20: Ver el modelo desde distintas vistas

Figura 2-21: Editar los valores predeterminados para el elemento

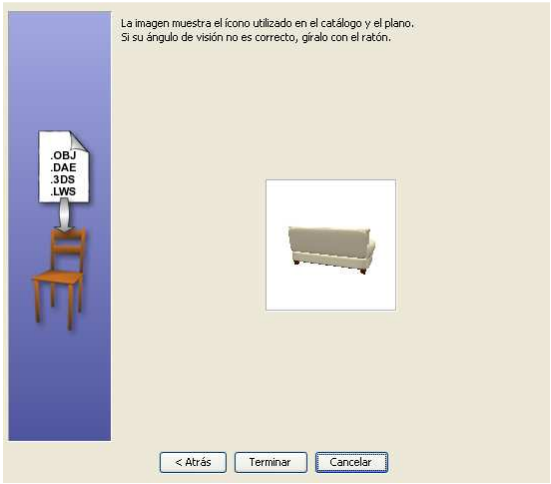


Figura 2-22: Finalizar la importación

En cualquier momento el usuario es capaz de guardar el plano diseñado hasta el momento, el cual se hará en formato .**sh3d.** El archivo resultante no es más que una serialización de las instancias del modelo (de SH3D especificadas en la Sección 2.1), llevada a cabo por el software a través de clases específicas del mismo. Este archivo podrá luego reabrirse para cualquier modificación.

Otra opción para guardar los datos de un plano es la de exportar a un archivo PDF. Vale aclarar que esta opción sólo guardará una imagen del diseño del plano hasta el momento de ejecutarse la acción, por lo que dicho archivo no podrá ser modificado en un futuro.

2.3. CÓMO EXTENDER SWEET HOME 3D

SH3D [SH3D] brinda la posibilidad de expandir la funcionalidad del software sin tocar el código fuente. Para esto, provee una estructura de clases a partir de la cual se deberán programar

paquetes externos llamados **plugins**. Estos últimos deberán guardarse en una carpeta específica del sistema operativo donde corre el software (cada SO tiene su propia carpeta) y deberán seguir una estructura de clases ya definida, la cual es mucho más fácil de entender y mucho más simple que la capa de modelo.

El conjunto de clases involucradas se encuentran agrupadas en el paquete **com.eteks.sweethome3d.plugin** y como se ve reflejado en la Figura 2-23, contiene solo tres clases (la programación de un nuevo plugin solo involucra dos de estas clases, **Plugin** y **PluginAction**).

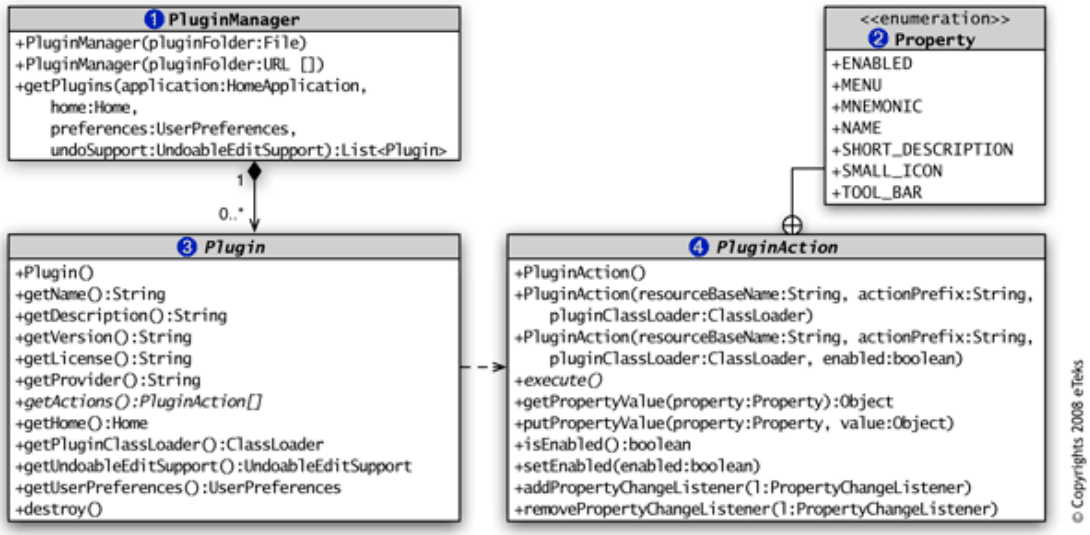


Figura 2-23: Estructura de plugins de SH3D [SH3D].

El mecanismo de plugins se inicia al momento en que el software es cargado, creando una instancia de la clase **PluginManager** (1), que se encarga de examinar la carpeta donde se guardan los plugins. Un plugin no es más que un archivo **.JAR** que contiene en su interior un conjunto de archivos que respetan la estructura especificada en la Figura 2-23. A continuación sólo explicaremos en grandes rasgos como procede SH3D para incorporar los plugin como parte del software que ve el usuario (para más detalle ver [SH3D]).

Cada vez que se inicia un nuevo plano, esta clase instanciará y configurará un objeto de la clase **Plugin** (3) por cada plugin encontrado en la carpeta. Luego, invocará al método ***getActions()*** para recuperar un conjunto de acciones que serán agregadas como ítems al menú o como botones de la barra de herramientas en la interfaz de usuario. Cada acción es una instancia de **PluginAction** (4), la cual es como una clase **javax.swing.Action**, la cual posee un método ***execute()*** además de sus propiedades modificables (a través de la interface **Property** (2)).

CAPITULO 3 – DECISIONES DE DISEÑO DE NUESTRA APLICACIÓN

3.1. INTRODUCCIÓN

Como ya se ha mencionado, se usará como base para este trabajo el SH3D, que provee herramientas para el dibujo de un plano. No obstante, para poder definir un espacio navegable, hacen falta un conjunto de características que no son provistas por esta herramienta y, por lo tanto, constituyen los aportes de esta tesis. En consecuencia, es imprescindible reestructurar ciertos conceptos definidos en el SH3D y definir algunos nuevos.

En términos de las características necesarias para definir un plano como "navegable", se puede mencionar la definición y reconocimiento de:

- un área que delimite el área total en donde se van a incluir los diferentes espacios a ser recorridos por el usuario, llamaremos a esta área Terreno. La característica del terreno entonces es que, cualquier punto transitable debe estar incluido en él.
- áreas (habitaciones, pasillos, etc.) por donde el usuario podrá transitar
- puntos de acceso
- áreas restringidas al usuario (esto podría variar según el perfil del usuario), es decir, por las que uno usuario no puede transitar
- ubicación espacial (en un eje de coordenadas (x, y)) de todos los elementos y áreas definidas y reconocidas en el plano. Esto permitirá, a futuro, transportar y escalar las posiciones de acuerdo al contexto particular de la aplicación que usa el plano transitable.
- objetos que determinarán áreas que no podrán ser transitadas, por lo que cualquier movimiento deberá ser dirigido con la intención de evitar o esquivar los mismos. Estos elementos se denominarán obstáculos
- objetos de interés en los espacios y que sirvan para guiar los posibles movimientos del usuario

Las funcionalidades del SH3D se centran en la posibilidad de dibujar un plano con una interface amigable para el usuario, pero al no ser una herramienta específica para la construcción de planos navegables, no provee las funcionalidades necesarias para contar con las características mencionadas.

Por otra parte el modelo básico de SH3D no es extensible, es decir, no existe la posibilidad de definir clases o subclases (o agregar atributos a clases existentes) que permitan definir las características necesarias en este trabajo. Las extensiones solo se pueden realizar creando un plugin con clases que quedan por fuera del modelo básico de SH3D.

Para alcanzar los objetivos propuestos se trabajó sobre la construcción de dos nuevos modelos (Figura 3-1): el primero, llamado modelo extendido, permite incorporar nuevas clases al

modelo básico del SH3D (a través, como se mencionó del uso de plugins), generando así un modelo que toma la información que nos es útil de la herramienta original, y define nuevos elementos y relaciones. Este modelo es dependiente de la plataforma original (SH3D).El segundo modelo, es independiente de la plataforma, y permite ser exportado para ser utilizado en otras aplicaciones. Este modelo se crea a partir del modelo extendido, y en él se definen clases específicas para la definición de un plano navegable. En la Figura 3-1 se puede ver cómo se relacionan los nuevos modelos.

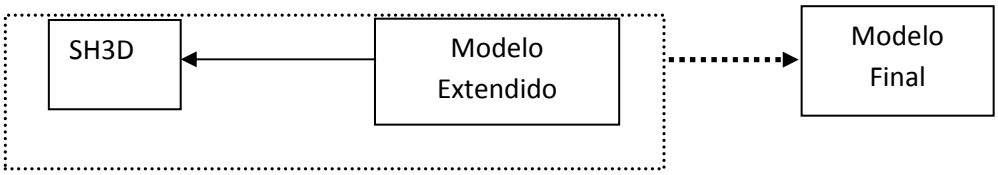


Figura 3-1: relación entre modelos

3.2. MODELO EXTENDIDO

Debido a que el SH3D no provee la posibilidad de manipular nuevas clases en su entorno gráfico, pero si lo permite en su entorno de plugins, creamos el modelo extendido con el objetivo de manipular y guardar objetos del modelo SH3D, organizándolos y agrupándolos acorde a nuestra problemática. Esto nos dará la posibilidad de observar los cambios que se realicen a través de la interfaz gráfica sobre la instanciación del modelo SH3D e ir re-agrupándolos en nuestro modelo a medida que estos se vayan dando.

Como punto de partida determinamos las clases de SH3D que se adaptan mejor a nuestras necesidades. Para mayor claridad, volvemos a reproducir en la Figura 3-2, el corazón de dicho modelo y recordamos cómo se utilizan cada una de ellas:

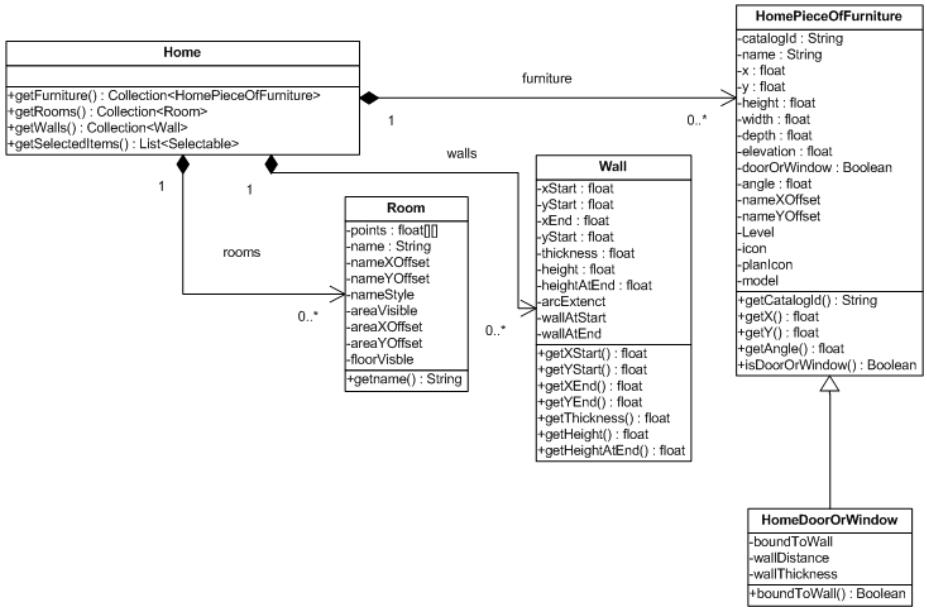


Figura 3-2: Tipos de objetos de SH3D accesibles desde plugins

- La clase **Room**: permite dibujar áreas, denominadas habitaciones, que no son más que un conjunto de coordenadas (x, y) y que representan una habitación en el plano estructural. No existe ninguna restricción por parte del SH3D para la creación de estos elementos, ni de posición ni de forma.

- La clase **Wall**: permite dibujar en el plano paredes, las cuales son líneas con un punto inicial y uno final también expresado en coordenadas (x, y) y provee atributos como el grosor, la altura en los extremos, referenciados sobre el eje Z. No existe ninguna restricción por parte del SH3D para la creación de estos elementos, ni de posición ni de forma.

- La clase **HomeDoorOrWindow:** permite dibujar en el plano puertas, ventanas y aberturas, las cuales son volúmenes con un ancho, una altura, una profundidad y una elevación que determinan su posición en los ejes de coordenadas (x, y, z). Además, proveen la posibilidad de identificar si están posicionadas sobre alguna pared existente. No existe ninguna restricción por parte del SH3D para la creación de estos elementos, ni de posición ni de forma.

- La clase **HomePieceofFurniture**: permite dibujar en el plano cualquier tipo de muebles, los cuales, al igual que las puertas, ventanas y aberturas, son volúmenes con un ancho, una altura, una profundidad y una elevación que determinan su posición en el eje de coordenadas (x, y, z). No existe ninguna restricción por parte del SH3D para la creación de estos elementos, ni de posición ni de forma.

Como se mencionó anteriormente, para crear nuestro modelo, se van a reutilizar ciertas clases y se van a definir nuevas clases y relaciones. En particular:

- El TERRENO se definirá como una instancia de la clase **Room.**

- Cada ESPACIO se creará a partir de al menos tres instancias de la clase **Wall.**

- Cada ACCESO conocerá una instancia de la clase **HomeDoorOrWindow** y otra de la clase **Wall**. Además se debe indicar si el mismo está o no bloqueado.

- Los TAGS y OBSTÁCULOS conocerán una instancia de la clase **HomePieceOfFurniture**. Ambos, deben conocer el espacio que los contiene. Los TAGS deben conocer el código asociado al mismo.

A partir de las consideraciones anteriores, el modelo extendido queda definido como lo muestra la Figura 3-3. Se puede apreciar que las nuevas clases conocen objetos particulares de SH3D.

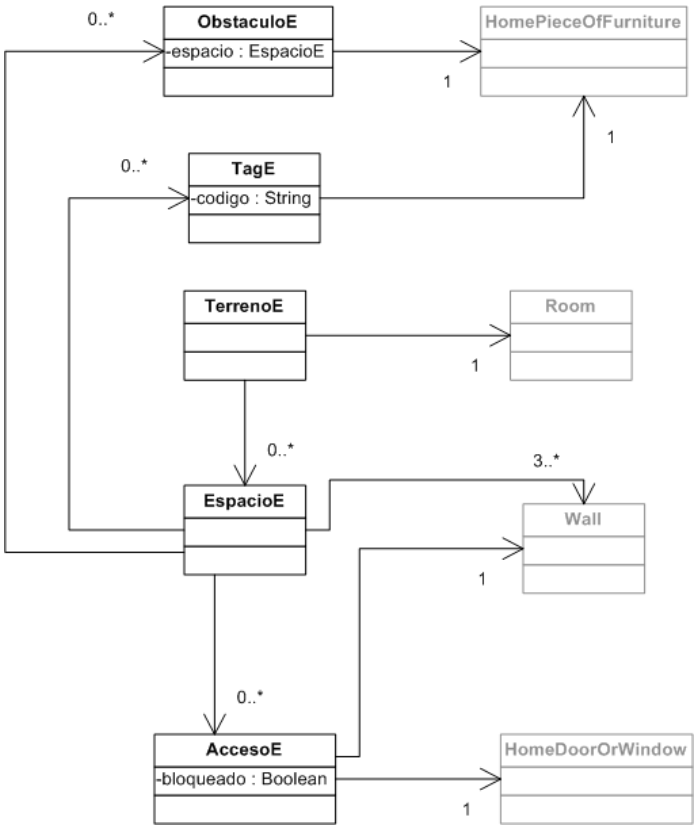


Figura 3-3: Modelo Extendido

La Figura 3-3 se ve claramente la relación entre las estructuras nuevas y las clases del SH3D. La relación entre las clases de nuestro modelo extendido son reducidas, debido a que solo se busca identificar qué clases del modelo SH3D son la base para nuestro modelo final, y poder manipularlos y darles una nueva lógica a los mismos. A continuación se explica la función de cada una de estas clases.

- **Clase TerrenoE**: tiene como función principal representar el área del plano navegable. Toda entidad existente en el mismo deberá estar dentro de él. Solo existirá una estructura de este tipo en el plano y deberá ser capaz de albergar a las demás entidades en su interior. Posee una relación con la clase **Room,** de la cual saldrá la información básica que definirá el contorno del plano.

- **Clase EspacioE (o ambiente):** representa un sub área del terreno con características individuales que lo identifica. Su principal objetivo es la de delimitar zonas transitables condicionalmente, a las cuales solo se podrán acceder cumpliendo ciertos requisitos (no solo de sus propios atributos sino de los atributos de otras estructuras). En un plano podrán existir la cantidad de **ambientes** que el usuario desee, siempre y cuando los mismos no se superpongan, excepto cuando un **ambiente** contenga totalmente a otro, caso en el cual se dará una relación entre ambos **ambientes**, y todos deberán estar incluidos dentro del **terreno**. Se relaciona con la clase Wall del modelo SH3D, que definirán el espacio o ambiente, sin necesidad de definir una clase en el modelo extendido para ellas.

Esto se debe a que la información provista por el modelo SH3D para estos objetos es suficiente para cumplir con el objetivo de nuestro modelo final.

- **Clase AccesoE:** identifica una zona del plano la cual será transitable condicionalmente (dependiendo únicamente de sus propios atributos) y determinará zonas por las que se pasará de un **ambiente** a otro, o del **terreno** a un **ambiente**. Un acceso deberá ubicarse únicamente sobre una posición específica del espacio para cumplir con su funcionalidad (la cual más adelante se explica). El **acceso** quedará directamente relacionado con el/los **espacio/s** en los que influye y se relaciona con la clase **HomeDoorOrWindow** que definirá el acceso propiamente dicho. A su vez conoce a través del atributo *wall* la instancia de la clase **Wall** a la que el acceso pertenece. Esto se debe a que un acceso para ser tal, deberá estar sobre una pared. Esta relación provee esa lógica. Por último, aparece el atributo *bloqueado,* el cual da la lógica pertinente a si el acceso permite el paso por él.

- **Clase TagE:** define objetos que poseen información del contexto y que son usados como sensores de la posición del usuario, por lo tanto, se utilizan además, para determinar los posibles movimientos a realizarse dentro del plano. Con respecto a la información de contexto, ésta depende directamente del contexto en que el plano es utilizado, y puede proveerse en forma de texto, video, página Web, etc. En un plano podrán existir la cantidad de **tags** que el diseñador desee, siempre y cuando se los posiciones dentro de un **ambiente** y no se superpongan con otras entidades. El **tag** quedará relacionado (incluido) con el **ambiente** de menor área que lo contenga. Posee la clase **HomePieceOfFurniture** que define las características de un objeto en el plano. Por otro lado conoce, a través del atributo *espacio,* la instancia de la clase **Espacio** de nuestro modelo extendido al cual el objeto pertenece. Y por último conoce, a través del atributo *código,* la información del contexto que la nueva clase manipula.

- **Clase ObstaculoE**: define una zona o área de un ambiente que no podrá ser transitada nunca. Los **obstáculos** no podrán superponerse con otras entidades y deberán estar posicionados si o si dentro de un **ambiente**. Un **obstáculo** quedará relacionado (incluido) con el **ambiente** de menor área que lo contenga y se relaciona con la clase **HomePieceOfFurniture** que lo define como un objeto dentro del plano. A su vez conoce a través del atributo espacio, la instancia de la clase **Espacio** de nuestro modelo extendido al cual el objeto pertenece.

El objetivo de este modelo extendido es almacenar, identificar y corroborar (al mismo tiempo en que el plano estructural es creado), las relaciones entre los modelos, y que cada elemento relacionado cumpla las precondiciones de las respectivas clases del modelo extendido con las cuales se relacionan (por ejemplo, una instancia de la clase **EspacioE** será creada si y solo si las paredes elegidas para componer el objeto forman un polígono con sus vértices. Por ejemplo, si las instancias de la clase **Wall** se modifican en el plano estructural de manera que dejen de formar un polígono cerrado, la instancia de la clase de nuestro modelo extendido deberá dejar de existir.

Es de destacar, que las relaciones entre las clases de este modelo se establecen a partir de atributos definidos en el SH#D, en particular a través del atributo sh3dObject.

A partir de todo esto, uno de los objetivos de nuestro plugin se centra en identificar cuales objetos del modelo SH3D se utilizarán en el modelo extendido, corroborando los requisitos de cada clase, así como también agregar información extra en caso de ser necesario (si bien solo se agrega información en las clases **Tag** y **Acceso,** esto se podría realizar a futuro en cualquier clase).

3.3. MODELO FINAL

Una vez definido nuestro modelo extendido, es necesario transformar dicho modelo a un modelo independiente del SH3D. Este nuevo modelo se presenta en la Figura 3-4.

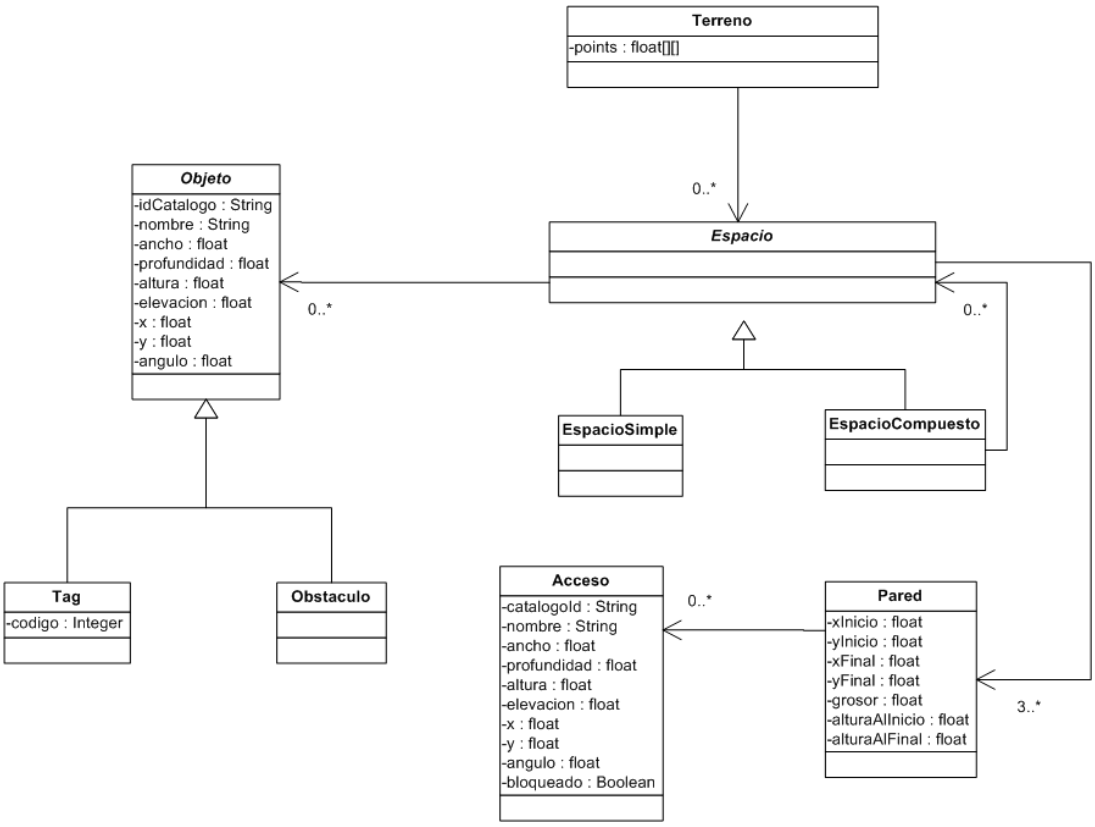


Figura 3-4: Modelo Final

Como se puede ver en la Figura 3-4, las clases de nuestro modelo final, son muy parecidas al modelo extendido, pero se agregan los atributos necesarios para que no dependan de objetos del SH3D. Las clases del modelo final son las siguientes:

- **Clase Terreno:** es la transformación de la clase **TerrenoE** existente en el modelo extendido. Identifica el área de trabajo representada a través de su atributo *points*, el cual no es más que un conjunto de coordenadas (x, y) las cuales delimitan y

controlan el posicionamiento de todos los demás objetos instanciados del modelo final. La instancia de esta clase es la que mantiene la relación con las instancias de la clase **Espacio** creadas a posteriori. Vale aclarar que, debido a la estructura de la clase Espacio y sus subclases, solo conocerá los espacios que no estén incluidos en ningún otro espacio (más adelante se explica la composición de los mismos).

- **Clase Pared:** es la transformación de cada una de las instancias de la clase **Wall** del modelo extendido, que componen al menos una instancia de la clase **EspacioE** del modelo extendido. Sus atributos *xInicio* e *yInicio* forman la coordenada inicial de la pared y sus atributos *XFin e yFin* forman la coordenada final de la misma. Posee además un conjunto de instancias de la clase **Acceso,** los cuales representan las posibles aberturas que tiene la pared, y por ende, serán los puntos de acceso hacia los espacios a los que pertenezca la misma. También se puede apreciar un conjunto de atributos (*grosor, alturaInicio, alturaFin)* que son los que le dan la forma a la pared.

- **Clase abstracta Espacio:** es la transformación de la clase **EspacioE** existente en el modelo extendido. Representa un ambiente e identifica un polígono incluido en el terreno, delimitado por un conjunto de instancias de la clase **Pared** (a través de las coordenadas de inicio y fin de cada una de ellas). Posee un conjunto de instancias de la clase **Obstaculo** y un conjunto de instancias de la clase **Tag,** los cuales representan los objetos que se encuentran dentro del ambiente.

- **Clase EspacioSimple:** subclase de la clase **EspacioE** que identifica ambientes que no poseen ambientes en su interior**.** Si bien no difiere en estructura ni comportamiento de su superclase, se optó por la sub clasificación para un mejor entendimiento del modelo y para facilitar una futura modificación de comportamiento de la misma.

- **Clase EspacioCompuesto:** subclase de la clase **EspacioE** que identifica ambientes que poseen otros ambientes en su interior (los cuales pueden ser Simples como compuestos)**.** Sus sub ambientes se encuentran almacenados en el atributo *espacios.* La colección de obstáculos y tags solo contendrán los objetos que se encuentran dentro del ambiente pero no dentro de sus sub ambientes.

- **Clase Acceso:** es la transformación de la clase **AccesoE** existente en el modelo extendido. Representan un punto de entrada a un ambiente. Debido a la estructura de los ambientes, un acceso debe conocer una y solo una pared. A partir de esta, el acceso permitirá la entrada a los ambientes a los que pertenece. Posee además un atributo que determina si el acceso está bloqueado, lo que cancelará la entrada por él. Sus otros atributos (*x, y, catalogoId, ancho, profundidad y altura*) dan la forma al acceso y su posicionamiento.

- **Clase abstracta Objeto:** representa cualquier tipo de objeto que puede encontrarse en un ambiente. Conoce su posición en el terreno a partir de sus atributos *x* e *y*, el tipo de objeto que es y su forma a partir de los atributos

idCatalogo, ancho, profundidad y altura), su elevación a partir del atributo *elevación* y el ángulo que forma su base con el piso a través del atributo *angulo*.

- **Clase Obstaculo:** es la transformación de la clase **ObstaculoE** existente en el modelo extendido. Subclase de la clase **Objeto** que identifica objetos que se interponen en el recorrido de un ambiente**.** Si bien no difiere en estructura ni comportamiento de su superclase, se optó por la sub clasificación para un mejor entendimiento del modelo y para facilitar una futura modificación de comportamiento de la misma.

- **Clase Tag:** es la transformación de la clase **TagE** existente en el modelo extendido. Subclase de la clase **Objeto** que representa los tags de nuestro sistema. Su función principal es la de proveer información guardada en una base de datos a partir de un código (atributo *codigo*).

La mayoría de las relaciones de este nuevo modelo se obtienen automáticamente del modelo extendido a partir de sus atributos, excepto en las relaciones entre espacios, las cuales se calculan al momento de la creación de las instancias del nuevo modelo.

La instanciación de este modelo es el punto final de nuestra aplicación, dando como resultado un plano transitable e independiente de los otros dos modelos. Esta instanciación es la que será exportada para su futuro uso en aplicaciones específicas.

3.4. REQUISITOS PARA LA INSTANCIACIÓN DEL MODELO

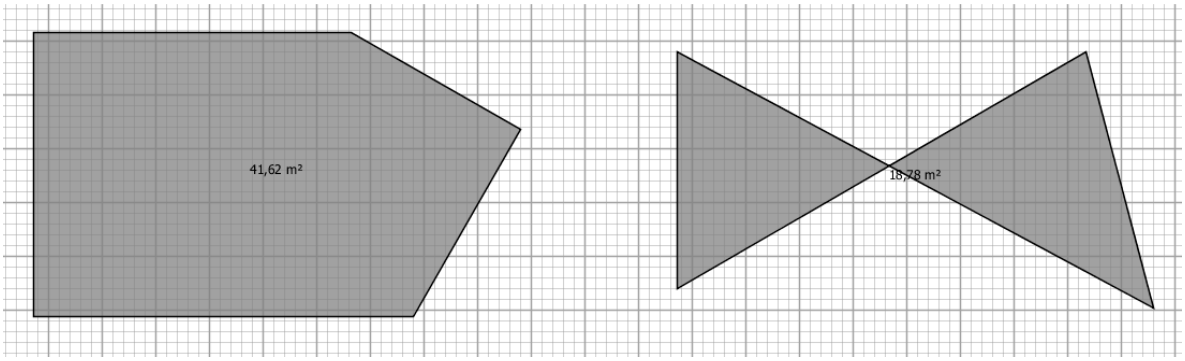
Una vez definido las estructuras y la transformación que se deben dar sobre objetos del SH3D, necesitamos determinar cuáles de estos serán participes del proceso. Para esto identificamos los requisitos que deben cumplir.

Partiendo de esta base, comenzamos con el **reconocimiento del terreno**. La función principal del terreno es la de delimitar todo el contenido del plano dentro de un área. Esta puede ser representada por un polígono, donde cada vértice es una coordenada en el espacio. Como se aprecia en la Figura 3-2, SH3D permite recuperar las instancias de la clase **Room**, los cuales representan las habitaciones del plano. Cada una de ellas tiene entre sus atributos el conjunto de coordenadas que la forma, por lo que las mismas son una posible representación de un terreno.

Debido a que la idea inicial era reconocer automáticamente el terreno a partir del plano dibujado por el SH3D, intentamos reconocer la habitación que cumpla ese rol. Un plano estructural debe permitir la existencia de varias habitaciones por lo que SH3D nos da la posibilidad de crear varias instancias de la clase **Room,** lo que imposibilita la automatización del reconocimiento del terreno.

Para solucionar este problema, se optó por hacer el reconocimiento del terreno manualmente, seleccionando en el plano la habitación que cumplirá dicho rol. Como se muestra en la Figura 3-5, no todas las habitaciones podrán identificar correctamente a un terreno. Solo podrán

cumplir ese rol las habitaciones que no formen polígonos complejos (se conoce como polígonos complejo al polígono que se cruza consigo mismo, es decir, que alguno de sus bordes se cruza con más de dos lados).



habitación apta para identificar un terreno **habitación NO apta para identificar un terreno**

Figura 3-5: Condiciones del terreno

Una vez definido el terreno, encaramos de la misma manera el reconocimiento de espacios. Para la representación de dichos objetos surgen dos posibilidades dentro de los objetos de SH3D por cada uno de ellos:

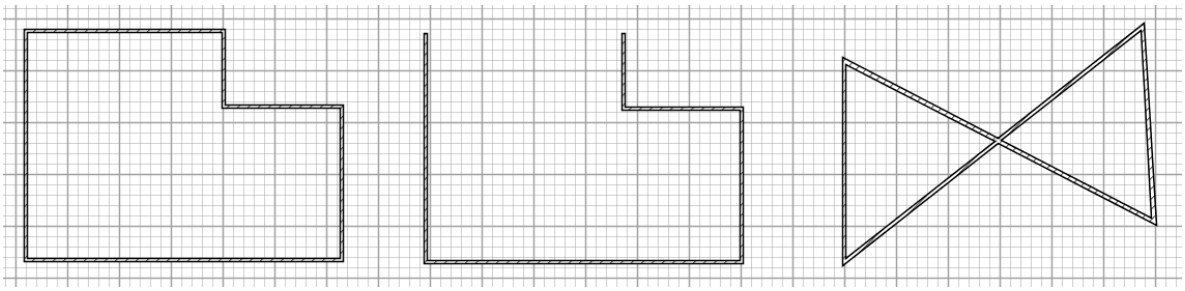
- Una instancia de la clase **Room.**
- Un conjunto de instancias de la clase **Wall.**

Ante estas opciones, debíamos evaluar los pro y los contra de cada una.

Utilizar una instancia de la clase **Room** permite el reconocimiento automático de todos los espacios en el plano (exceptuando al que identifica al terreno), por lo que a simple vista es tentador su uso como tal. En cambio, el uso de un conjunto de instancias de la clase **Wall**, nos obliga a reconocer manualmente cada uno de los espacios, seleccionando el conjunto en cuestión en el plano y dejando almacenada la identificación. Por otro lado, un conjunto de paredes identifica semántica y gráficamente mejor a un espacio que una habitación.

A simple vista, la opción más cómoda es la primera, pero viendo a posteriori la necesidad de reconocer accesos (y la opción utilizada para esto se acomodaba mejor con el reconocimiento de ambientes a través de un conjunto de paredes) y dándole prioridad a la representación gráfica, se optó por la segunda.

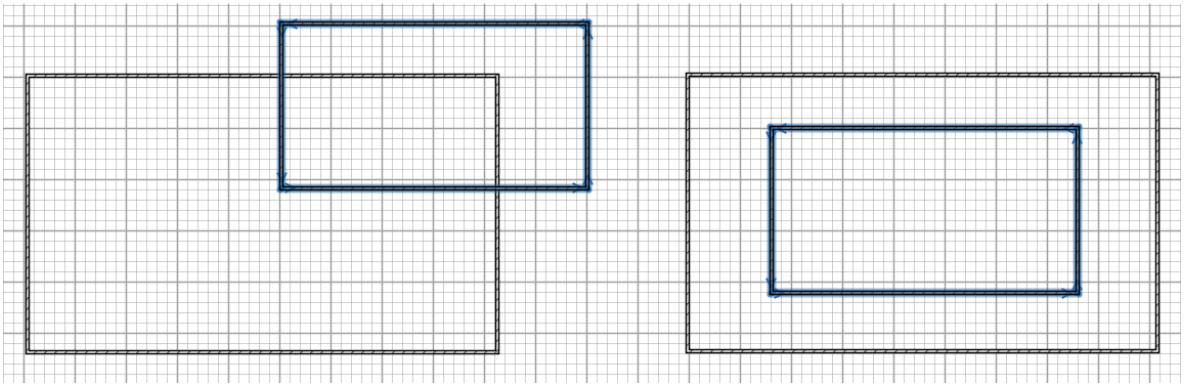
Además de definir que los espacios están compuestos por un conjunto de paredes, es necesario determinar la disposición de las paredes para que formen un espacio correcto. Para cumplir este objetivo, las paredes deben formar un polígono cerrado y NO complejo. La Figura 3-6 muestra ejemplos de espacios bien definidos y espacios no definibles en nuestra herramienta. Vale aclarar que el último espacio de la Figura 3-6 no es UN espacio correcto, pero si se dividen las paredes que se cruzan de la manera correcta, se podrían identificar correctamente.



CORRECTO **INCORRECTO** **INCORRECTO**

Figura 3-6: condiciones de espacios.

Por otro lado, es necesario que los polígonos que forman cada uno de los espacios no se superpongan parcialmente entre sí, como se muestra en la Figura 3-7. La única superposición de ambientes es la inclusión, que es la que determina si un espacio es simple o compuesto (un ambiente que incluya a otros será compuesto y aquel que no incluya a ningún otro espacio será simple),



INCORRECTO **CORRECTO**

Figura 3-7: superposición de espacios

Otro requisito importante a cumplir es que un espacio compuesto no puede compartir paredes con sus sub espacios, tal como se muestra en la Figura 3-8, donde el sub espacio (redondeado en rojo) comparte paredes con el espacio que lo incluye (redondeado en azul). Si esto ocurre, se deberá rever la estructura de los mismos, de manera tal que la nueva distribución quede como lo muestra la Figura 3-8, donde se pueden apreciar dos espacios simples compartiendo paredes (las cuales se muestran en color verde).

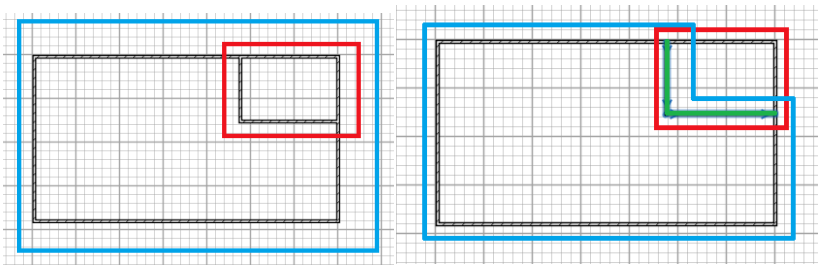


Figura 3-8: espacios mal identificados

Y por último, pero no menos importante, cualquier espacio identificado, sea cual fuere su tipo, debe estar incluido totalmente dentro del terreno identificado, por lo que antes de identificar cualquier espacio deberá existir un terreno identificado (Figura 3-9).

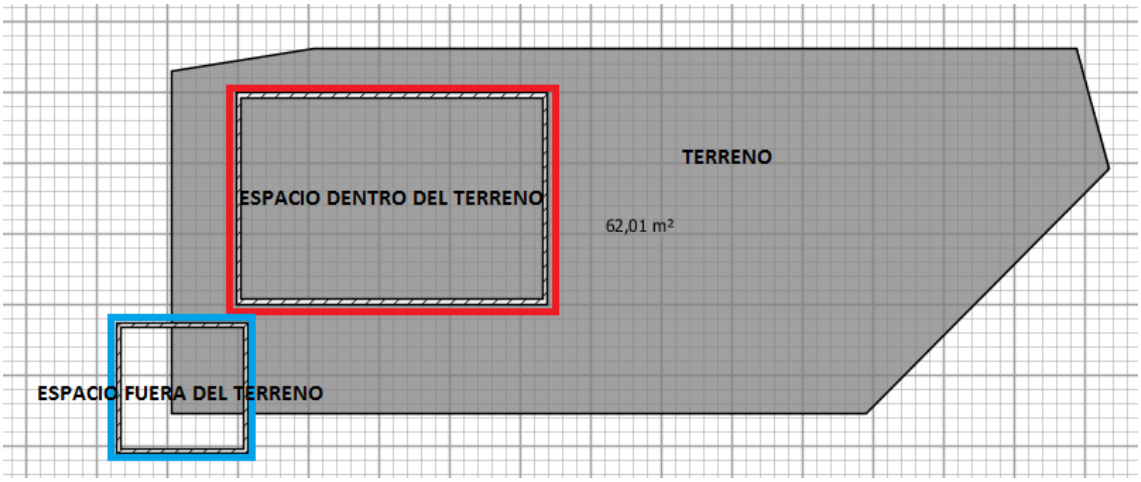


Figura 3-9: espacios fuera y dentro del terreno

Siguiendo estos requisitos, buscamos encontrar un método que identifique la distribución de espacios automáticamente. Si vemos la Figura 3-10, observamos que entre un conjunto de paredes bien ubicadas, existe la posibilidad de identificar más de una distribución de espacios donde ambas cumplen los requisitos, y al estudiar la situación, no encontramos un criterio que permita la elección automática entre ellos. Por ende, creamos la identificación explicita, la cual a partir de la selección de un conjunto de paredes en el plano, se las identifica como un espacio (siempre y cuando cumpla con los requisitos). Remitiéndonos nuevamente a la Figura 3-10, se deberá optar entre alguna de las dos opciones.

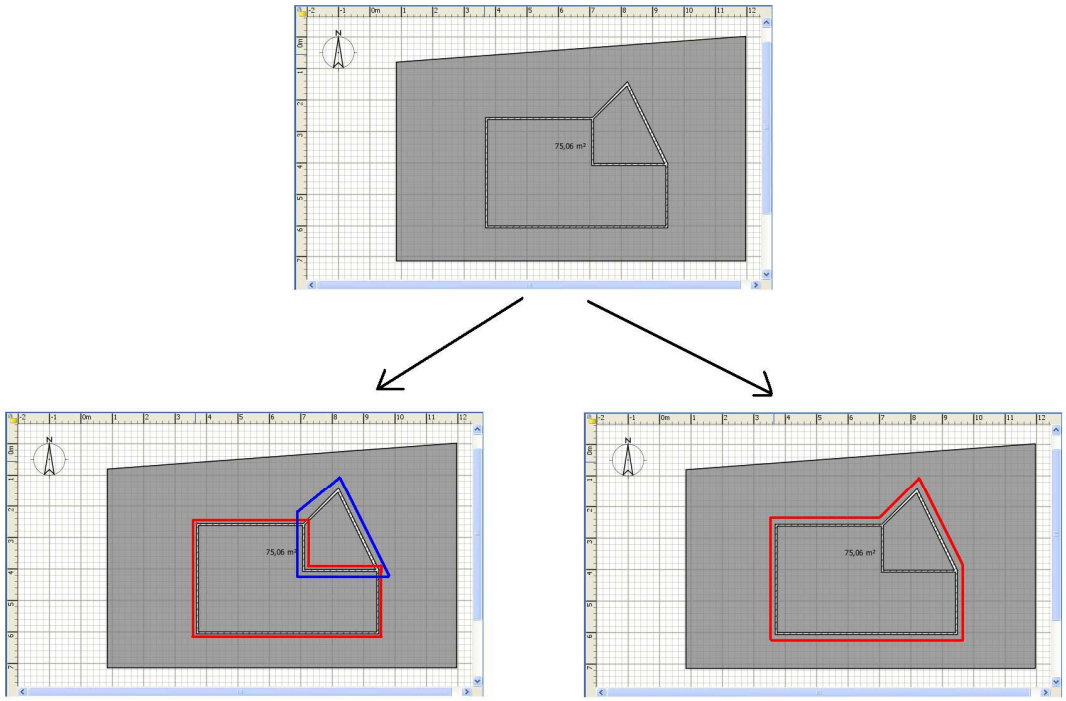


Figura 3-10: posibles identificaciones de espacios

Con respecto a los accesos, SH3D nos provee en su catalogo de muebles, las puertas y ventanas, las cuales al agregarlos al plano, se generan como instancias de la clase **HomeDoorOrWindow**. Dichas instancias conocen si están sobre una pared o no, lo que determina que en el plano se muestren incrustadas en una pared. Debido a que la clase **Home** almacena todos los muebles en el atributo *furniture*, y las puertas y ventanas forman parte de esos objetos, es posible recuperar de forma automática, las puertas y ventanas de nuestro plano y además determinar si están ubicadas sobre paredes que pertenezcan a espacios previamente identificados. Debido a esto la elección de estos objetos como accesos es instantánea.

Dado que estos objetos son la mejor opción para identificar accesos, y se colocan sobre paredes, nos llevó a inclinarnos en el uso de paredes para la identificación de espacios (como se explicó más arriba en el texto). Si bien es posible mediante cálculos identificar si un acceso se encuentra sobre un lado de una habitación, gráficamente el software no está predefinido para ubicar fácilmente una puerta o una ventana sobre una arista de una habitación, por lo que hicimos prevalecer esta cuestión gráfica y semántica por sobre cualquier otra, al momento de elegir la representación de los espacios. Como se mostró anteriormente en la Figura 3-4, los accesos tienen conocimiento de la pared a la que pertenecen, así como las paredes tienen conocimiento de los accesos que posee.

Vale destacar, que si bien es posible identificar automáticamente cuáles son las instancias de esta clase que formarán parte de nuestro modelo, decidimos optar por el reconocimiento manual, debido a que encontramos útil mantener información extra por cada acceso (atributo *bloqueado,* explicado anteriormente en el texto*)*. Reconocer explícitamente un acceso nos

permite cargarle dicha información así como también calcular la pared a la que está unido y relacionarlo en nuestro modelo extendido.

Si bien la decisión de utilizar instancias de la clase **HomeDoorOrWindow** como accesos es la adecuada, no cualquier instancia de esta clase podrá ser identificada con tal fin. Debido a que los accesos determinan entradas a los espacios, es necesario que los accesos estén ubicados sobre paredes que pertenezcan a espacios identificados previamente. Vale aclarar que instancias de la clase **HomeDoorOrWindow** sin una pared asociada o asociados a paredes que no pertenecen a un espacio válido, no podrán ser identificadas como accesos (Figura 3-11).

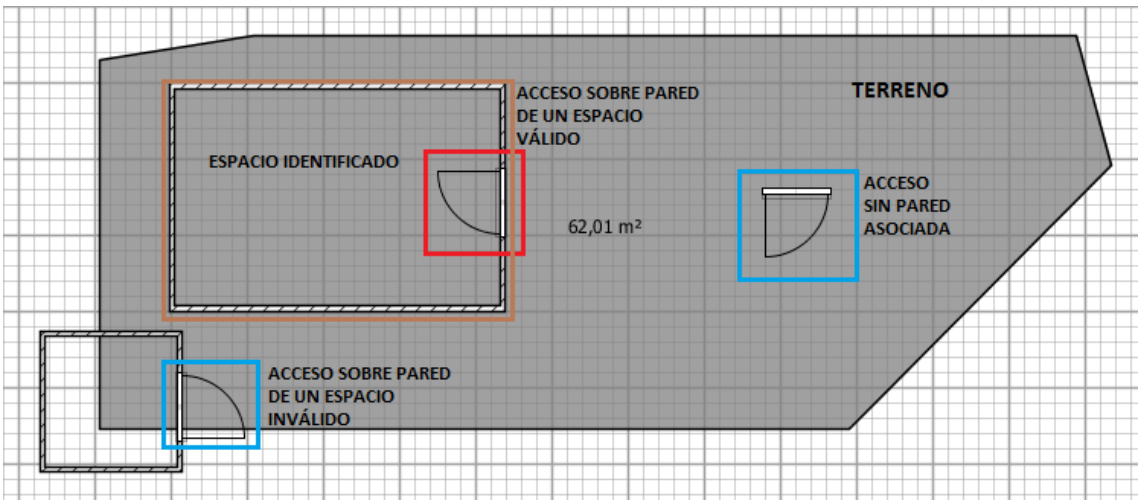


Figura 3-11: accesos válidos e inválidos

Por último llegamos a la identificación de tags y obstáculos. Para ello utilizamos las instancias de la clase **HomePieceOfFurniture** que no sean instancias de su subclase **HomeDoorOrWindow**.

Para la representación de los tags, agregamos al catalogo de muebles de SH3D un nuevo elemento al cual llamamos TAG. Si bien cada vez que se agrega un mueble del catalogo al plano, SH3D crea una instancia de la clase **HomePieceOfFurniture**, es posible determinar a qué tipo de mueble representa cada instancia a partir de su atributo *catalogId.* De esta manera, existe la posibilidad de identificar automáticamente a cual tipo de mueble la instancia representa, y por ende, podríamos identificar automáticamente todos los tags que existen en un plano.

Sin embargo, no se optó por esta opción de reconocimiento automático, ya que, al igual que ocurrió con los accesos, nos es útil mantener cierta información por cada tag. Por lo tanto, se decidió identificarlos manualmente mediante previa selección en el plano, cargándoles la información extra necesaria en ese momento, permitiendo modificarla en cualquier momento. Cabe destacar, que si bien la existencia de un mueble de tipo TAG en el catálogo de SH3D nos ayuda a entender mejor el plano, cualquier elemento, incluso los que no son de este tipo de mueble, podrá ser identificado como tal.

Para la representación de los obstáculos se usó la misma clase que los tags, pero sin la necesidad de agregar ningún nuevo tipo de mueble en el catálogo de SH3D. Se optó por reconocer los obstáculos por descarte, es decir, las instancias de la clase **HomePieceOfFurniture** guardadas en el atributo *furniture* de una instancia de la clase **Home** que no hayan sido identificadas como tags serán automáticamente asociadas como obstáculos.

A diferencia de los obstáculos, la clase **Tag** de nuestro modelo extendido posee el atributo adicional *codigo,* el cual es ingresado explícitamente por nuestra herramienta y determina la información del contexto del tag.

Debido a que tanto los obstáculos como los tags deberán estar incluidos si o si en un espacio válido e identificado previamente (Figura 3-12, si bien se muestran elementos del tipo obstáculo, se cumple lo mismo para los tags), las instancias de la clase mantienen en su atributo *espacio,* el conjunto de paredes reconocidas previamente como un espacio en la que están incluidos.

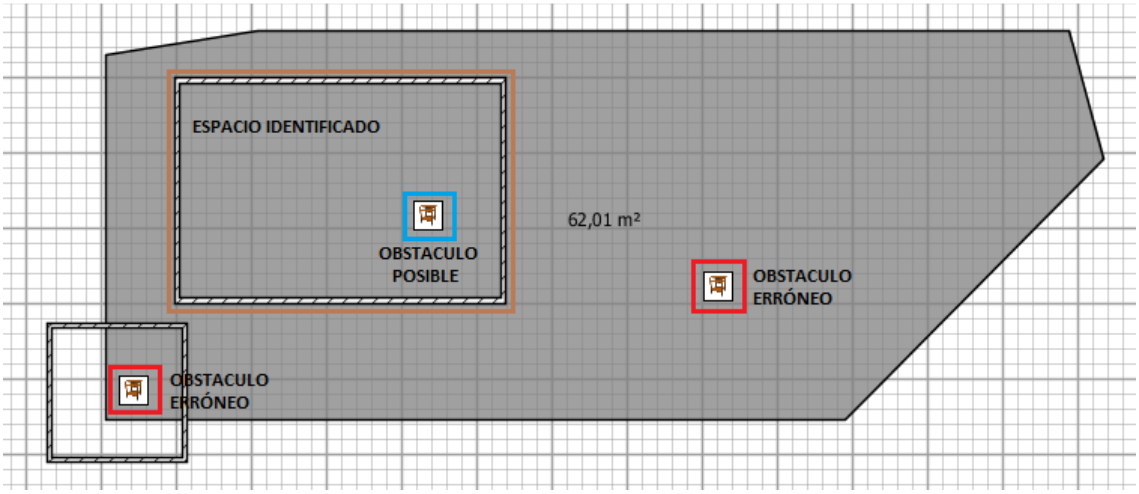


Figura 3-12: obstáculos válidos e inválidos

Cabe destacar además, que para saber cuáles elementos identificarán obstáculos, debemos saber primero cuales identifican tags, la creación de las instancias de la clase **Obstaculo** se hace como paso previo al momento de transformar el modelo extendido al modelo final.

CAPITULO 4 - IMPLEMENTACIÓN DEL PROTOTIPO

Habiendo definido las adaptaciones que requiere el modelo SH3D para cumplir nuestro objetivo, en esta etapa desarrollaremos dichas extensiones. Para esto utilizamos el mecanismo de plugins que el software provee.

Si nos remitimos a la Figura 22 del capítulo 2, vemos que, para que una clase se comporte como plugin, debe extender de **com.eteks.sweethome3d.plugin.Plugin** y sobre escribir el método *getActions(),* el cual devuelve un conjunto de instancias de la clase **com.eteks.sweethome3d.plugin.PluginAction**. Cada una de estas instancias implementará una función nueva, la cual será acoplada al software. Como se muestra la Figura 4-1, nuestra clase **CargadorActions** implementará la lógica de Plugin, mientras que cada una de las clases que extienden de **PluginAction** implementa una función nueva.

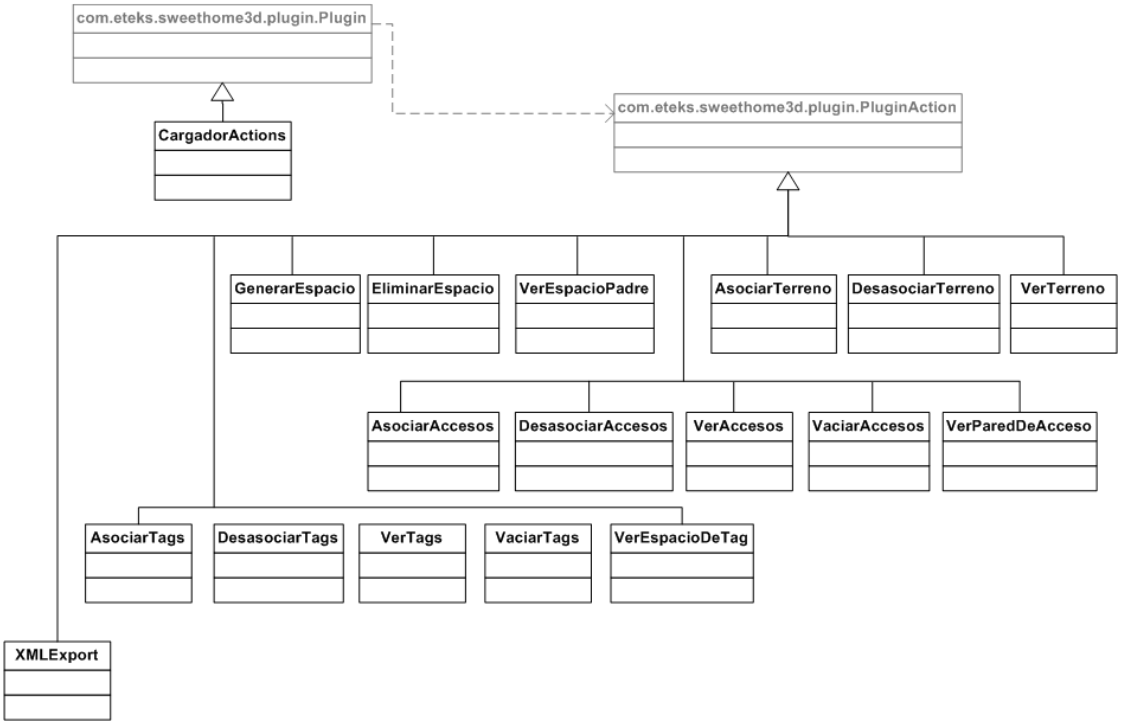


FIGURA 4-1: nuestro plugin y sus acciones.

Para ordenar mejor el código, ubicamos las acciones en un conjunto de paquetes, tal como lo muestra la Tabla 4-1, distribuidas como se observa en la Tabla 4-2.

ACCIONES RELACIONADAS CON	PAQUETE
el TERRENO	Plugins.actions.terreno
los ESPACIOS	Plugins.actions.espacios
los ACCESOS	Plugins.actions.accesos

los TAGS	Plugins.actions.tags
los OBSTACULOS	Plugins.actions
Crear los objetos del nuevo modelo a partir de las identificaciones de objetos	

Tabla 4-1: paquetes y las acciones que implementa cada uno

PAQUETE	ACCION	COMPORTAMIENTO
Plugins.actions.terreno	AsociarTerreno	Identifica y almacena una habitación seleccionada en el plano que representará el terreno.
	DesasociarTerreno	Desasocia la habitación previamente seleccionada como terreno.
	VerTerreno	Selecciona en el plano la habitación previamente identificada como terreno.
Plugins.actions.espacios	GenerarEspacio	Asocia y almacena (en caso de cumplir con los requisitos) un conjunto de paredes seleccionado en el plano como un espacio.
	EliminarEspacio	Desasocia a un conjunto de paredes seleccionadas en el plano como espacio, en caso de haber sido asociado previamente.
	VerEspacioPadre	Muestra, en caso de existir, el espacio inmediato en el que un conjunto de paredes seleccionadas en el plano (si fue previamente asociado como espacio) está incluido.
Plugins.actions.accesos	AsociarAccesos	Asocia y almacena un conjunto de puertas y ventanas seleccionadas en el plano como accesos.
	DesasociarAccesos	Desasocia un conjunto de puertas y ventanas seleccionadas en el plano como accesos (en caso de haber sido asociados previamente).
	VerAccesos	Selecciona en el plano los accesos previamente asociados.
	VaciarAccesos	Desasocia todos los accesos previamente asociados.
	VerParedDeAcceso	Selecciona en el plano la pared

		en la que se encuentra una puerta o ventana seleccionada en el plano.
Plugins.actions.tags	AsociarTags	Asocia y almacena un conjunto de muebles seleccionados en el plano como tags.
	DesasociarTags	Desasocia un conjunto de muebles seleccionados en el plano como tags (en caso de haber sido asociados previamente).
	VerTags	Selecciona en el plano los tags previamente asociados.
	VaciarTags	Desasocia todos los tags previamente asociados.
	VerEspacioDeTag	Selecciona en el plano el espacio en el que se encuentra un tag seleccionado en el plano.
Plugins.actions	XmlExport	Transforma las asociaciones previas al nuevo modelo de clases y lo serializa en un XML.

Tabla 4-2: paquetes y sus clases.

A continuación detallamos cada acción y sus respectivas clases. En cada caso, se presenta el diagrama de secuencia y se detalla el comportamiento correspondiente.

4.1. ACCIONES RELACIONADAS A LA IDENTIFICACION DEL TERRENO

- Acción **AsociarTerreno**
- Acción **VerTerreno**
- Acción **DesasociarTerreno**

Acción AsociarTerreno

Esta acción recupera un objeto seleccionado en el plano y lo identifica, en caso de cumplir con los requisitos detallados a continuación, como terreno.

Los requisitos son:

1- Seleccionar en el plano un y solo un objeto.
2- El objeto seleccionado debe ser instancia de la clase **Room.**
3- No debe existir al momento de ejecutar la acción un terreno identificado.
4- Las líneas que forman los puntos de la habitación no deben cruzarse entre sí.

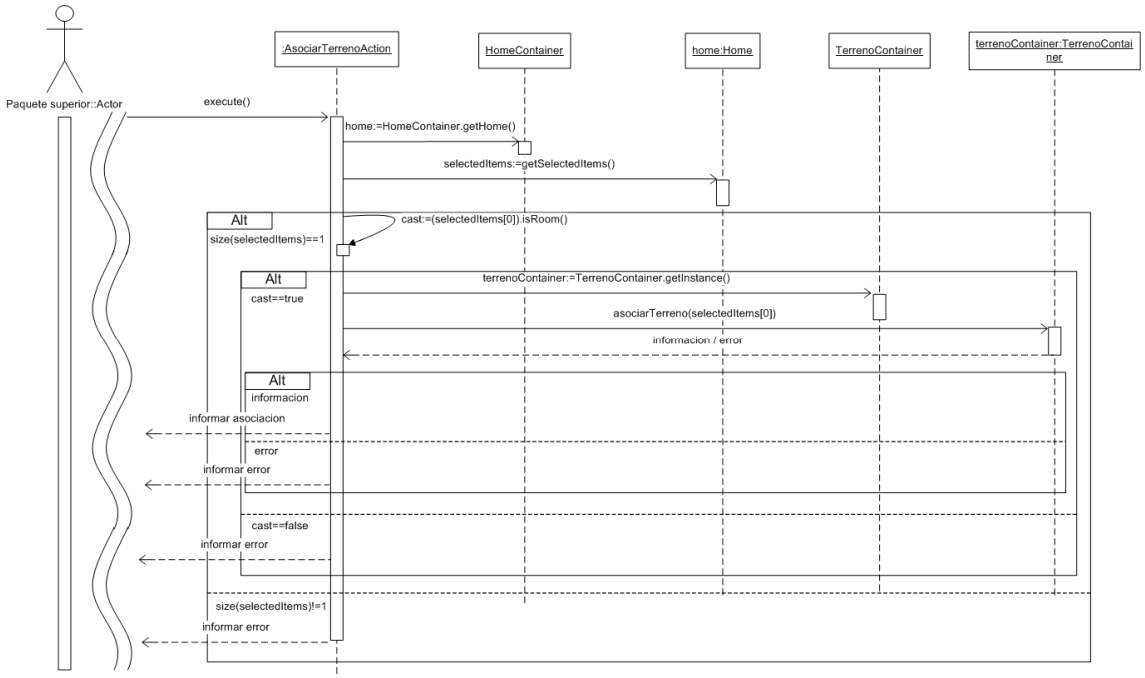


Figura 4-2: diagrama de secuencia del método execute() de la clase AsociarTerreno

Como se muestra en la Figura 4-2, se comienza corroborando el requisito 1, invocando al método *getSelectedItems()* de la instancia de la clase **Home,** el cual retorna la colección de elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es igual a 1, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

Para corroborar el requisito 2, se verifica que el objeto seleccionado en el plano sea una instancia de la clase **Room** y se lo pasa como parámetro en la invocación al método *asociarTerreno(Room habitacion)* (la explotación y explicación de este método se puede observar en el Anexo A) sobre la instancia de la clase **TerrenoContainer**. En caso de no serlo, el requisito 2 no se cumple, se captura la excepción y se informa en pantalla el error. En caso contrario, se procede con la ejecución del método *asociarTerreno(Room habitación)*. Se verifica el requisito 3, controlando que no exista ningún terreno definido previamente y se el requisito 4, verificando que los lados de la habitación no se crucen. Si alguno de estos no se cumplen, se informa del error en pantalla. En caso contrario se instancia la clase **TerrenoE** a partir de la habitación seleccionada y se guarda en la instancia de la clase **TerrenoContainer.**

Acción VerTerreno

Esta acción selecciona en el plano la instancia de la clase **Room** que fue identificada como terreno previamente.

Los requisitos de esta acción son:

1- Haber identificado un terreno previamente.

Como se muestra en la Figura 4-3, se debe invocar al método *getTerreno()* de la instancia de la clase **TerrenoContainer**. Dicho método disparará la excepción **TerrenoNoDefinidoException** en caso de no existir un terreno definido. En caso contrario, devolverá la instancia de la clase **TerrenoE** que identifica al terreno.

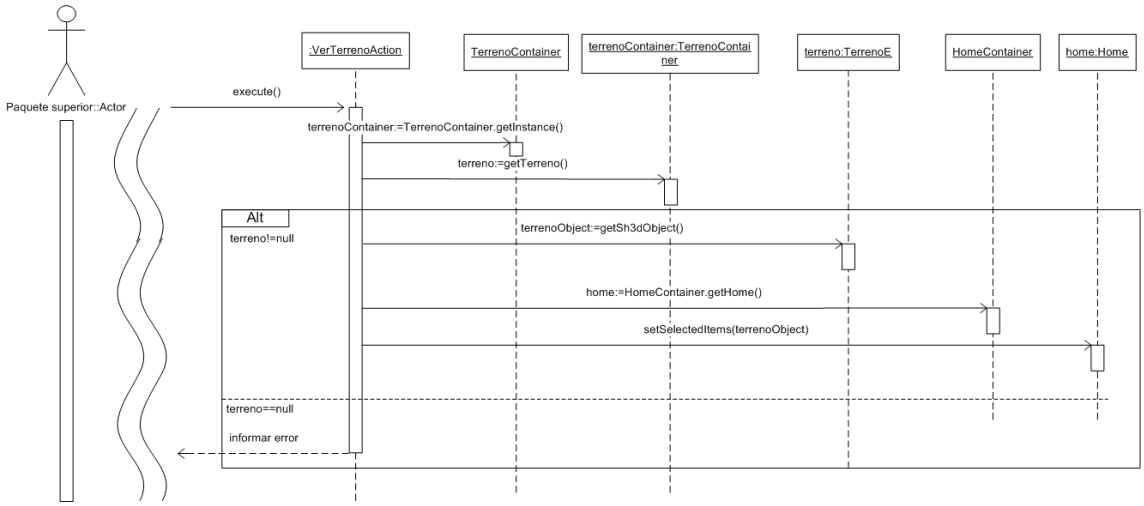


Figura 4-3: diagrama de secuencia del método execute() de la clase VerTerreno

En caso de que no haya terreno definido, se captura la excepción y se informa en pantalla el error. En caso contrario, se recupera la instancia de la clase **Room** (a partir del método *getSh3dObject()* de la clase **TerrenoE**) que identifica al terreno, y se selecciona en el plano el objeto a través del método *setSelectedItems(Collection<Room> rooms)* de la instancia de la clase **Home.**

<u>Acción DesasociarTerreno</u>

Esta acción deja sin efecto una identificación de terreno previa.

Los requisitos de esta acción son:

1- Haber identificado un terreno previamente.

Como se muestra en la Figura 4-4, se comienza invocando al método *limpiarTerreno()* de la instancia de la clase **TerrenoContainer**. En caso de no existir un terreno identificado, el método dispara la excepción **TerrenoNoDefinidoException**, la cual será capturada y se informará el error en pantalla. En caso contrario, la instancia de la clase **TerrenoContainer** invocará a su método *desasociarTerreno()*, el cual se encargará de vaciar su atributo *terreno* Luego, se invoca al método *vaciar()* de la instancia de la clase **EspaciosContainer,** el cual procederá a borrar todos los espacios previamente identificados. Por último, se informa en pantalla que la función se realizó con éxito.

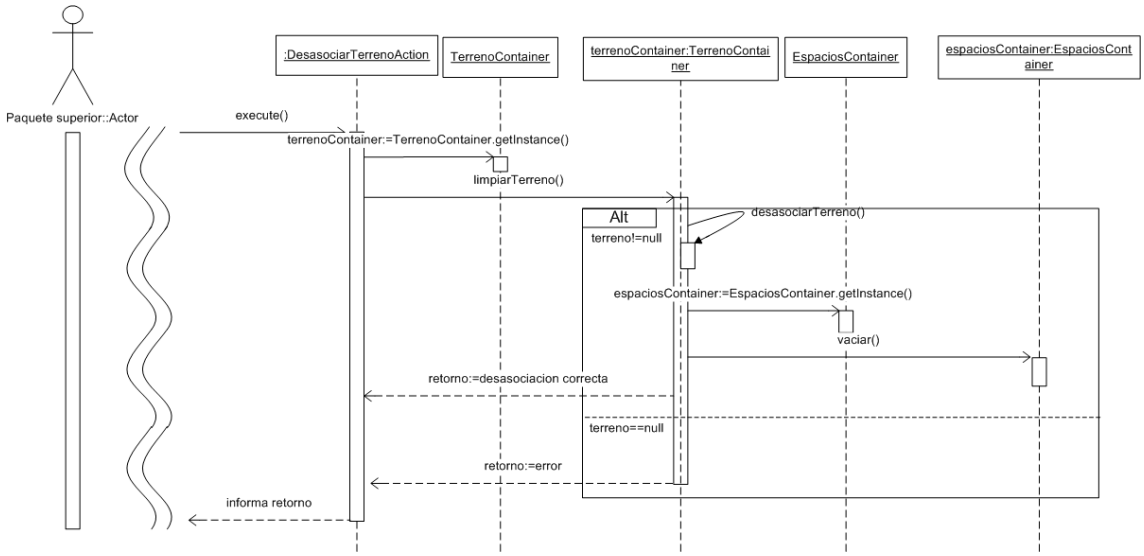


Figura 4-4: diagrama de secuencia del método execute() de la clase DesasociarTerreno

4.2. ACCIONES RELACIONADAS A LA IDENTIFICACIÓN DE ESPACIOS

- Acción **GenerarEspacio**
- Acción **VerEspacioPadre**
- Acción **EliminarEspacio**

Acción GenerarEspacio

Esta acción recupera los objetos seleccionados en el plano y los identifica, en caso de cumplir con los requisitos detallados a continuación, como un espacio.

Los requisitos son:

1- Seleccionar en el plano un conjunto de al menos tres objetos.
2- Todos los objetos seleccionados deben ser paredes (instancias de la clase **Wall**).
3- Las paredes deben formar, a partir de sus coordenadas de inicio y de fin, un polígono cerrado.
4- El conjunto de paredes no pueden haber sido identificadas como espacio previamente
5- Debe existir un terreno identificado.
6- El área del espacio debe estar totalmente incluido dentro del área que abarca el terreno.
7- El área del espacio no debe superponerse con el área de ningún otro espacio, excepto cuando la superposición significa una inclusión total, caso en cual los espacios no podrán compartir pared alguna.

Como se muestra en la Figura 4-5 se comienza corroborando el requisito 1, invocando al método *getSelectedItems()* de la instancia de la clase **Home**, el cual retorna la colección de

elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es mayor a 2, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

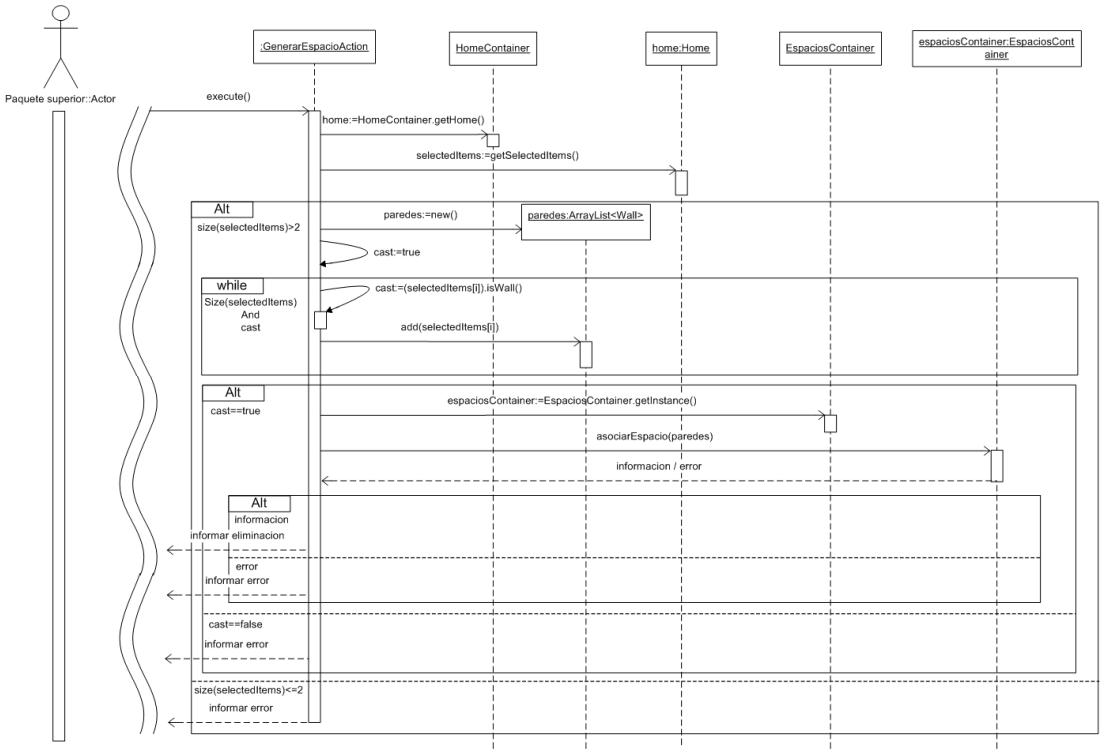


Figura 4-5: diagrama de secuencia del método execute() de la clase GenerarEspacio

Para corroborar el requisito 2, se verifican que los objetos seleccionados sean todos instancias de la clase **Wall.** En caso de cumplir el requisito, se pasa el conjunto de paredes seleccionadas como parámetro en la invocación al método *asociarEspacio(ArrayList<Wall> paredes)* de la instancia de la clase **EspaciosContainer**. En caso contrario, el requisito 2 no se cumple por lo que se informa el error en pantalla el error. La explotación y explicación del método *asociarEspacio(ArrayList<Wall> paredes)* se puede ver en el Anexo A. El mismo se encargará de corroborar los siguientes requisitos.

En caso de que el conjunto de paredes cumplan con los requisitos para ser asociadas como un espacio, se instancia la clase **EspacioE** se a partir de las y se guarda en la instancia de la clase **EspaciosContainer.** En caso contrario se informa el error en pantalla.

Acción VerEspacioPadre

Esta acción recupera los objetos seleccionados en el plano, los cuales deben haber sido identificados como un espacio previamente, e identifica el espacio en el que está incluido, siempre y cuando haya un espacio identificado previamente con estas características.

Los requisitos de esta acción son:

1- Seleccionar en el plano un conjunto de al menos tres objetos.
2- Todos los objetos seleccionados deben ser paredes (instancias de la clase **Wall**)**.**
3- Las paredes seleccionadas deben haber sido identificadas como un espacio previamente
4- Debe existir un espacio identificado previamente que incluya al espacio seleccionado.

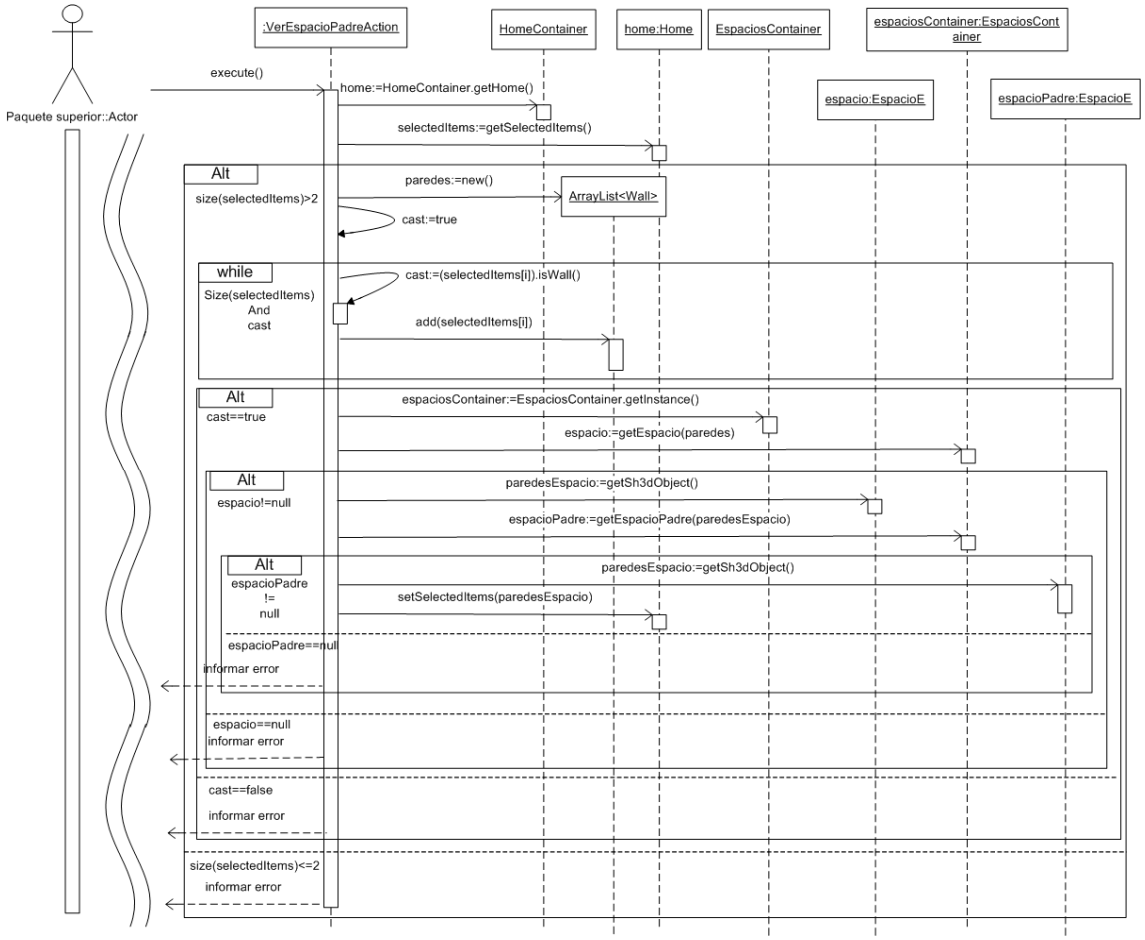


Figura 4-6: diagrama de secuencia del método execute() de la clase VerEspacioPadre

Como se muestra en la Figura 4-6*,* se comienza corroborando el requisito 1, invocando al método *getSelectedItems()* de la instancia de la clase **Home**, el cual retorna la colección de elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es mayor a 2, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

Para corroborar el requisito 2, se verifican que los objetos seleccionados sean todos instancias de la clase **Wall**. En caso de no serlo, el requisito 2 no se cumple y se informa en pantalla el error. En caso contrario, se pasa la colección de elementos seleccionados como parámetro en la invocación al método *getEspacio(ArrayList<Wall> paredes)* de la instancia de la clase **EspaciosContainer**, el cual se encarga de recuperar la instancia de la clase **EspacioE** almacenada que representa al espacio identificado previamente formado únicamente por

todas las paredes seleccionadas. Este método no retornará ningún elemento en caso de no existir dicho espacio, caso en el cual el requisito 3 no se cumple y se notifica el error en pantalla. En caso de retornar una instancia de la clase **EspacioE**, se procede a corroborar que exista un espacio previamente identificado que contenga al objeto.

Para que un espacio contenga a otro, el área que forma el "espacio hijo" deberá estar completamente incluida en el área que forma el espacio padre. Para esto, se invoca al método *getEspacioPadre(Espacio espacio)* de la instancia de la clase **EspaciosContainer**. Dicho método utiliza los puntos vértices del polígono cerrado que forma el espacio para obtener la forma del espacio seleccionado, y busca entre todos los espacios identificados el espacio cuya área incluya totalmente al área obtenida. En caso de existir, retorna la instancia de la clase **EspacioE** que cumpla con el objetivo, a partir de la cual se obtienen todas las paredes del mismo. Las mismas se seleccionan en el plano y se informa en pantalla que las paredes seleccionadas son el "espacio padre". En caso de no retornar elemento alguno implica que no existe un espacio padre identificado para las paredes seleccionadas, por lo que el error se informa en pantalla.

Acción EliminarEspacio

Esta acción deja sin efecto la identificación de un espacio

Los requisitos de esta función son:

1- Seleccionar en el plano un conjunto de al menos tres objetos.
2- Todos los objetos seleccionados deben ser paredes (instancias de la clase **Wall**).
3- Las paredes deben haber sido identificadas previamente como un espacio

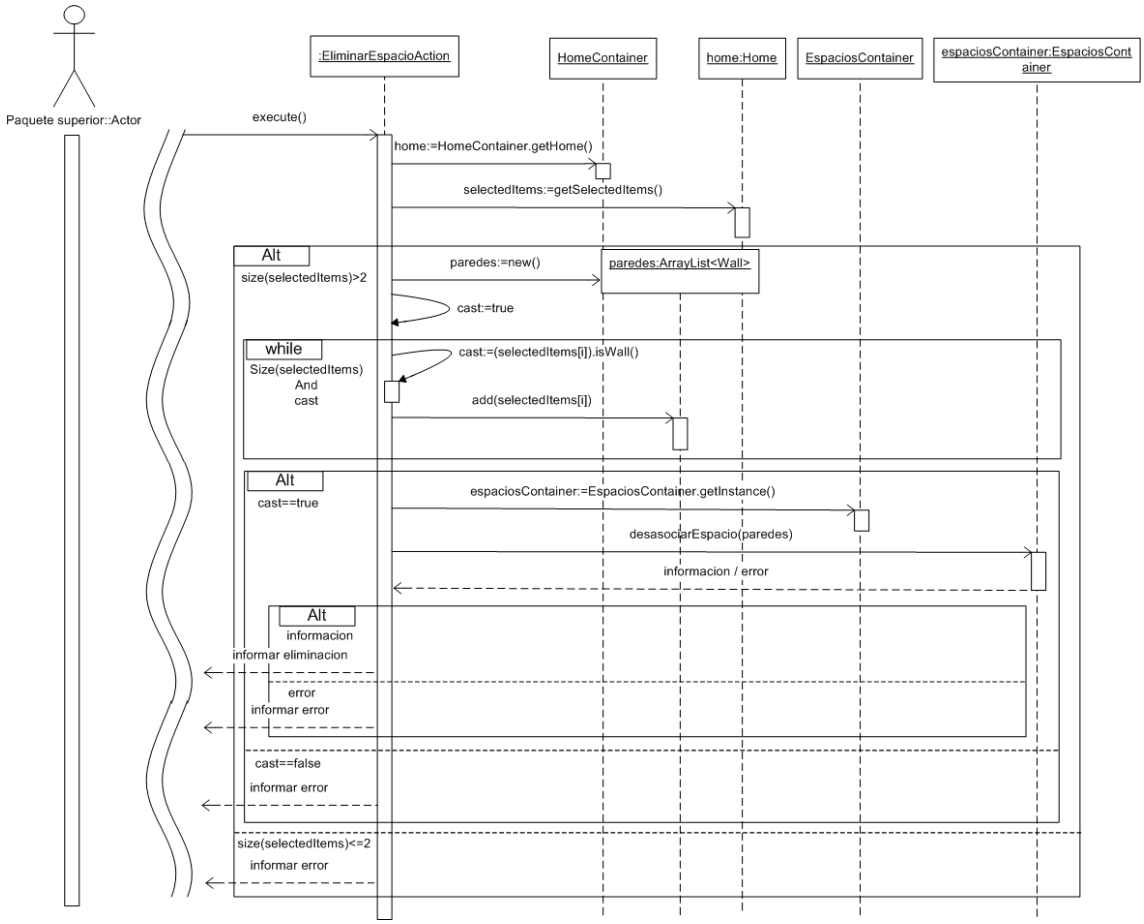


Figura 4-7: diagrama de secuencia del método execute() de la clase EliminarEspacio

Como se muestra en la Figura 4-7, se comienza corroborando el requisito 1, invocando al método *getSelectedItems()* de la instancia de la clase **Home,** el cual retorna la colección de elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es mayor a 2, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

Para corroborar el requisito 2, se verifican que los objetos seleccionados sean todos instancias de la clase **Wall**. En caso de no serlo, el requisito 2 no se cumple y se informa en pantalla el error. En caso contrario, se pasa la colección de elementos seleccionados como parámetro en la invocación al método des*asociarEspacio(ArrayList<Wall> paredes)* (Figura 4-21a) de la instancia de la clase **EspaciosContainer.** La explotación y explicación del método des*asociarEspacio(ArrayList<Wall> paredes)* se puede ver en el Anexo A. El mismo se encargará de corroborar los siguientes requisitos. En caso de que el conjunto de paredes hayan sido identificadas previamente como un espacio, se elimina la instancia de la clase **EspacioE** formada únicamente por el conjunto de paredes seleccionadas de la lista de espacios identificados. En caso contrario se informa el error en pantalla.

4.3. ACCIONES RELACIONADAS A LA IDENTIFICACION DE ACCESOS

- Acción **AsociarAccesos**
- Acción **VerAccesos**
- Acción **VerParedDeAcceso**
- Acción **DesasociarAccesos**
- Acción **VaciarAccesos**

Acción AsociarAccesos

Esta acción recupera los objetos seleccionados en el plano y los identifica, en caso de cumplir con los requisitos detallados a continuación, como accesos.

Los requisitos son:

1- Seleccionar en el plano un conjunto de objetos.
2- Los objetos seleccionados deben ser instancia de la clase **HomeDoorOrWindow.**
3- Los objetos no deben haber sido identificados como accesos previamente.
4- Los objetos seleccionados deben estar posados sobre una pared perteneciente a un espacio identificado.

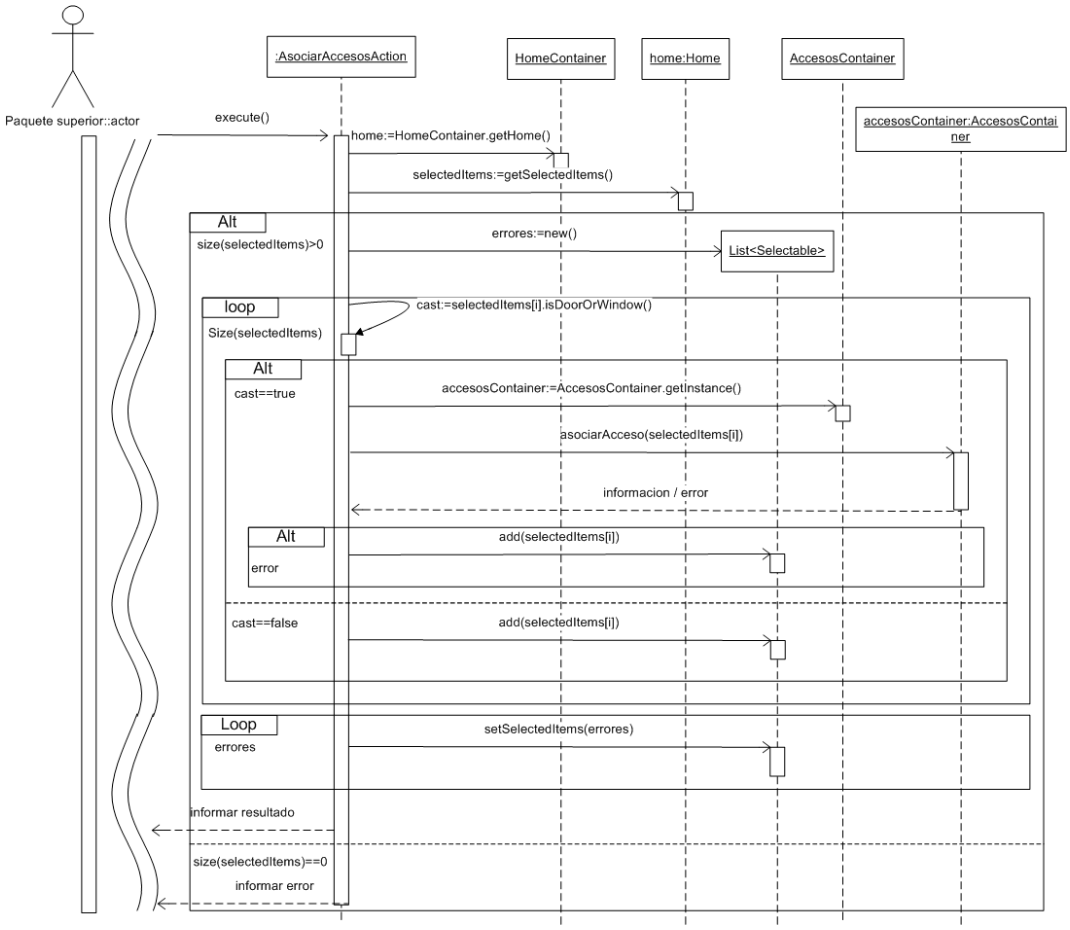


Figura 4-8: diagrama de secuencia del método execute() de la clase AsociarAccesos

Cabe destacar que en caso de que no todos los elementos seleccionados en el plano cumplan los requisitos, la acción se ejecutará parcialmente, es decir, que solo serán identificados como accesos los objetos que cumplan los requisitos 2, 3 y 4. Los elementos que no cumplan alguno de los 4 requisitos quedarán seleccionados en el plano y será notificado en pantalla el error particular de cada uno de ellos.

Como se muestra en la figura 4-8, se comienza corroborando el requisito 1. Se invoca al método *getSelectedItems()* de la instancia de la clase **Home,** el cual retorna la colección de elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es mayor o igual a 1, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

Debido a que la función solo identifica como accesos los objetos que cumplan los requisitos 2, 3 y 4, y los que no deben ser notificados en pantalla, se procede a iterar sobre la lista de elementos seleccionados invocando en cada una de las iteraciones al método *asociarAcceso(Object pieza)* de la instancia de la clase **AccesosContainer**, pasándole como parámetro el objeto actual de la iteración (la explotación y explicación de este método se puede ver en el Anexo A). Por cada uno de estos elementos, el método determina si puede ser identificado como acceso o no, y en caso positivo, se instancia la clase **AccesoE** a partir de la instancia de la clase **HomeDoorOrWindow** y se guarda en la instancia de la clase **AccesosContainer.**

Al finalizar el método, se informarán cuales fueron los objetos que cumplieron los requisitos, y por lo tanto identificados como accesos, y cuáles no.

Acción VerAccesos

Esta acción selecciona en el plano los elementos que fueron identificados como accesos.

Los requisitos de esta acción son:

1- Haber identificado al menos un acceso previamente.

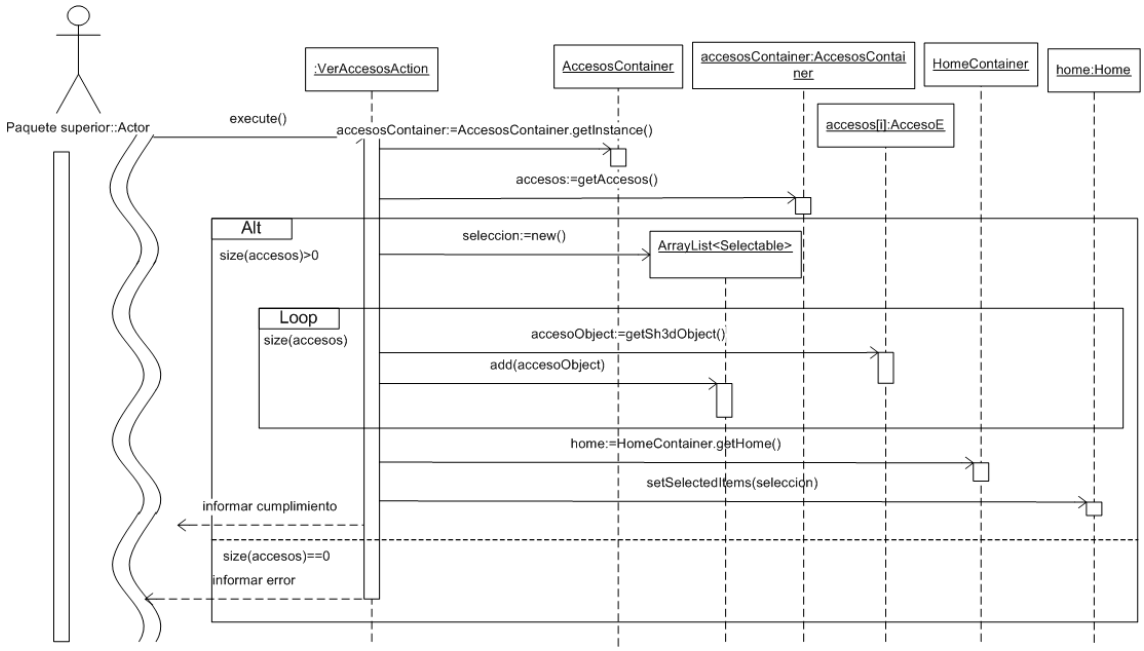


Figura 4-9: diagrama de secuencia del método execute() de la clase VerAccesos

Como se muestra en la Figura 4-9, para llevar a cabo esta función solo se debe invocar al método *getAccesos()* de la instancia de la clase **AccesosContainer**. Dicho método dispara la excepción **SinElementosException** en caso de no existir ningún acceso identificado y se notifica el error en pantalla. En caso contrario, devuelve el conjunto de instancias de la clase **AccesoE** que representan los accesos identificados hasta el momento. Por cada elemento en este conjunto se recupera la instancia de la clase **HomeDoorOrWindow** (a partir del método *getSh3dObject()* de la clase **AccesoE**) y se lo guarda en la variable local *accesos* del método.

Una vez finalizada la iteración se seleccionan en el plano los objetos a través del método *setSelectedItems(Collection<HomeDoorOrWindow> piece)* de la instancia de la clase **Home**.

Acción VerParedDeAcceso

Esta acción selecciona en el plano la pared a la que pertenece un acceso.

Los requisitos de esta acción son:

1- Seleccionar en el plano un y solo un objeto.
2- El objeto seleccionado debe haber sido identificado como acceso.

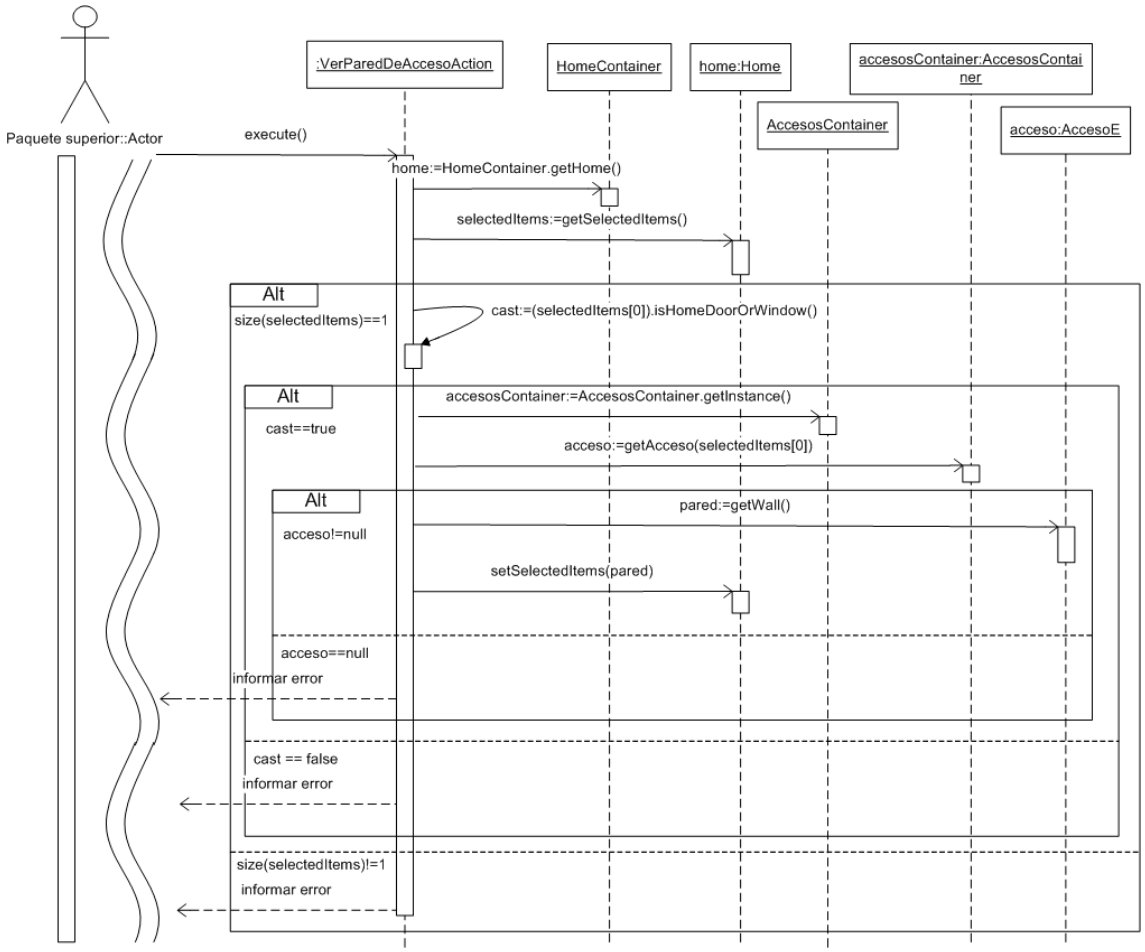


Figura 4-10: diagrama de secuencia del método execute() de la clase VerParedDeAcceso

Como lo muestra la Figura 4-10, la función comienza corroborando el requisito 1. Se invoca al método *getSelectedItems()* de la instancia de la clase **Home,** el cual retorna la colección de elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es mayor o igual a 1, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

A continuación se recupera el objeto seleccionado del plano y se verifica que sea instancia de la clase **HomeDoorOrWindow.** Si esto falla se notifica en pantalla que el objeto no puede ser un acceso. En caso contrario se invoca al método *getAcceso(HomeDoorOrWindow acceso)* de la instancia de la clase **AccesosContainer**.

Este método retornará la instancia de la clase **AccesoE** que representa al acceso identificado previamente. Para lograr esto recorre la lista de accesos guardada en su atributo *accesos* y verifica que alguno de ellos contenga al objeto seleccionado en el plano.

En caso de que el resultado del método *getAcceso(HomeDoorOrWindow acceso)* de la instancia de la clase **AccesosContainer** no retorne ningún elemento, se informa en pantalla de que no existe un acceso previamente identificado para el objeto seleccionado en el plano. En caso contrario, se recupera la instancia de la clase **Wall** del acceso (a partir del método

getWall() de la clase **AccesoE**) y se selecciona en el plano la pared a través del método *setSelectedItems(Collection<Selectable> piece)* de la instancia de la clase **Home**.

Acción DesasociarAccesos

Esta acción deja sin efecto las identificaciones como accesos de los elementos seleccionados en el plano.

Los requisitos de esta acción son:

1- Seleccionar en el plano un conjunto de objetos.
2- Los objetos deben haber sido identificados como accesos previamente.

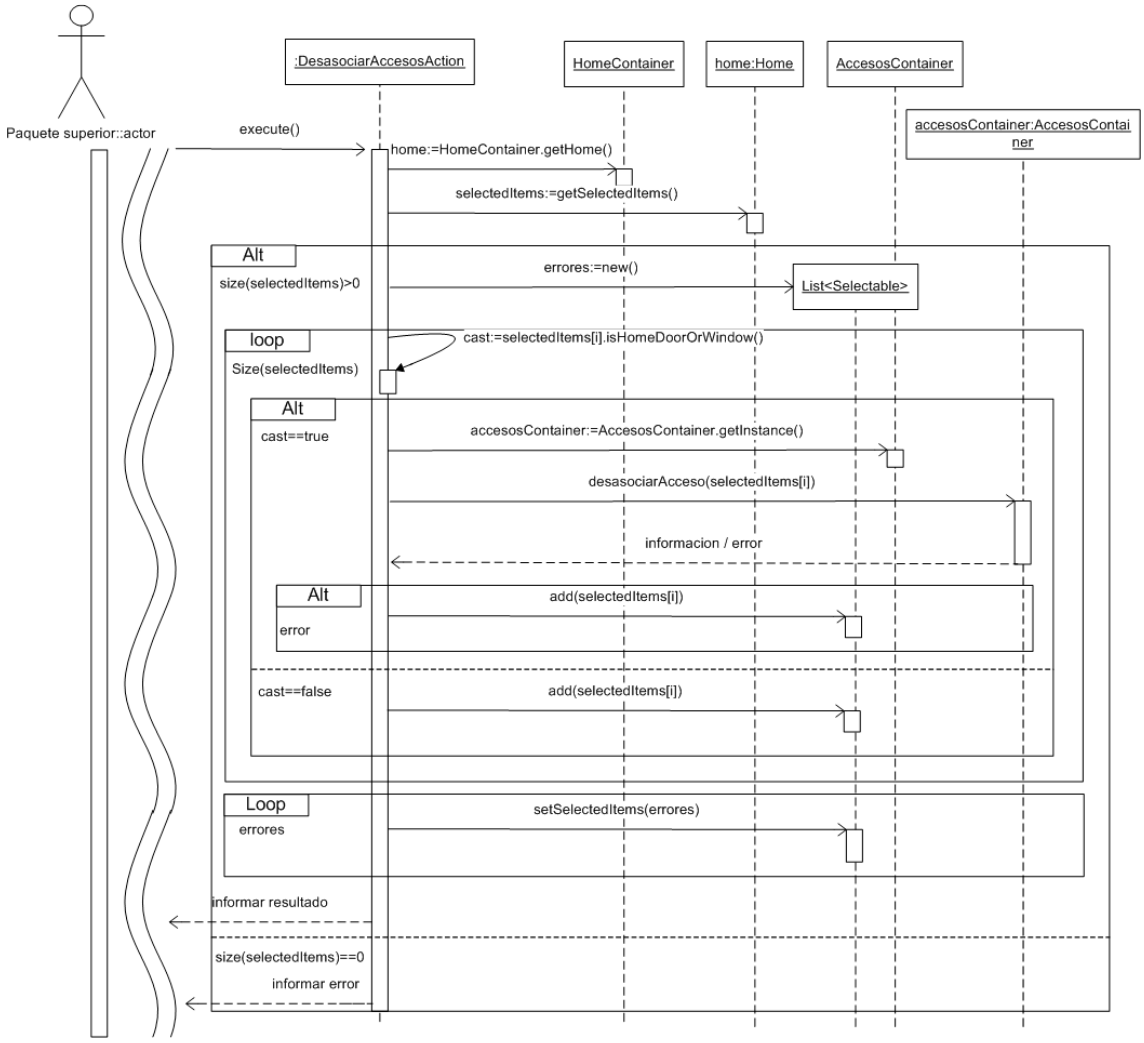


Figura 4-11: diagrama de secuencia del método execute() de la clase DesasociarAccesos

Cabe destacar que en caso de que no todos los elementos seleccionados hayan sido identificados como accesos, la acción se ejecutará parcialmente, es decir, que solo serán eliminados de la lista de accesos los objetos que cumplan el requisito 2. Los objetos que no

cumplan dicho requisito quedarán seleccionados en el plano y será notificado en pantalla el error particular de cada uno de ellos.

Como se muestra en la Figura 4-11, se comienza corroborando el requisito 1. Se invoca al método *getSelectedItems()* de la instancia de la clase **Home**, el cual retorna la colección de elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es mayor o igual a 1, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

Debido a que la función solo elimina como accesos los objetos que cumplan el requisito 2 y los que no deben ser notificados en pantalla, se procede a iterar sobre la lista de elementos seleccionados invocando en cada una de las iteraciones al método des*asociarAcceso(Object pieza)* de la instancia de la clase **AccesosContainer**, pasándole como parámetro el objeto actual de la iteración (la explotación y explicación de este método se puede observar en el Anexo A).

Por cada elemento seleccionado se notificará si la acción pudo ser ejecutada o no, y se notificará la causa en caso de ser negativa.

Acción VaciarAccesos

Esta acción deja sin efecto todas las identificaciones de accesos hechas hasta el momento.

Los requisitos de esta acción son:

1- Debe existir al menos un objeto identificado como acceso.

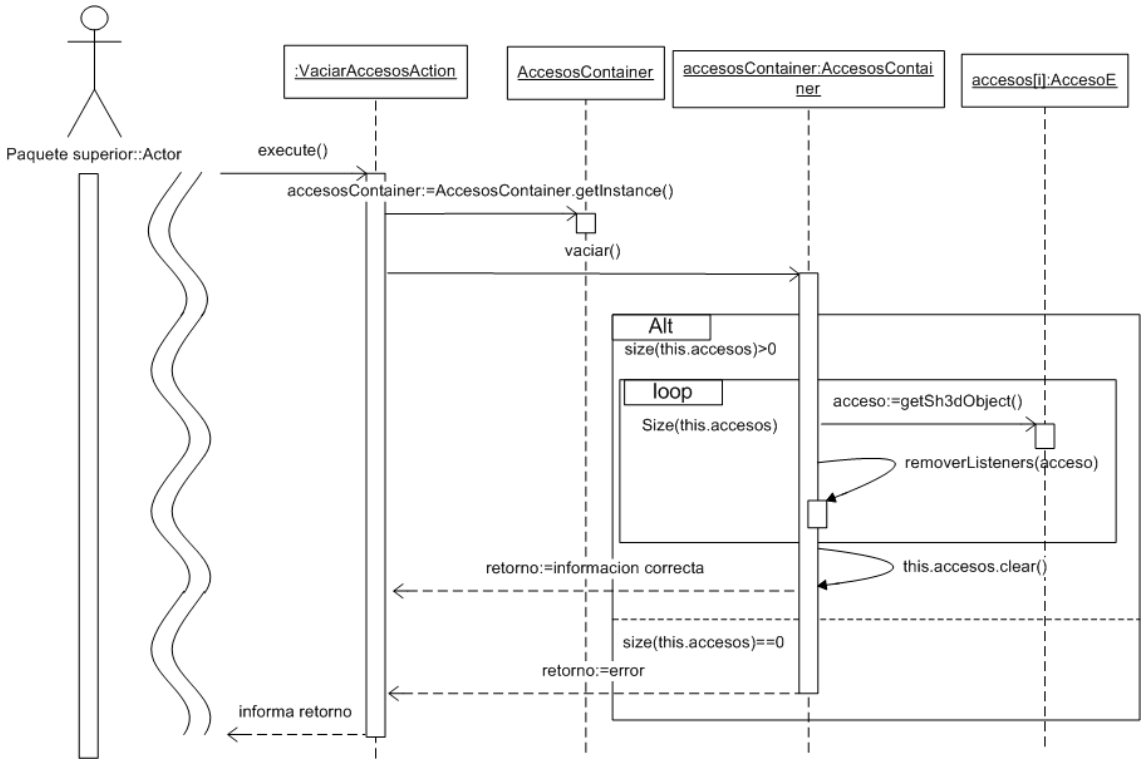


Figura 4-12: diagrama de secuencia del método execute() de la clase VaciarAccesos

Como lo muestra la Figura 4-12, el método *execute()* de la clase **VaciarAccesosAction** invoca al método *vaciar()* de la instancia de la clase **AccesosContainer.** En caso de que dicha instancia posea elementos en su atributo *accesos.* Por último, al retornar el control al método *execute()* de la clase **VaciarAccesosAction** se informa que la acción fue realizada con éxito.

En caso de que el atributo *accesos* de la instancia de la clase **AccesosContainer** no posea ningún elemento se notifica el error en pantalla.

4.4. ACCIONES RELACIONADAS A LA IDENTIFICACION DE TAGS

- Acción **AsociarTags**
- Acción **VerTags**
- Acción **VerEspacioDeTag**
- Acción **DesasociarTags**
- Acción **VaciarTags**

Acción AsociarTags

Esta acción recupera los objetos seleccionados en el plano y los identifica, en caso de cumplir con los requisitos detallados a continuación, como tags.

Los requisitos son:

1- Seleccionar en el plano un conjunto de objetos.
2- Los objetos seleccionados deben ser instancia de la clase **HomePieceOfFurniture.**
3- Los objetos seleccionados no pueden ser instancia de la clase **HomeDoorOrWindow.**
4- Los objetos no deben haber sido identificados como tags previamente.
5- Los objetos seleccionados deben estar incluidos en el área de un espacio previamente identificado.

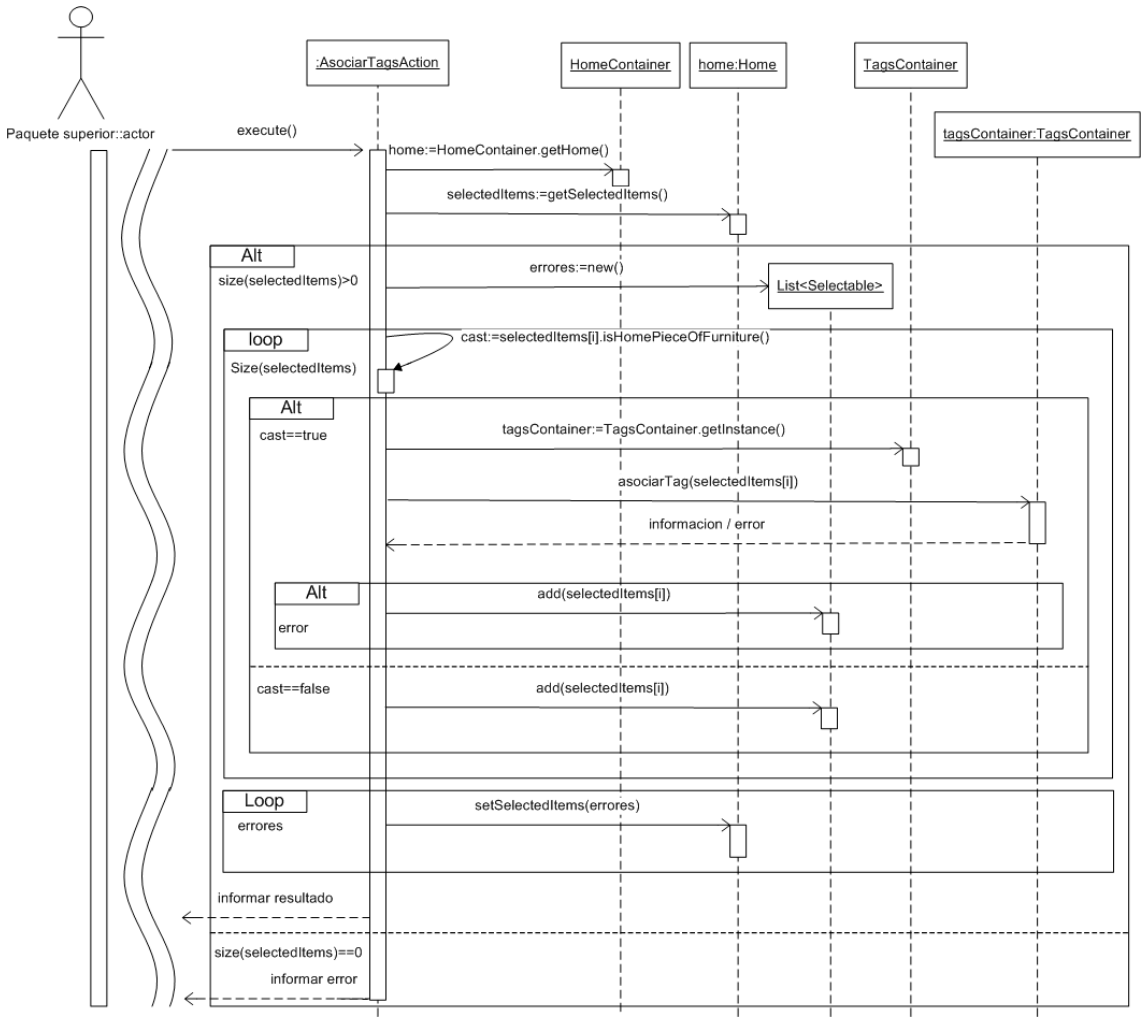


Figura 4-13: diagrama de secuencia del método execute() de la clase AsociarTags

Cabe destacar que en caso de que no todos los elementos seleccionados cumplan los requisitos, la acción se ejecuta parcialmente, es decir, que solo son identificados como accesos los objetos que cumplan los requisitos 2, 3, 4 y 5. Los elementos que no cumplan alguno de estos requisitos quedarán seleccionados en el plano y será notificado en pantalla el error particular de cada uno de ellos.

Como lo muestra la Figura 4-13, se comienza corroborando el requisito 1. Se invoca al método *getSelectedItems()* de la instancia de la clase **Home**, el cual retorna la colección de elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es mayor o

igual a 1, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

Debido a que la función solo identifica como accesos los objetos que cumplan los requisitos 2, 3, 4 y 5, y los que no deben ser notificados en pantalla, se procede a iterar sobre la lista de elementos seleccionados invocando en cada una de las iteraciones al método *asociarTag(Object pieza)* (la explotación y explicación de este método se puede ver en el Anexo A) de la instancia de la clase **TagsContainer**, pasándole como parámetro el objeto actual de la iteración.

Este método determina si el elemento puede ser o no identificado como tag, verificando los requisitos restantes. En caso afirmativo se instancia la clase **TagE** a partir de la instancia de la clase **HomePieceOfFurniture** y se guarda en la instancia de la clase **TagsContainer**. En caso contrario se informa la causa del resultado.

Acción VerTags

Esta acción selecciona en el plano los elementos, que hasta el momento, fueron identificados como tags.

Los requisitos de esta acción son:

1- Haber identificado al menos un tag previamente.

Como lo muestra la Figura 4-14, para llevar a cabo esta acción solo se debe invocar al método *getTags()* de la instancia de la clase **TagsContainer**. Dicho método dispara la excepción **SinElementosException** en caso de no existir ningún tag identificado previamente y se informará el error en pantalla. En caso contrario, devolverá el conjunto de instancias de la clase **TagE** que representan los tags identificados hasta el momento.

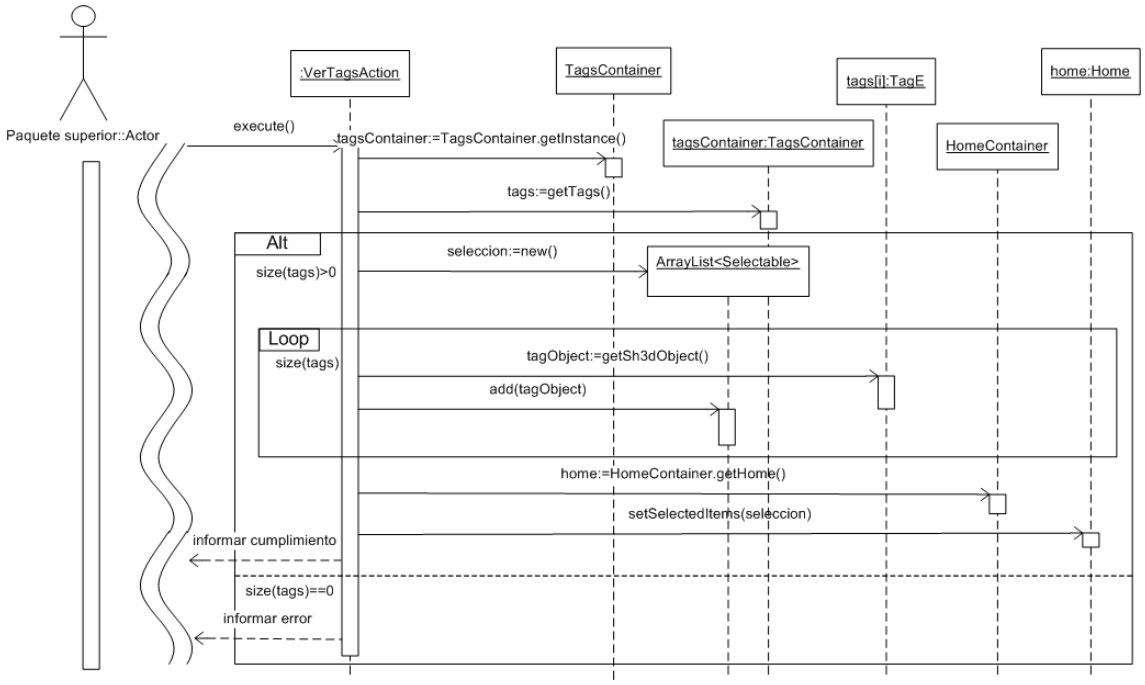


Figura 4-14: diagrama de secuencia del método execute() de la clase VerTags

Por cada tag en el conjunto retornado, se recupera la instancia de la clase **HomePieceOfFurniture** (a partir del método *getSh3dObject()* de la clase **TagE**) y se lo guarda en la variable local *tags* del método. Una vez finalizada la iteración se seleccionan en el plano los objetos a través del método *setSelectedItems(Collection<Selectable> piece)* de la instancia de la clase **Home.**

Acción VerEspacioDeTag

Esta acción selecciona en el plano las paredes que forman el espacio en el que el tag está incluido

Los requisitos de esta acción son:

1- Seleccionar en el plano un y solo un objeto.
2- El objeto seleccionado debe haber sido identificado como tag.

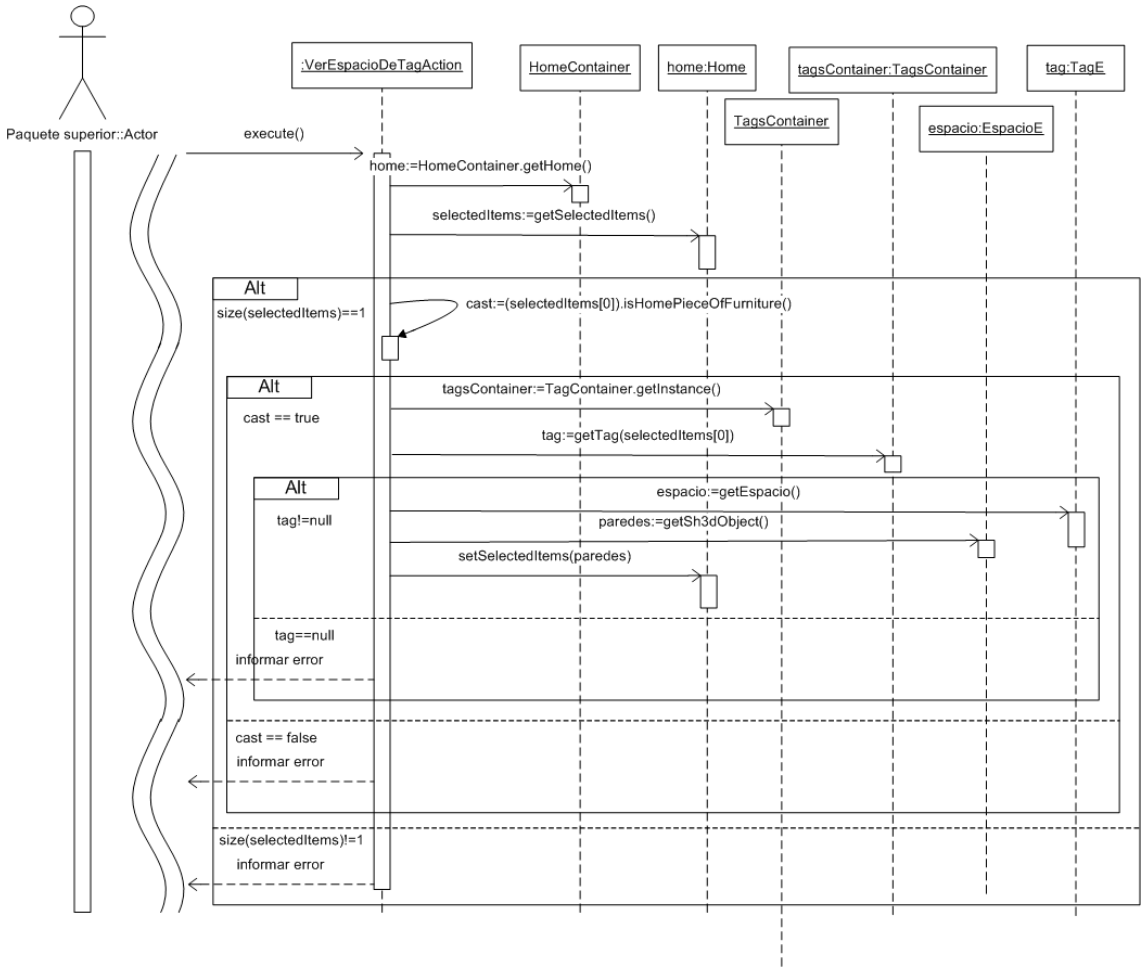


Figura 4-15: diagrama de secuencia del método execute() de la clase VerEspacioDeTag

Como lo muestra la Figura 4-15, la acción comienza corroborando el requisito 1. Se invoca al método *getSelectedItems()* de la instancia de la clase **Home,** el cual retorna la colección de elementos seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es igual a 1, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

A continuación se recupera el objeto seleccionado del plano y se verifica que sea instancia de la clase **HomePieceOfFurniture.** Si esto falla se notifica en pantalla que el objeto no puede ser un tag. En caso contrario se invoca al método *getTag(HomePieceOfFurniture tag)* de la instancia de la clase **TagsContainer**. Este método retorna la instancia de la clase **TagE** que contiene al elemento. Para lograr esto recorre la lista de tags guardada en su atributo *tags* y verifica que alguno de ellos contenga al objeto seleccionado en el plano guardado en su atributo sh3dObject. En caso de no existir se informa que el elemento no es un tag identificado.

Una vez que verificado el requisito 2, se recuperan las paredes que identifican al espacio que incluye al tag, y se las selecciona en el plano a través del método *setSelectedItems(Collection<Selectable> piece)* de la instancia de la clase **Home.**

Acción DesasociarTags

Esta acción deja sin efecto las identificaciones como tags de los elementos seleccionados en el plano.

Los requisitos de esta función son:

1- Seleccionar en el plano un conjunto de objetos.
2- Los objetos deben haber sido identificados como tags previamente.

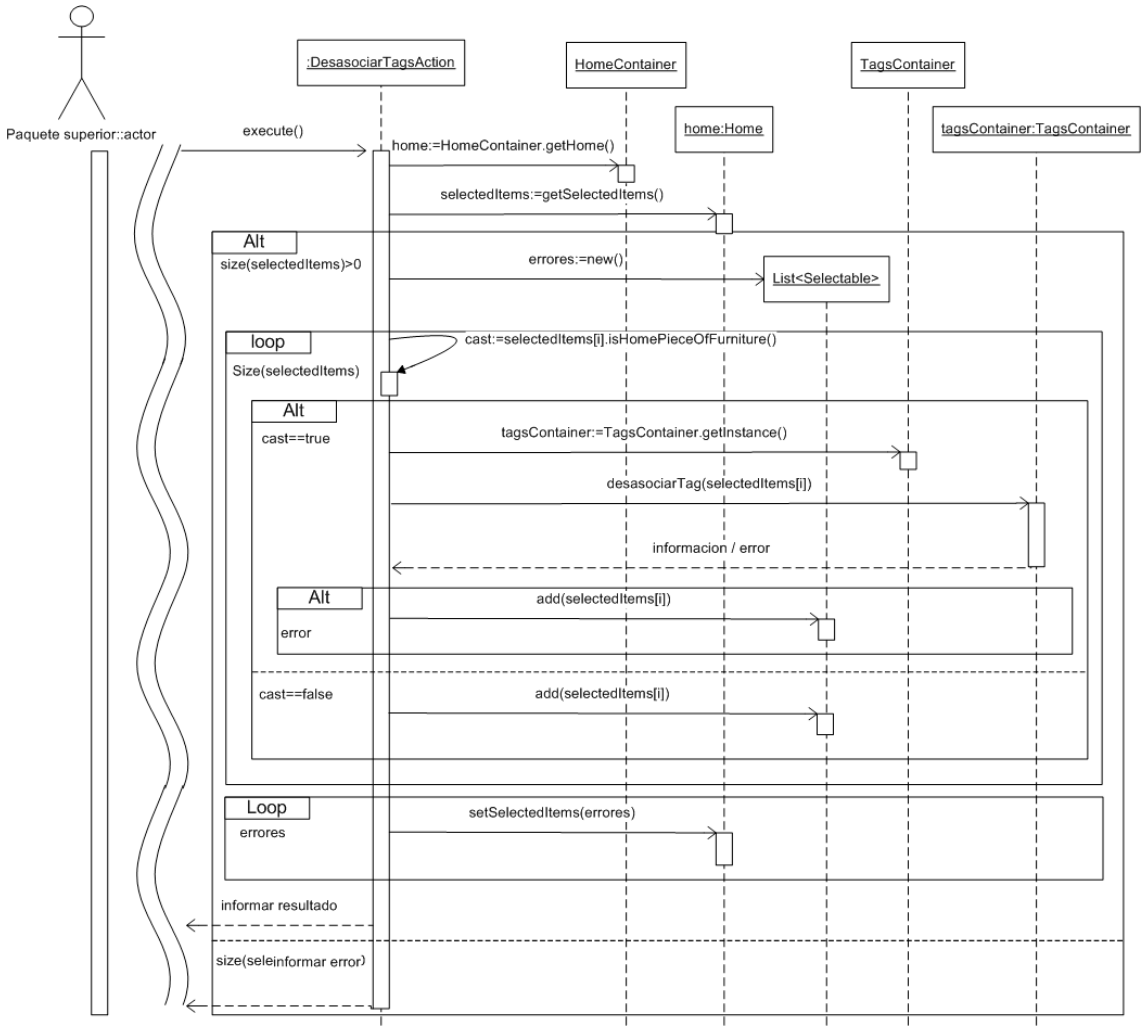


Figura 4-16: diagrama de secuencia del método execute() de la clase DesasociarTags

Cabe destacar que en caso de que no todos los elementos seleccionados hayan sido identificados como tags, la acción se ejecuta parcialmente, es decir, que solo serán eliminados de la lista de tags identificados los objetos que cumplan el requisito 2. Los objetos que no cumplan dicho requisito quedarán seleccionados en el plano y será notificado en pantalla el error particular de cada uno de ellos.

Como lo muestra la Figura 4-16, se comienza corroborando el requisito 1. Se invoca al método *getSelectedItems()* de la instancia de la clase **Home,** el cual retorna la colección de elementos

seleccionados en el plano**,** y al resultado se le aplica el método *size()*. Si el resultado es mayor o igual a 1, el requisito 1 se cumple y se procede a corroborar los demás requisitos. En caso contrario se informa el error en pantalla.

Debido a que la acción solo elimina los objetos que cumplan el requisito 2 y los que no deben ser notificados en pantalla, se procede a iterar sobre la lista de elementos seleccionados invocando en cada una de las iteraciones al método des*asociarTag(Object pieza)* de la instancia de la clase **TagsContainer** (la explotación y explicación de este método se puede observar en el Anexo A), pasándole como parámetro el objeto actual de la iteración.

Este método dejará sin efecto la identificación de cada uno de los tags seleccionados e identificará en el plano los elementos que no cumplan el requisito 2.

Acción VaciarTags

Esta acción deja sin efecto todas las identificaciones de tags hechas hasta el momento.

Los requisitos de esta acción son:

1- Debe existir al menos un objeto identificado como tag.

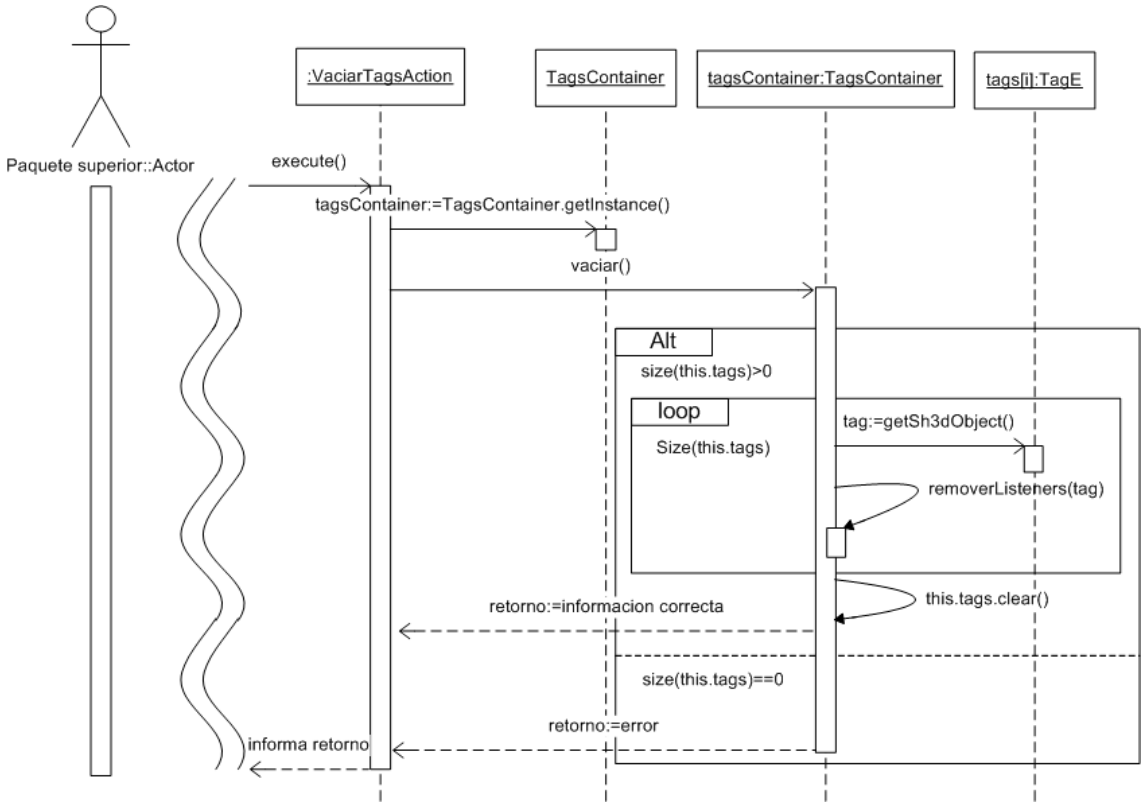


Figura 4-17: diagrama de secuencia del método execute() de la clase VaciarTags

Como lo muestra la Figura 4-17, el método *execute()* de la clase **VaciarTagsAction** invoca al método *vaciar()* de la instancia de la clase **TagsContainer.** En caso de que dicha instancia posea elementos en su atributo *tags,* eliminará cada uno de ellos de la misma. Por último, al retornar el control al método *execute()* de la clase **VaciarTagsAction** se informa que la acción fue realizada con éxito.

En caso de que el atributo *tags* de la instancia de la clase **TagsContainer** no posea ningún elemento, se dispara una excepción instancia de la clase **SinElementosException** y se notifica el error en pantalla.

4.5. ACCIONES RELACIONADAS A LA EXPORTACIÓN DE LAS IDENTIFICACIONES

- Acción **XMLExport**

Acción XMLExport

Esta acción se encarga de recuperar los objetos del modelo SH3D identificados en el modelo extendido, y a partir de ellos crear las instancias del modelo final, para concluir con la exportación del mismo en un archivo XML.

La secuencia de código, como se muestra en la Figura 4-18*,* es la siguiente:

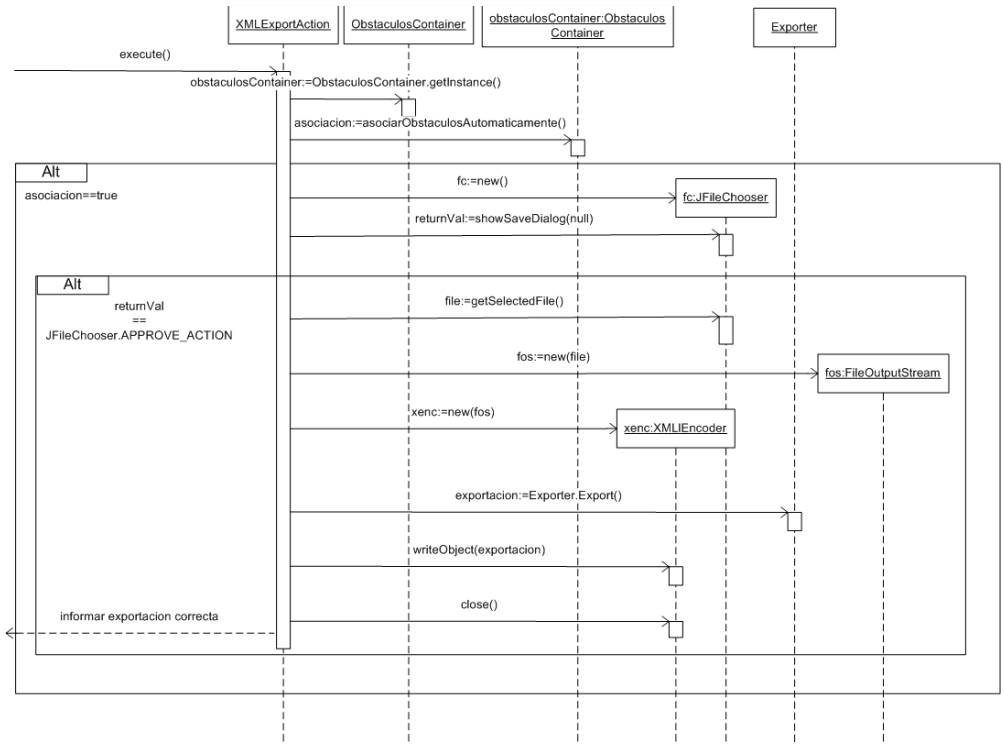


Figura 4-18: diagrama de secuencia del método execute() de la clase XMLExport

Primero se utiliza el método *asociarObstaculosAutomaticamente()* de la instancia de la clase **ObstaculosContainer** para identificar automáticamente los obstáculos (la explotación y explicación de este método se puede ver en el Anexo A). En caso de que este método retorne FALSE, implica que elementos del plano que no han sido identificados aun como tags, poseen errores lógico. El sistema informará el error en pantalla y proveerá al usuario decidir si desea seguir con la exportación excluyendo esos elementos o revisar el plano.

En caso de seguir la ejecución, el método provee la opción de definir la ubicación del archivo XML donde se exportará el modelo final. Para decidir cuál será el archivo de almacenaje se utiliza la clase **JFileChooser** de la API de Java, el cual guarda la ruta ingresada por el usuario. Una vez decidido esto se crea una instancia de la clase **FileOutputStream** configurándole la ruta de la instancia de la clase **JFileChooser.**

Luego se crea una instancia de la clase XMLEncoder con la instancia de la clase **FileOutputStream.** Una vez llegado a este punto, es necesario instanciar nuestro modelo final con los elementos del modelo SH3D identificados en el modelo extendido. Esto se logra a partir del método estático *Export()* de la clase **Exporter** (el cual se explica a continuación).

Una vez instanciado el modelo final, se pasa como parámetro el resultado en la invocación del método *writeObject(Object object)* sobre la instancia de la clase **FileOutputStream**. Esto finalizará el proceso creando un archivo XML con el modelo final transcripto en él.

Clase Exporter y su método estático *Export()*

Es importante destacar al método de esta clase, ya que serán los encargados de, a partir del modelo extendido que se fue instanciando con el uso de la herramienta, instanciar el modelo final.

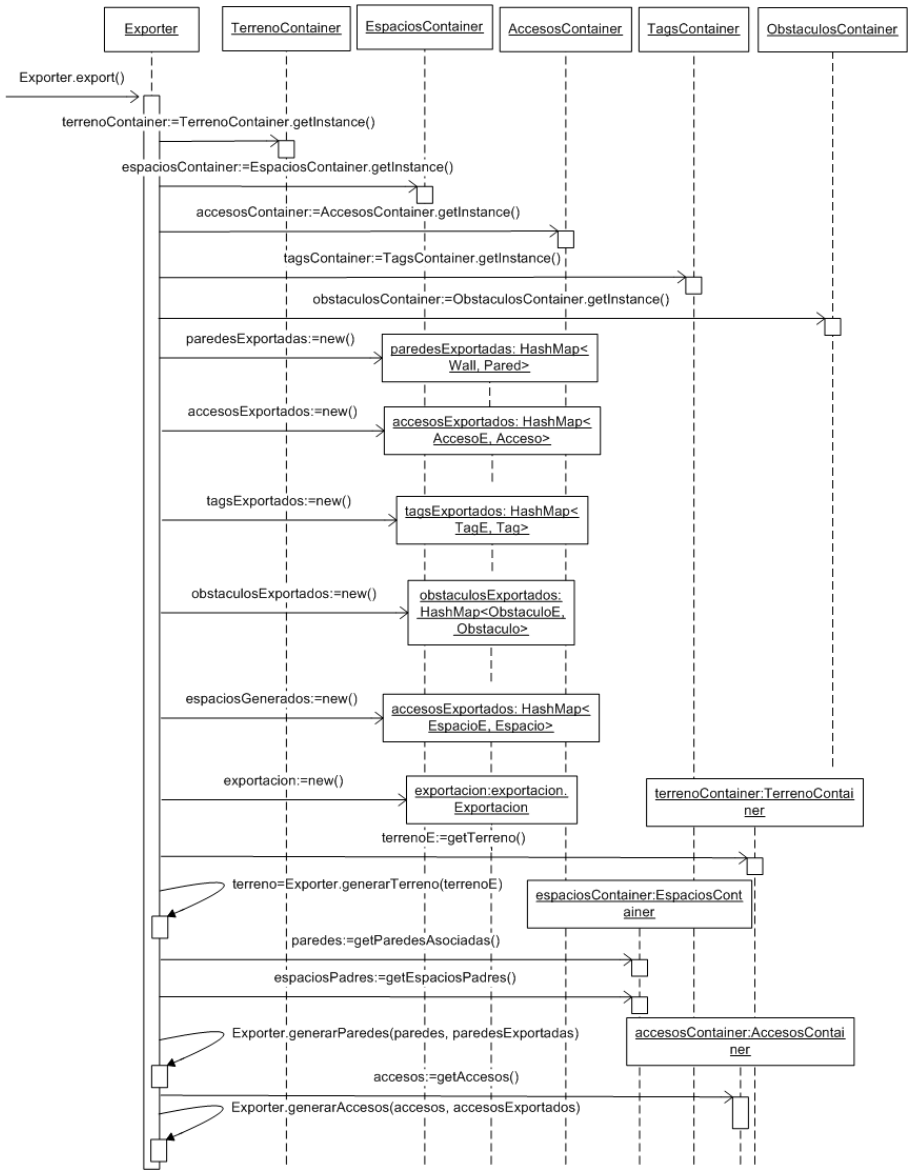


Figura 4-19: diagrama de secuencia del método estático export() de la clase Exporter

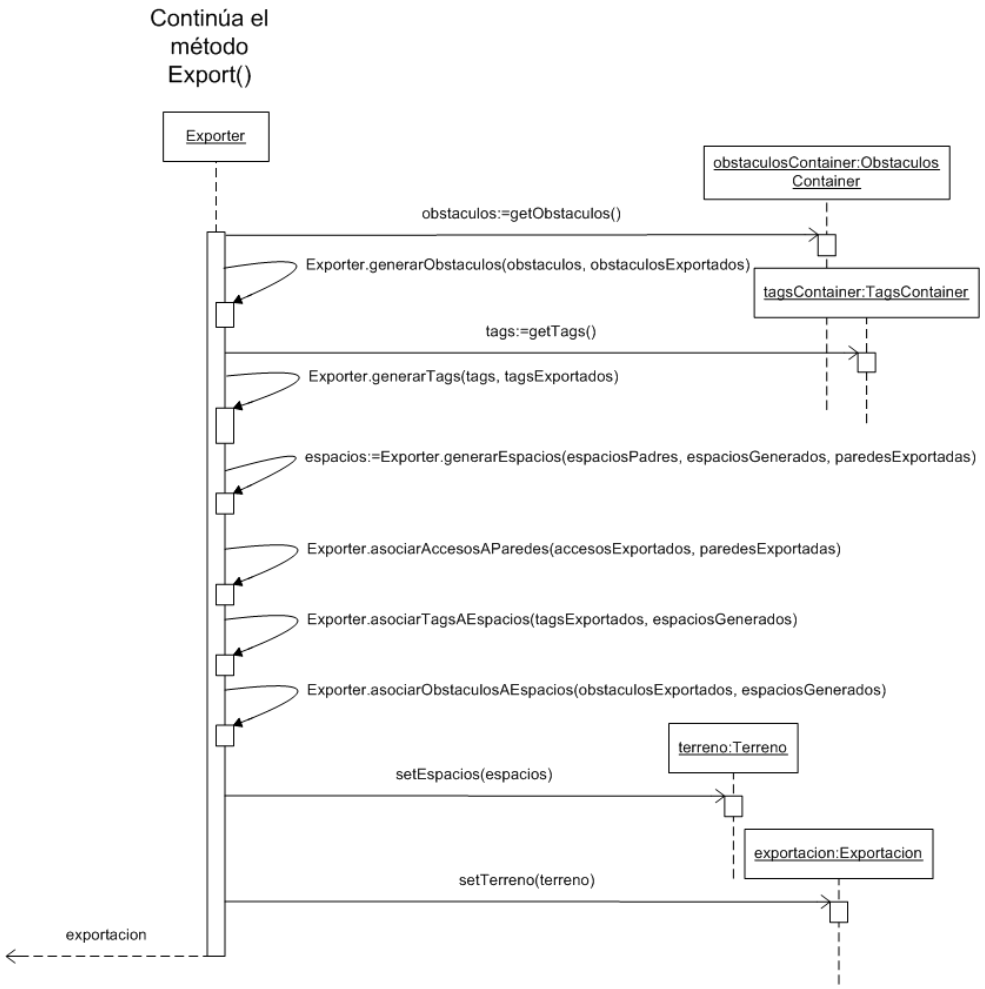


Figura 4-20: diagrama de secuencia del método estático export() de la clase Exporter (continuacion)

Como se muestra en la Figura 4-19 y 4-20, el resultado final será una instancia de la clase **Exportacion**, la cual será el punto de entrada al modelo final instanciado. Para instanciar el nuevo modelo y relacionar los objetos del mismo, se optó por separar el proceso en dos pasos. El primero será crear las instancias y el segundo relacionarlas.

Se comienza instanciando la clase **Terreno**. Como se observa en la Figura 4-19, se procede invocando al método estático *generarTerreno(TerrenoE terreno)*, pasando como parámetro la instancia de la clase **TerrenoE** recuperada de la instancia de la clase **TerrenoContainer.** Este método no hace más que invocar al método estático *transform(Room terreno)* de la clase **TerrenoExporter** (Figura 4-21) el cual retorna la instancia de la clase **Terreno** que se corresponde con la instancia de la clase **TerrenoE**. La nueva instancia del modelo final es independiente del modelo SH3D y se guarda localmente en el método *Export()* para las futuras relaciones.

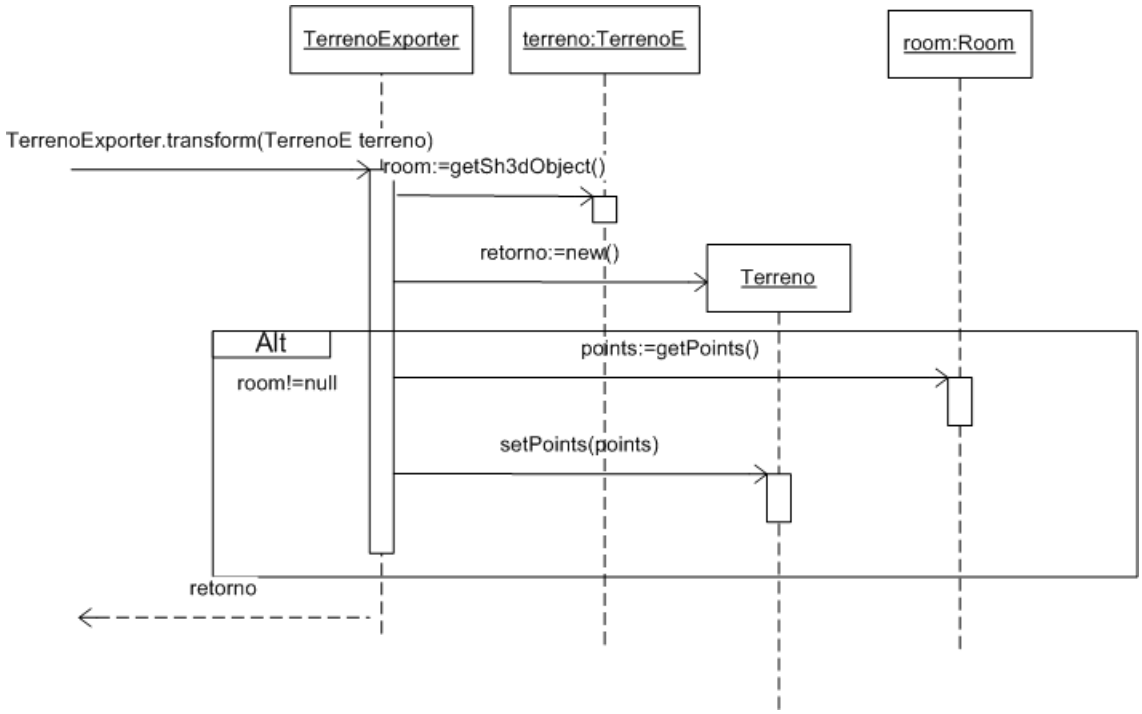


Figura 4-21: diagrama de secuencia del método estático transform() de la clase TerrenoExporter

De la misma manera (y como se puede observar en el Anexo A) se crean las instancias de la clase **Pared**, **Acceso**, **Obstaculo** y **Tag**. Primero se obtienen las instancias creadas del modelo extendido y a partir de ellas se generan las instancias del modelo final.

Luego se procede con la instanciación de los espacios. Como se muestra en la Figura 4-22, para cumplir con esta tarea, se invoca al método estático *generarEspacios(List<EspacioE> espacios, HashMap<EspacioE, Espacio> espaciosGenerados, HashMap<Wall, Pared> paredesExportadas)* de la clase **Exporter.**

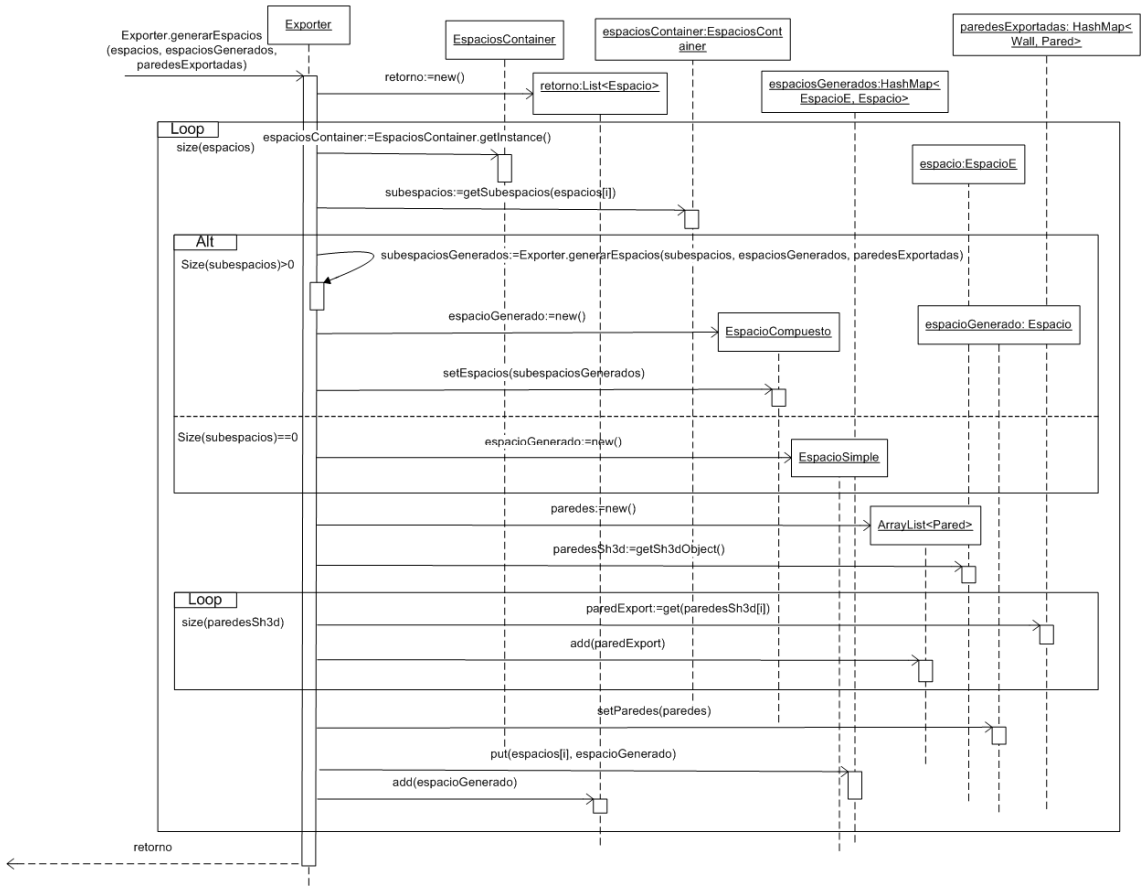


Figura 4-22: diagrama de secuencia del método estático generarEspacios() de la clase Exporter

Dicho método se invoca recursivamente para crear la estructura de espacios de nuestro modelo. La primera invocación recibe como primer parámetro (*espacios*) la lista de instancias de la clase **EspacioE** cuya área no está incluida en ningún otro espacio. Este conjunto de instancias se recupera, como se observa en la Figura 4-20, invocando al método *getEspaciosPadres()* sobre la instancia de la clase **EspaciosContainer.** El segundo parámetro será un HashMap vacío (el cual almacenará las instancias de la clase **Espacio** que se irán creando relacionado con su instancia respectiva de la clase **EspacioE**) y en el tercero la lista de instancias de la clase **Pared** de nuestro modelo creadas anteriormente.

El método *generarEspacios(List<EspacioE> espacios, HashMap<EspacioE, Espacio> espaciosGenerados, HashMap<Wall, Pared> paredesExportadas)* itera sobre el primer parámetro. Lo primero que hace es crear la instancia de la subclase de **Espacio** que corresponda. En caso de que el espacio tenga espacios incluidos se crea una instancia de la clase **EspacioCompuesto.** En caso contrario se crea una instancia de la clase **EspacioSimple.** En ambos casos se relacionan con las instancias de la clase **Pared** creadas anteriormente (las cuales se recuperan del parámetro *paredesExportadas)*.

Para decidir que subclase de la clase **Espacio** de nuestro modelo se debe instanciar, se recuperan los sub espacios de la instancia actual de iteración.

Si retorna uno o más espacios, se instancia la clase **EspacioCompuesto,** y se relaciona con las instancias de la clase **Espacio** que se crean a partir de la invocación recursiva del método *generarEspacios(List<EspacioE> espacios, HashMap<EspacioE, Espacio> espaciosGenerados, HashMap<Wall, Pared> paredesExportadas),* excepto que en esta oportunidad se pasa como primer parámetros el conjunto de los sub espacios recuperados. En caso de no existir sub espacios, se instancia la clase **EspacioSimple** y continúa con la iteración.

Una vez finalizada la primera invocación al método, la lista de instancias de la clase **Espacio** que se retorna contendrá todas las instancias cuya área no está incluida en ningún otro espacio, y cada uno de ellos estará compuesto por otras instancias de la misma clase, además de estar compuestos por instancias de la clase **Pared.** De esta manera, además de instanciar los espacios, se comienza con las relaciones del modelo final (instancias de las sub clases de **Espacio** relacionadas entre sí y con instancias de la clase **Pared**).

Como siguiente paso, se relacionan las instancias de la clase **Acceso** con las instancias de la clase **Pared.** Con el objetivo de cumplir este requerimiento se invoca al método estático *asociarAccesosAParedes(HashMap<AccesoE, Acceso> accesosExportados, HashMap<Wall, Pared> paredesExportadas)* (Figura 4-23), el cual itera sobre la lista de instancias de la clase **AccesoE** (conjunto de claves del parámetro *accesosExportados)*. Por cada una de ellas recuperamos la instancia de la clase **Acceso** que la representa en el modelo final (y guardada en el parámetro *accesosExportados)* y la instancia de la clase **Wall** que posee en su atributo *wall.* A partir de esta última obtenemos la instancia de la clase **Pared** que la representa en el modelo final (guardada en el parámetro *paredesExportadas)*. Una vez recuperadas las instancias de nuestro modelo final, las relacionamos de manera tal que cada instancia de la clase **Pared** estará compuesta por las instancias de la clase **Acceso** que corresponda según el modelo extendido.

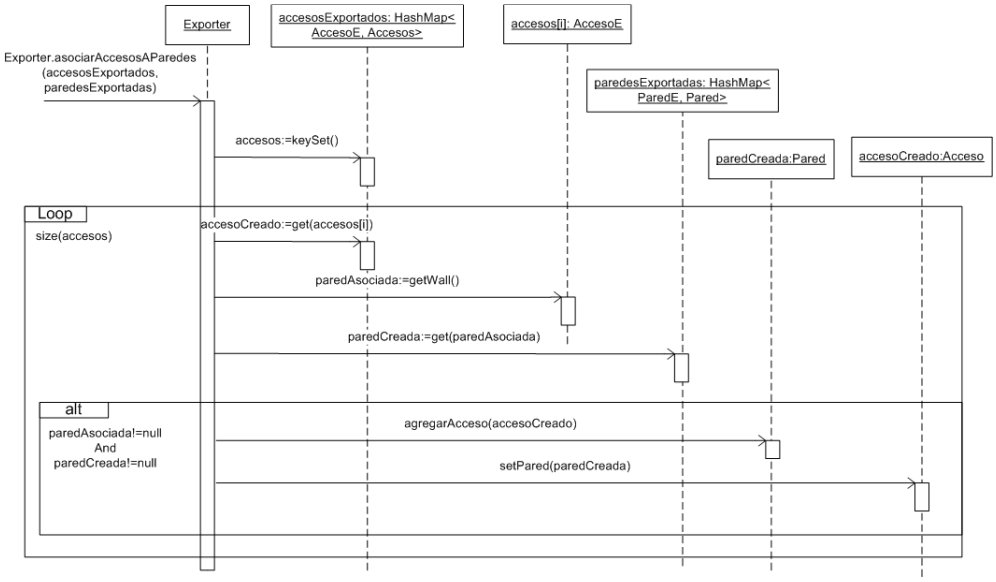


Figura 4-23: diagrama de secuencia del método estático asociarAccesosAParedes() de la clase Exporter

De manera similar se relacionan las instancias de la clase **Tag** (Figura 4-24) y de la clase **Obstaculo** (de manera idéntica a la Figura 4-24, pero invocando el método *asociarObstaculosAEspacio(HashMap<ObstaculoE, Obstaculo> obstaculosExportados, HashMap<EspacioE, Espacio> espaciosGenerados),* el cual se puede observar en el Anexo A) con las instancias de la clase **Espacio**, siempre cumpliendo las relaciones identificadas en el modelo extendido.

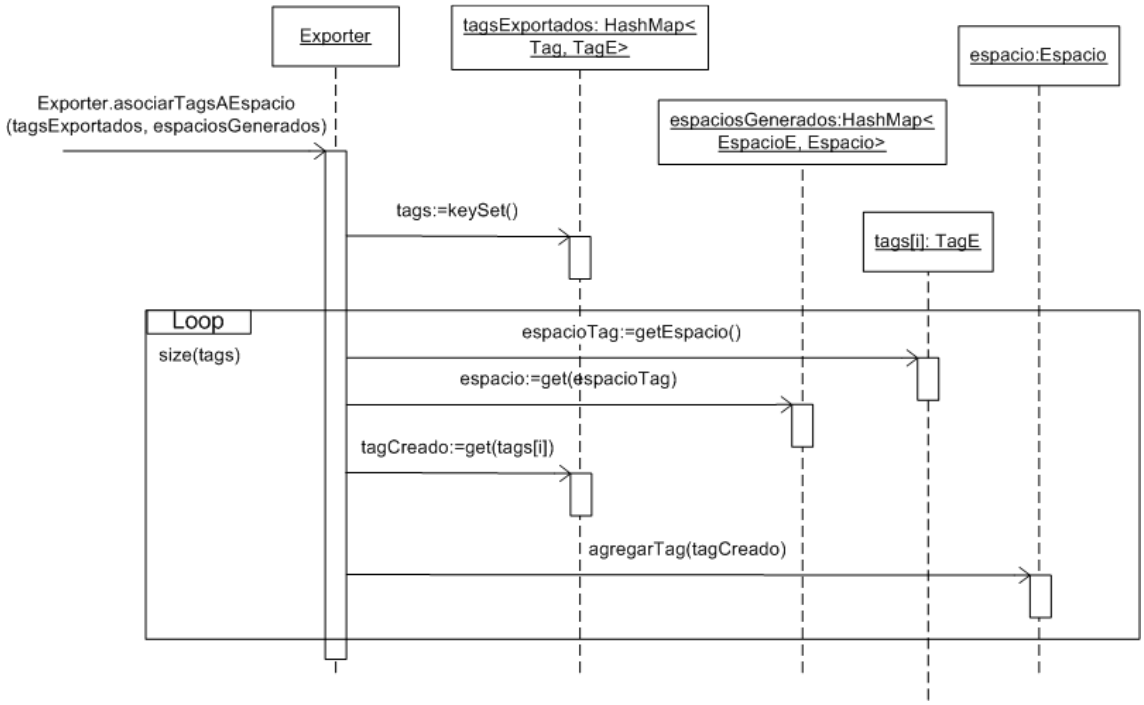


Figura 4-24: diagrama de secuencia del método estático asociarTagsAEspacio() de la clase Exporter

Para finalizar y retomando la Figura 4-19, se relaciona la lista de instancias de las sub clases de **Espacio** cuya área no está incluida en ningún otro espacio, con la instancia de la clase **Terreno**. De esta forma se habrán creado todos los objetos de nuestro modelo final y relaciones entre ellos. El método *Export()* retorna una instancia de la clase **Exportacion** que incluye la instancia de la clase **Terreno**.

CAPITULO 5 - USO DE LA HERRAMIENTA DESARROLLADA

En los capítulos anteriores se explicó cuáles son las características necesarias para que un plano sea transitable, y el modelo desarrollado para definirlo. En esta sección desarrollaremos un ejemplo de cómo transformamos un plano estructural en un plano transitable con nuestra aplicación.

Si bien a continuación explicamos el proceso de transformación en un orden determinado, vale aclarar que ese orden puede variar, siempre y cuando se respeten las restricciones explicadas en el capítulo anterior (por ejemplo, un tag deberá ser identificado luego de haberse identificado el espacio al que pertenece, o un espacio luego de identificarse el terreno).

La Figura 5-1 nos muestra el software (SH3D) en su estado inicial, sin ninguna función adicional. Como ya henos mencionado, esta herramienta permite definir la estructura del plano, por lo que las herramientas disponibles están relacionadas con esta facilidad. Además se presenta el plano que se utilizará como ejemplo para la transformación.

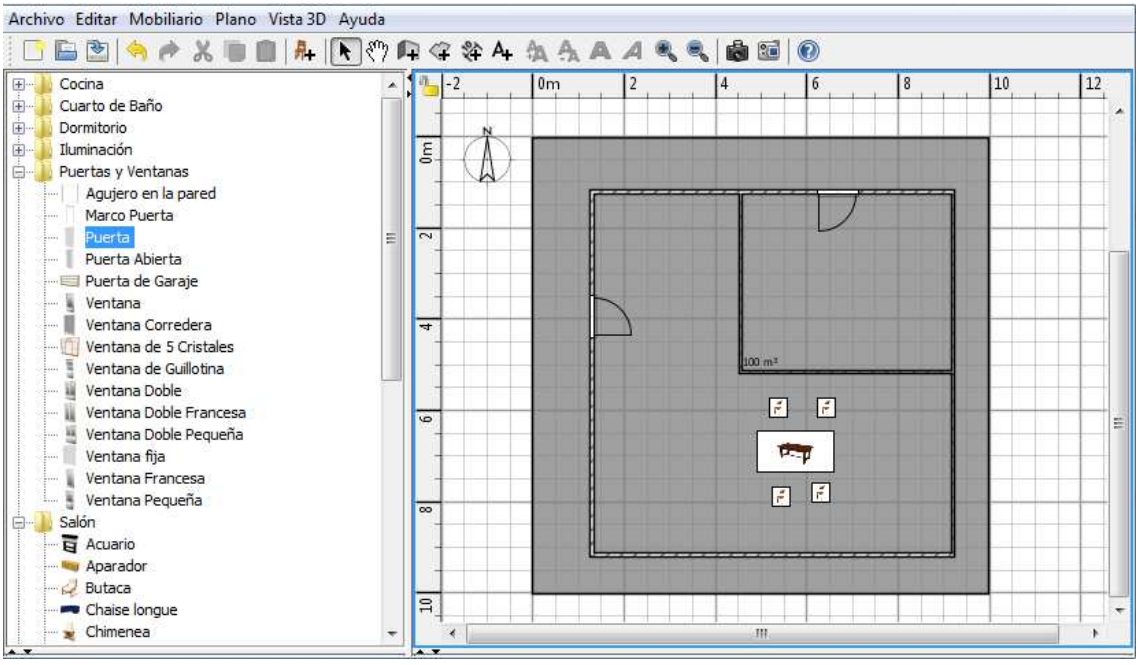


Figura 5-1: plano a transformar

En la Figura 5-2 se presenta la herramienta desarrollada, recuadradas en azul, se ve el nuevo menú a través del cual se implementan las nuevas funciones.

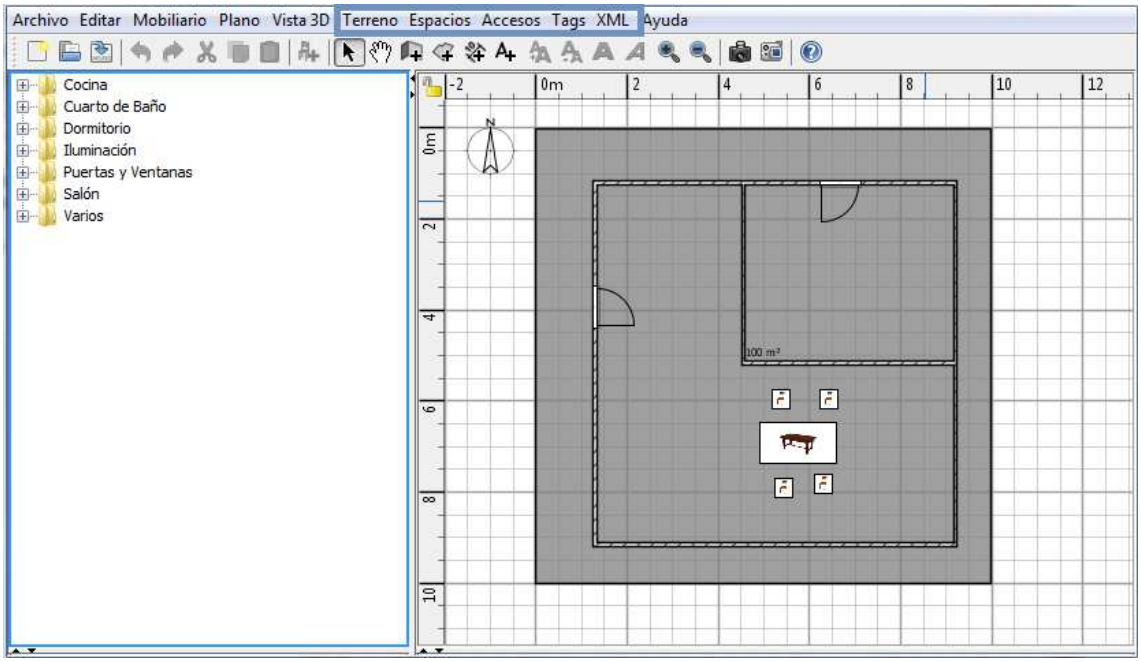


Figura 5-2: menú de funciones de nuestra herramienta

La Tabla 5-1 muestra los menús que fueron agregados al SH3D y sus respectivas funciones.

Menú	**Funciones**
Terreno	• Asociar terreno • Desasociar terreno • Marcar terreno
Espacios	• Generar espacio • Eliminar espacio • Marcar espacio padre • Dividir paredes seleccionadas • Dividir todas las paredes
Accesos	• Asociar elementos seleccionados • Bloquear/desbloquear acceso • Ver pared a la que pertenece el acceso • Desasociar elementos seleccionados • Desasociar todos los accesos • Ver accesos asociados
Tags	• Asociar elementos seleccionados • Cargar código • Desasociar elementos seleccionados • Ver tags asociados • Desasociar todos los tags • Ver espacio al que pertenece el tag
XML	• Exportar modelo a XML

Tabla 5-1: menús agregados al SH3D y sus respectivas funciones

La inclusión del plugin desarrollado incorpora el tag como un nuevo elemento en la aplicación, esto se puede ver en el menú de la Figura 5-3. Además se ha incorporado en esta figura un nuevo mueble al plano, mientras que en la Figura 5-4 se puede observar que se adicionó un Tag.

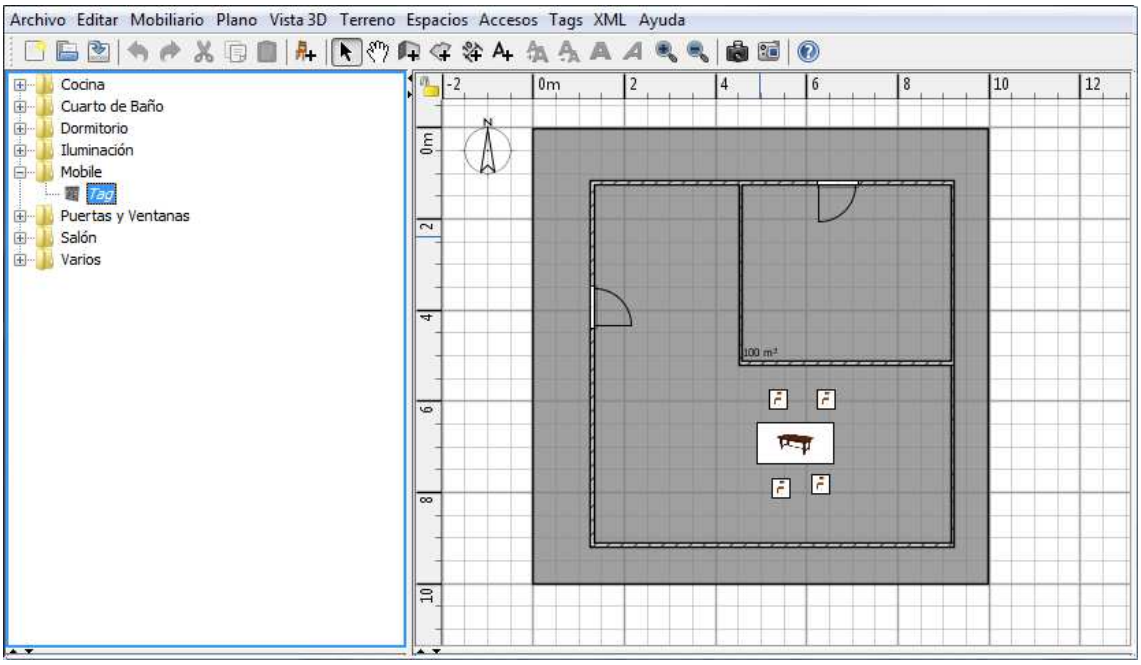


Figura 5-3: selección de un tag en el catálogo de muebles

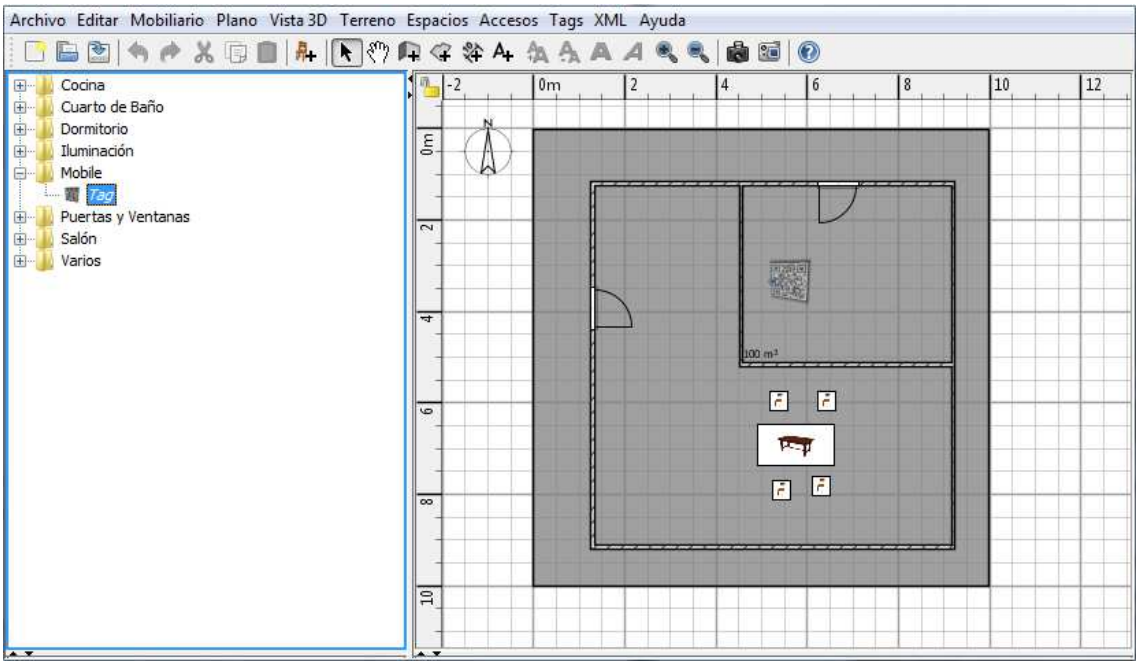


Figura 5-4: tag agregado en el plano

A partir de este momento, se puede comenzar a transformar el plano, en términos de los elementos definidos par un plano transitable.

Comenzamos seleccionando la habitación que será identificada como terreno, y nos dirigimos al menú “Terreno”-> “Asociar como terreno”, tal como se muestra en la Figura 5-5a. Si dicho elemento cumple con los requisitos que debe poseer un terreno, la identificación quedará guardada (Figura 5-5b). En este ejemplo seguiremos como si esto hubiera ocurrido.

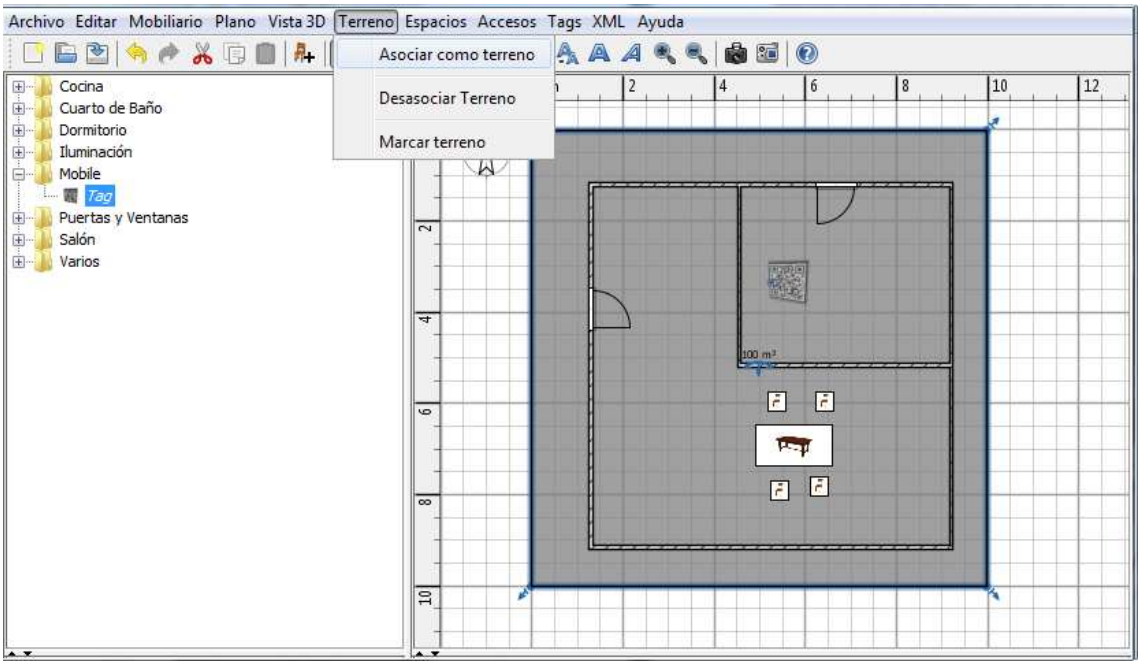


Figura 5-5a: identificación del terreno

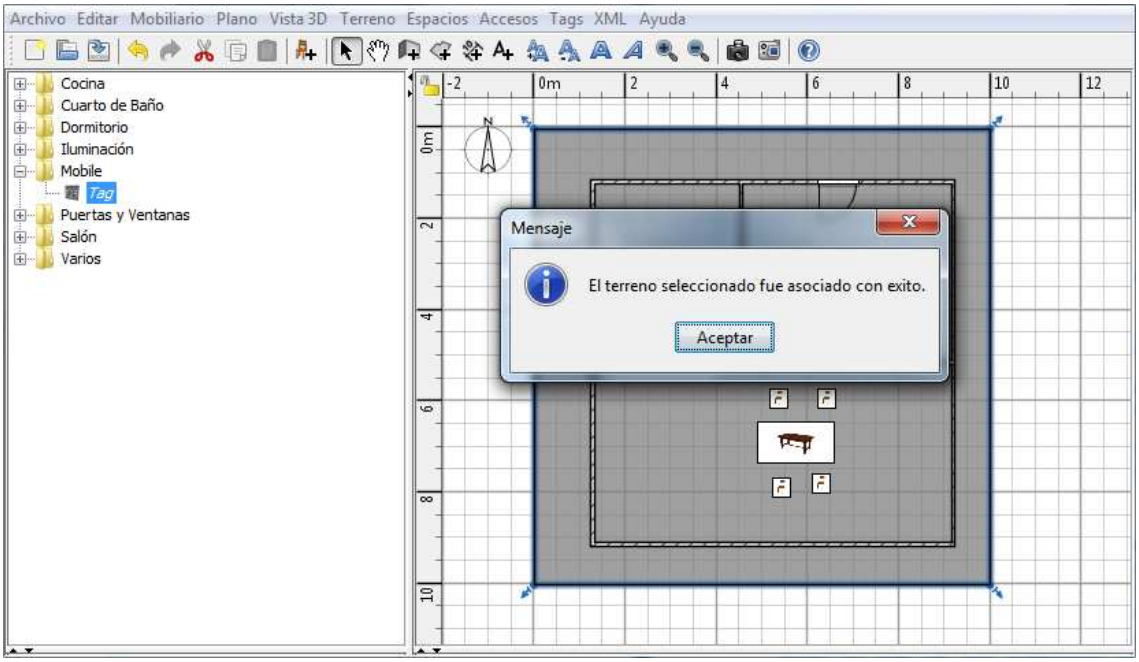


Figura 5-5b: resultado satisfactorio de la identificación del terreno

Como se puede apreciar en la Figura 5-5a, el menú de Terreno provee 2 funcionalidades mas:

- **Desasociar Terreno:** dejará sin efecto la identificación del terreno previamente realizada (solo en caso de haber sido realizada).
- **Marcar Terreno:** seleccionará en el plano la habitación que fue identificada como terreno previamente (solo en caso de existir).

Luego procedemos definiendo los espacios, los cuales deberán identificarse uno por uno. Para esto seleccionamos las paredes del mismo en el plano, y vamos al menú “Espacios” -> “Generar Espacio”, tal como se muestra en la Figura 5-6a. Si la selección cumple con los requisitos que debe poseer un espacio, la identificación quedará guardada (Figura 5-6b). Nuestro ejemplo considera espacios bien formados, pero vale aclarar que en caso de no serlos, se deberán modificar las paredes que estén mal. Estas modificaciones no afectan a los espacios previamente identificados que no contienen a la pared, pero si a los que SI la tienen entre sus paredes. En caso de modificar una pared que pertenece a un espacio previamente identificado, éste será controlado para corroborar que sigue cumpliendo los requisitos de un espacio. En caso de no cumplirlos se eliminará al mismo de la lista de espacios identificados. De cualquier forma, en este ejemplo no tendremos en cuenta modificaciones de paredes y supondremos que los espacios están todos bien formados desde el vamos.

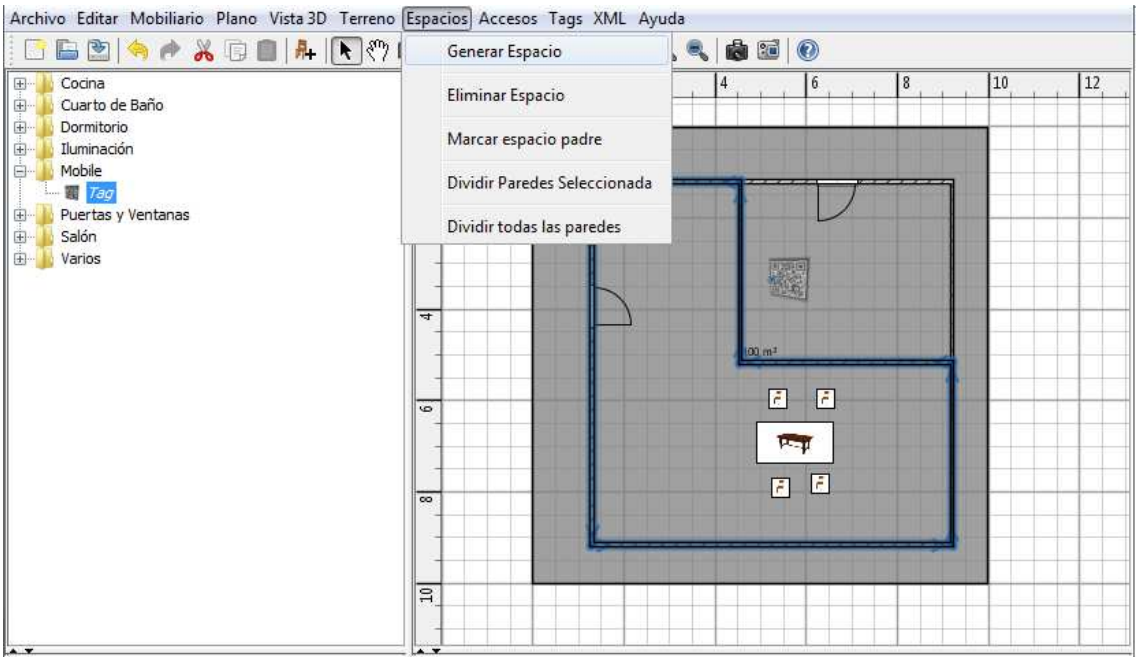


Figura 5-6a: identificación de un espacio

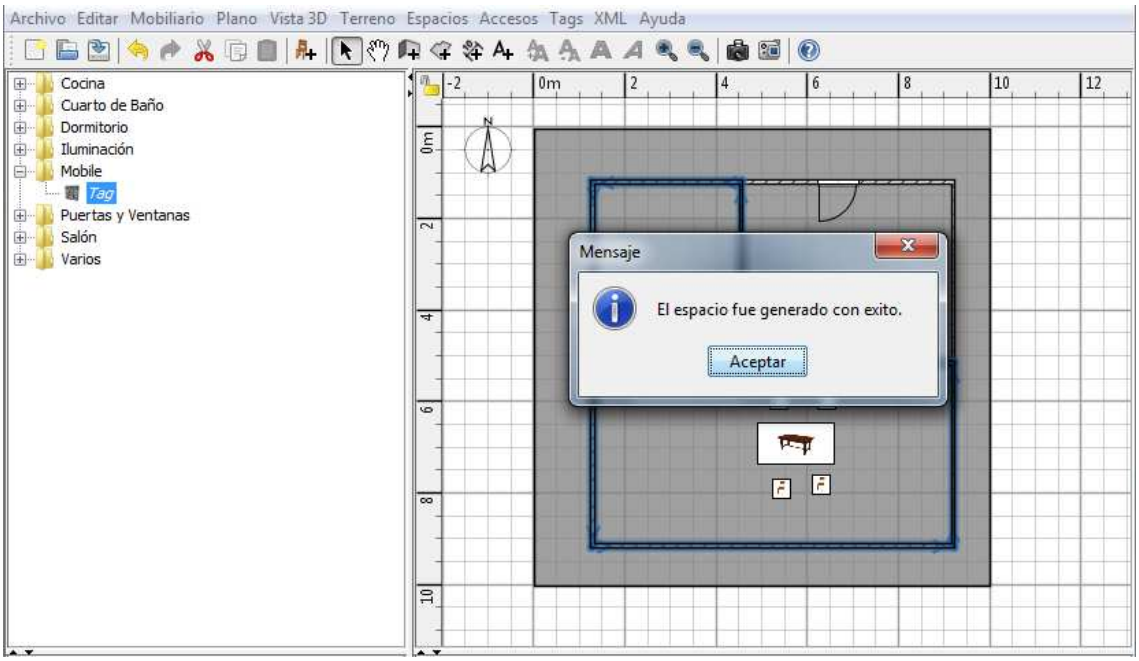


Figura 5-6b: resultado satisfactorio de la identificación de un espacio

Identificado el primer espacio, existen dos caminos a seguir. Podemos empezar a identificar accesos y tags que se encuentren en el mismo, o identificar el resto de los espacios. Nuestro ejemplo seguirá el segundo camino. Para esto repetimos la misma metodología que la explicada anteriormente para cada uno de los espacios.

Como se puede aprecia en la Figura 5-6a, el menú de Espacios provee 4 funcionalidades mas:

- **Eliminar Espacio:** a partir de un conjunto de paredes seleccionadas en el plano, se dejará sin efecto la identificación previa del espacio que forman las mismas (en caso de haber sido realizada previamente)
- **Marcar Espacio Padre:** a partir de un conjunto de paredes seleccionadas en el plano, selecciona otro conjunto de paredes que formen un espacio, hayan sido identificadas como tal y que lo contengan. En caso de no existir se informa la situación.
- **Dividir paredes Seleccionadas:** función que a partir de una pared seleccionada en el plano, divide a la misma en los puntos de intersección con las demas paredes.
- **Dividir todas las paredes:** divide todas las paredes del plano en las intersecciones entre ellas.

A continuación nos embarcaremos en identificar los accesos a cada uno de los espacios. Tal como se muestra en la Figura 5-7a, seleccionamos las puertas y ventanas que serán identificadas como accesos y vamos al menú “Accesos” -> “Asociar elementos seleccionados”. Si alguno de los elementos seleccionados no cumplen los requisitos para ser identificados

como accesos, se notificará en pantalla, en caso contrario, la identificación quedará almacenada (Figura 5-7b).

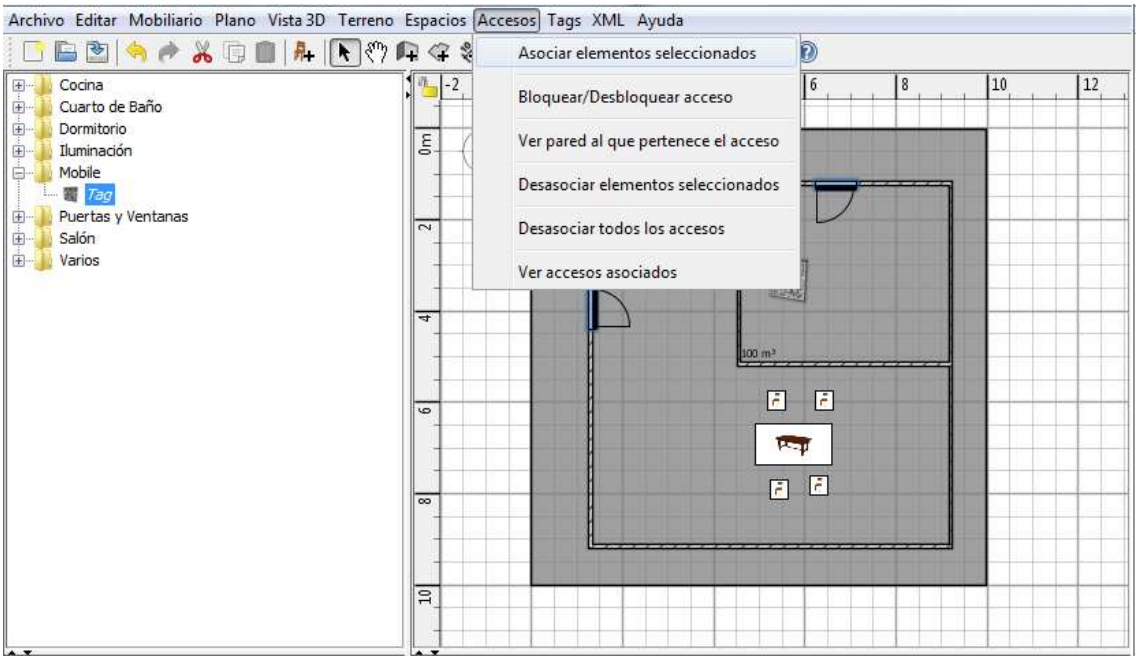


Figura 5-7a: identificación de accesos

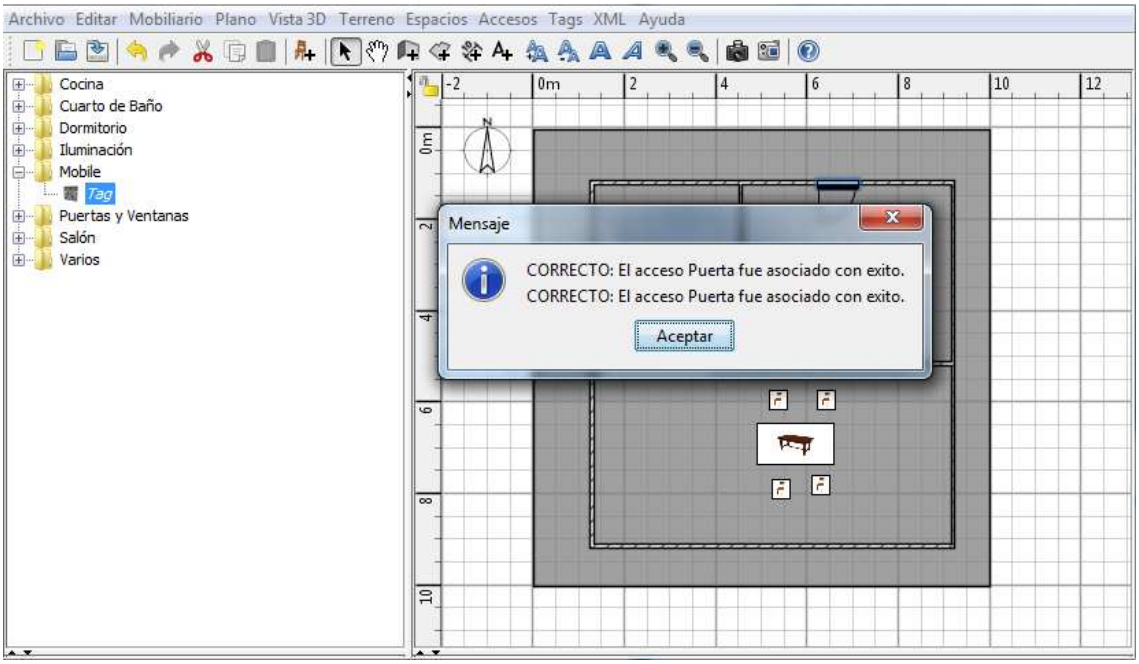


Figura 5-7b: resultado satisfactorio de la identificación de accesos

Como se puede aprecia en la Figura 5-7a, el menú de Accesos provee 5 funcionalidades mas:

- **Bloquear/Desbloquear Acceso:** a partir de un acceso seleccionado, el cual debe haber sido identificado como tal previamente, se le setea el atributo bloqueado en true en caso de no haber sido bloqueado antes y false en caso contrario. Por defecto, un acceso se crea con el atributo *bloqueado* en false. En el ejemplo bloquearemos uno de los accesos.

- **Ver Pared al que pertenece el acceso:** a partir de un acceso seleccionado, el cual debe haber sido identificado como tal previamente, se selecciona en el plano la pared a la que el acceso esta asociado.

- **Desasociar elementos seleccionados: :** a partir de un conjunto de accesos seleccionados, los cuales deben haber sido identificados como tales previamente, se deja sin efecto dicha identificación.

- **Desasociar todos los accesos:** elimina todas las identificaciones de accesos hechas hasta el momento.

- **Ver accesos asociados:** selecciona en el plano las puertas y ventanas que hayan sido identificadas como accesos hasta el momento.

Por último, antes de exportar el plano, nos falta identificar los tags del mismo. Tal como se muestra en la Figura 5-8a, seleccionamos los elementos que serán identificadas como tags y vamos al menú “Tags” -> “Asociar elementos seleccionados”. Si alguno de los elementos seleccionados no cumplen los requisitos para ser identificados como tags, se notificará en pantalla, en caso contrario, la identificación quedará almacenada (Figura 5-8b).

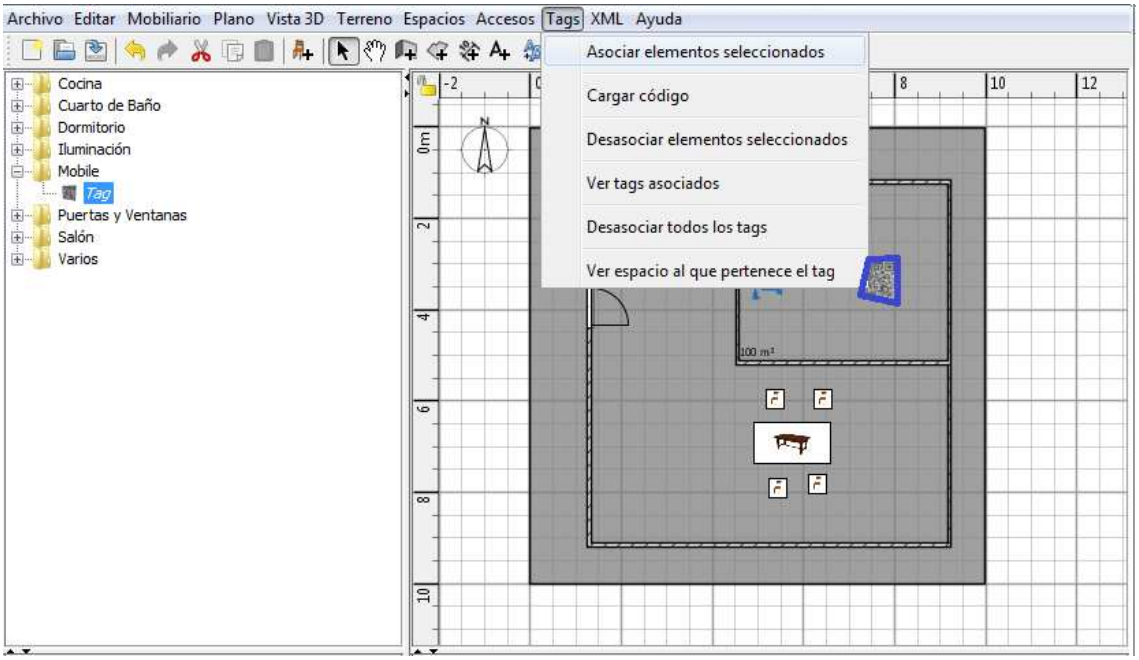


Figura 5-8a: identificación de tags

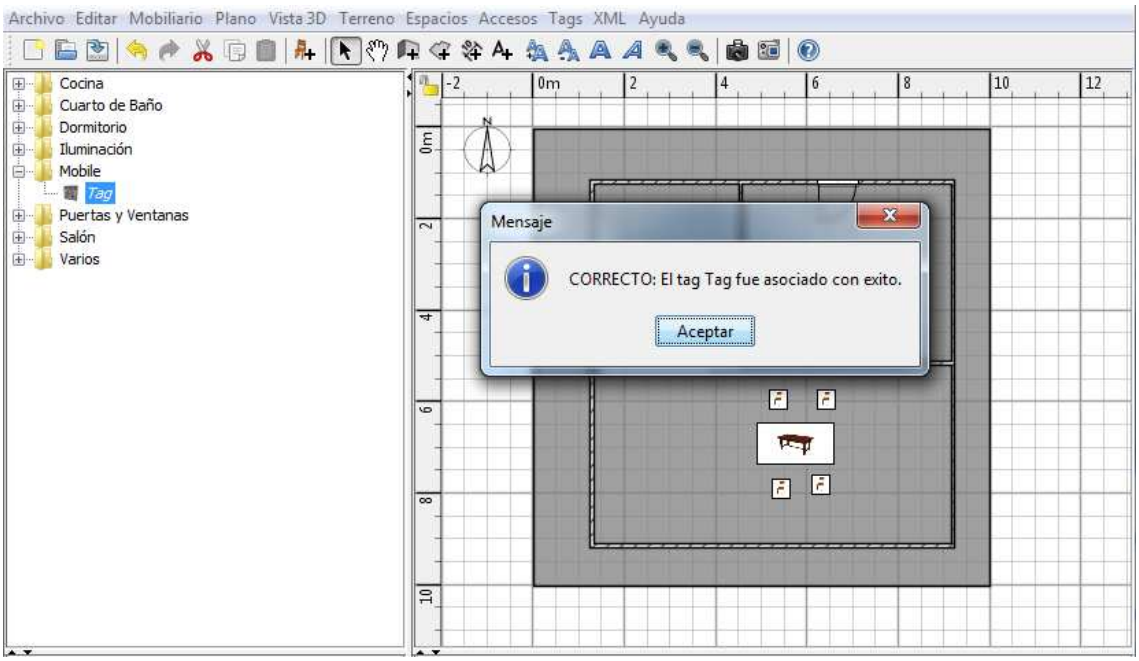


Figura 5-8b: resultado satisfactorio de la identificación de tags

Como se puede aprecia en la Figura 5-8a, el menú de Tags provee 5 funcionalidades mas:

- **Cargar Código:** a partir de un tag previamente identificado seleccionado en el plano, se permitirá ingresarle un código que información especifica del mismo. En el ejemplo cargaremos a cada tag un código particular, diferenciandolos en el nombre de cada uno de ellos (la Figura 5-9 muestra la pantalla que ingresa el código en el tag seleccionado).

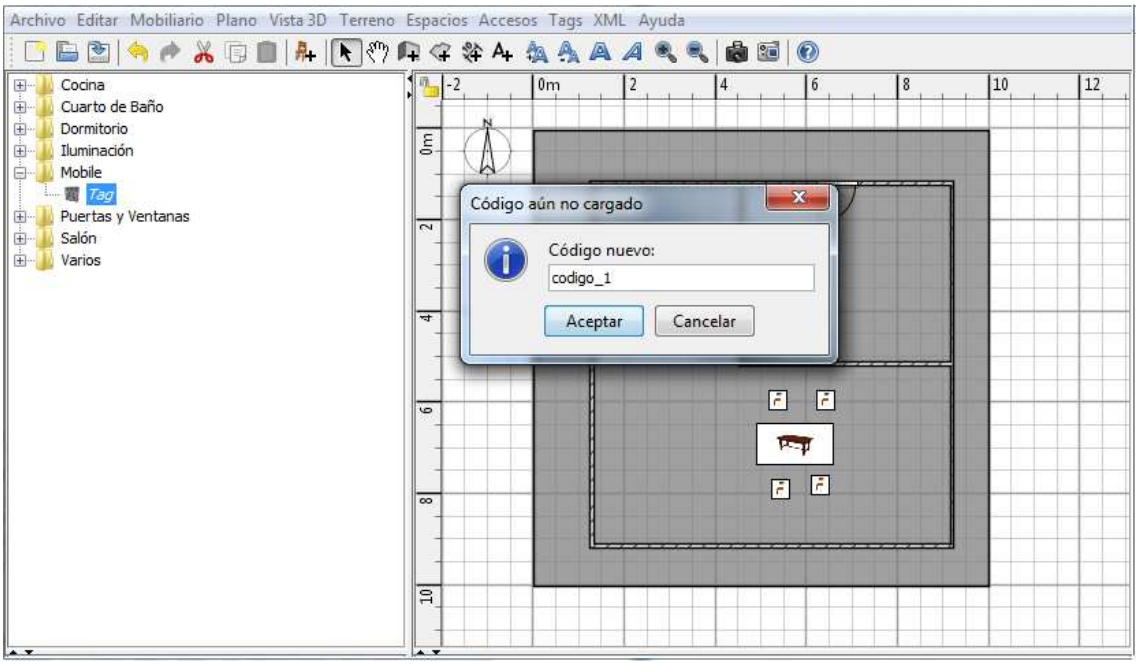


Figura 5-9: carga del código de un tag

- **Desasociar elementos seleccionados:** a partir de un conjunto de tags seleccionados, los cuales deben haber sido identificados como tales previamente, se deja sin efecto dicha identificación.
- **Ver Tags asociados:** selecciona en el plano los elementos que hayan sido identificadoas como tags hasta el momento.
- **Desasociar todos los tags:** elimina todas las identificaciones de tags hechas hasta el momento.
- **Ver espacio al que pertenece el tag:** a partir de un elemento seleccionado en el plano que haya sido previamente identificado como tag, se selecciona el conjunto d eparedes que forman el espacio al que el tag pertenece y esta incluído.

Una vez finalizado este proceso de identificar las estructuras necesarias se procede a exportar el mismo. Vale aclarar que los elementos que son obstáculos quedarán automáticamente identificados durante este proceso de exportación. Como se muestra en la Figura 5-10a, vamos al menú “XML” -> “Exportar Modelo a XML”.

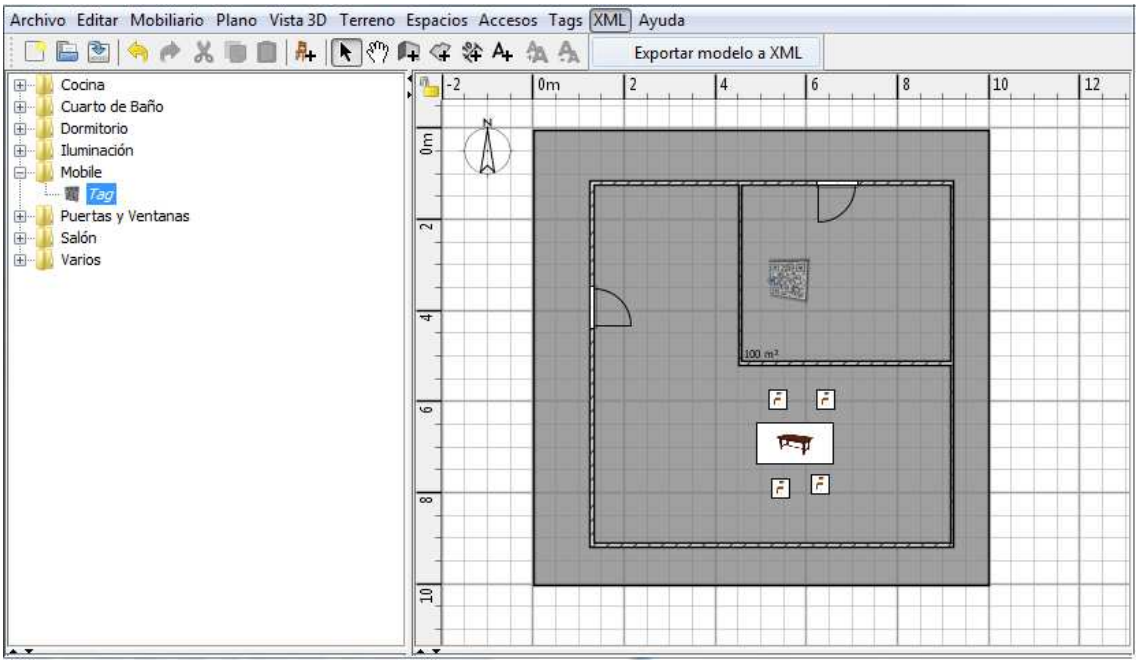


Figura 5-10a: exportación del plano

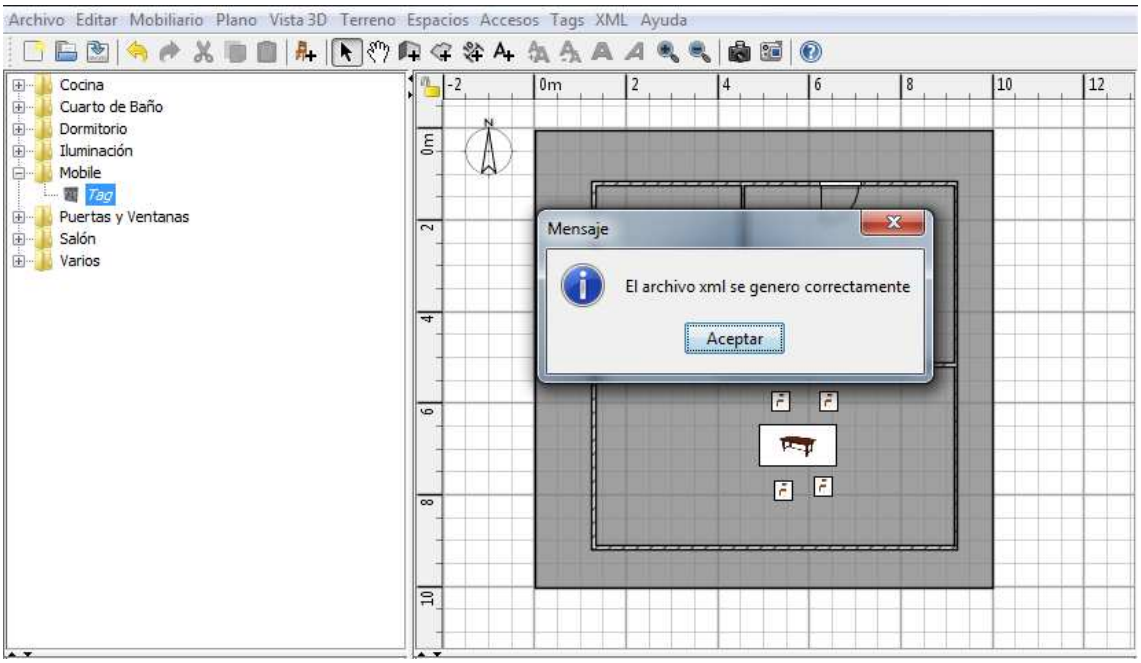


Figura 5-10b: resultado satisfactorio de la exportación del plano

En caso de que haya elementos que puedan ser identificados como obstáculos y no cumplan los requisitos, se notificará en pantalla. De cualquier modo la exportación se realiza sin incluir dichos elementos. Luego se selecciona la ubicación del archivo donde exportará el plano y se acepta. Si nos dirigimos a esta ubicación veremos que se creó un archivo XML (el código de este archivo se puede observar en el Anexo B). Como se observa en la Figura 5-10b, se notifica la correcta ejecución de la exportación.

CAPITULO 6 - CONCLUSIONES Y TRABAJOS A FUTURO

Investigando el contexto de las aplicaciones de navegación indoor, encontramos que uno de los problemas en este dominio, es la generación de planos navegables. La característica de estos planos, es que no se reducen a un "dibujo" o esquema como en los planos convencionales, sino que deben estar geo-referenciados, de manera tal que sea posible ubicar al usuario, para brindarles servicios de localización. Este posicionamiento puede ser de tipo global o local, pero lo cierto es la tecnología necesaria para definirlos es más compleja.

A partir del problema planteado, y, teniendo en cuenta la falta de herramientas para el desarrollo y diseño de estos planos navegables, así como la existencia de tantas otras para la creación de planos convencionales, comenzamos a investigar sobre la posibilidad de transformar un plano convencional a un plano navegable.

Lo primero que hicimos fue buscar una herramienta basada en software libre que nos permita adaptar el plano resultante a nuestras necesidades con el menor costo posible, utilizando el código generado por la misma. Fue aquí que encontramos el SH3D, un software que no solo es libre, sino también que nos da la posibilidad de agregarle funcionalidad sin necesidad de tocar el código fuente pero si utilizando su modelo de datos.

El SH3D nos provee una herramienta gráfica para crear cualquier tipo de plano convencional, y a partir de un mecanismo de plugins utilizar la información creada con el objetivo que deseemos. Debido a la imposibilidad de detectar automáticamente las estructuras necesarias para un plano navegable en el modelo de SH3D, se creó un plugin capaz de determinar cual información del plano convencional cumplirá un rol en el plano navegable y cual será ese rol.

Se comenzó creando un modelo con las clases necesarias para permitir la navegación indoor del usuario. En dicho modelo existen entidades que definen: terreno, el cual delimita el área de recorrido del plano, espacios, los cuales tienen accesos para ingresar, obstáculos que cortan el paso y elementos denominados tags que funcionarán como identificadores de lugares (puntos de interés) y que permitirán ubicar al usuario dentro del plano.

Una vez creado el modelo necesario para hacer a un plano navegable, codificamos un plugin que permite identificar cuáles estructuras creadas con el SH3D formarán parte de nuestro plano navegable y cuál será su función. El SH3D utiliza Java como software de base, por lo tanto, este plugin, se desarrolló también en Java para compatibilizar el funcionamiento. Además, una vez creado el mismo, se lo puede exportar a un formato standard como es XML.

Si bien el plugin creado nos provee las funcionalidades básicas para crear un plano navegable, es importante remarcar que no se modificó el código fuente del SH3D, lo que nos da la posibilidad de utilizar cualquier versión futura del software (siempre y cuando mantenga dicho mecanismo de extensión) y aprovechar las mejoras del mismo.

Además, el costo de dibujar un plano navegable, ahora no es más que dibujar un plano convencional con SH3D, y detectar a partir del mismo las nuevas estructuras. Esto nos ahorra la utilización de un software de dibujo, el cual es un trabajo largo y engorroso.

Con respecto a los trabajos futuros, si bien las funcionalidades básicas fueron abarcadas en el desarrollo de nuestra tesis, quedaron pendientes en este trabajo, un conjunto de funcionalidades que pueden ser de mucha utilidad más adelante y que se describen a continuación:

- **Guardar un plano parcial**: sería una buena herramienta a futuro poder guardar un trabajo en un momento determinado con el objetivo de retomarlo en otro momento. Si bien el plano SH3D se puede guardar y retomar más adelante, en el desarrollo realizado, las identificaciones de estructuras navegables realizadas se pierden en el momento en que el software se cierra; esto implica que la información sobre las nuevas estructuras definidas se pierde.

- **Deshacer y rehacer identificaciones de estructuras:** si bien SH3D provee un mecanismo de rehacer y deshacer cambios, en nuestro trabajo tuvimos que dejarlo de lado porque nos alejábamos de nuestro objetivo principal. Sin embargo, la programación de nuestro plugin y el modelo de SH3D nos permite en un futuro agregar esta funcionalidad en la identificación de estructuras navegables.

- **Dividir paredes que se cruzan entre sí**: este punto fue encarado en un principio y parcialmente realizado (el plugin creado incluye la funcionalidad de división automática de paredes), pero esta funcionalidad necesita ser analizada con más detalle para resolver criterios de división y corrección de esa división, que, como no constituían el objetivo principal de la tesis, decidimos no desarrollar. Esta funcionalidad es útil cuando queremos transformar un plano donde existen paredes que se cruzan en puntos que no son extremos de las mismas. Esto nos permitiría utilizar las paredes resultantes en la identificación de espacios.

- **Ampliar datos de las estructuras identificadas:** cualquier dato necesario en las estructuras identificadas y exportadas que aún no estén incluidos podrán ser agregados en un futuro, haciendo hincapié en las necesidades de cada contexto particular.

- **Trabajar con estructuras de muchos niveles:** SH3D nos da la posibilidad de dibujar planos de varios niveles. Nuestro plugin solo trabaja con el nivel 0 o inicial, el cual es la plataforma. Es por esto que en un futuro se podría agregar al mismo la identificación de estructuras en más de un nivel.

- **Agregar un sistema de referencia al plano navegable:** SH3D utiliza el eje cartesiano para dibujar cualquier plano, permitiéndonos usar tanto los ejes positivos como los negativos. En el contexto de nuestro problema cada punto será una coordenada en el mundo. Un buen trabajo a futuro es poder darle al plano navegable la coordenada geográfica que se corresponde con el punto (0, 0) utilizado por SH3D y una escala para transformar entre ambos tipos de localización.

BIBLIOGRAFÍA

[Chang et al., 2007] Chang Y.J, Tsai S.K., Chang Y.S. and Wang T.Y., A Novel Wayfinding System Based on Geo-Coded QR-Codes for Individuals With Cognitive Impairments. Assets '07, Proceedings of the 9th international ACM SIGACCESS conference on Computers and accessibility, 231-232.

[de By et al., 2001] de By Rolf A., Ellis M.C., Georgiadou Y., Kainz W., Knippers R.A., Kraak M.-J., Radwan M.M., Sides E.J., Sun Y., Weir M.J.C. and van Westen C.J., Principles of Geographics Information Systems. ITC Educational Textbook Series 1.

[Dempsey, 2003] Dempsey. M., Indoor Positioning Systems in Healthcare. Radianse Inc. White Paper.

[Forman et al., 1994] Forman G.H. and Zahorjan j., The challenges of mobile computing. Computing Milieux.

[García Gonzáles et al.] García Gonzáles H., de las Heras Villalón G. and San Martín del Pozo S., Sistemas de Localización basados en TOA.

[Gartner et al., 2004] Gartner G., Frank A. and Retscher G., Pedestrian navigation system in mixed indoor outdoor environment - The navio project.

[Google Maps], página Web: http://maps.google.com

[Gu et al., 2009] Gu Y. and Lo A., A Survey of Indoor Positioning System for Wireless Personal Networks. IEEE Communications Survey & Tutorials, Vol. 11, Nº 1.

[Hightower J et al., 2001] Hightower J. and Borrielo G., Location Systems for Ubiquitous Computing.

[Kolozdziej et al., 2006] Kolozdziej K. and Hjelm J., Local Positioning Systems: LBS Applications and Services. CRC press Taylor & Francis Group, Capítulo 5.

[Mautz, 2008] Mautz R., Combination of Indoor and Outdoor Positioning. 1st International Conference on Machine Control Positioning & Guidance.

[Midtbø et al., 2012] Midtbø T., Nossum A.S. Haakonsen T.A. and Nordan R.P.V., Are indoor positioning systems mature for cartographic tasks?. Proceeding – Autocarto 2012.

[Mobile Application], página Web: http://www.techopedia.com/definition/2953/mobile-application-mobile-app

[¿Qué son los servicios basados en localización?], página Web: http://lbspro.com/?q=que-son-servicios-localizacion-LBS

[Soon, 2008] Soon T.J., QR Code. Synthesis Journal 2008, Section 3.

[Streeter et al., 1985] Streeter L.A., Vitello D. and Wonsiewicz S.A., How to Tell People Where to Go: Comparing Navigational Aids. International Journal of Man-Machine Studies. Vol.22, Issue 5, 549-562.

[Sweet Home 3D], página Web: http://www.sweethome3d.com

[Tsetsos et al., 2005] Tsetsos V., Anagnostopoulos C., Kikiras P, Hasiotis P. and Hadjiefthmiades S., A Human-Centered Semantic Navigation System for Indoor Environments. Pervasive Services, 2005. ICPS '05. Proceedings. International Conference On (Julio 2005), 146-155.

ANEXO A – PROTOTIPO

ANEXO A .1 - EXPLOTACIÓN DE MÉTODOS

Explotación del método *asociarTerreno(Room habitacion)* de la clase TerrenoContainer

El método *asociarTerreno(Room habitacion)* (Figura AA-1) identifica como terreno a la instancia de la clase **Room** pasada como parámetro, siempre y cuando cumpla con los siguientes requisitos:

1. No debe existir al momento de ejecutar la acción un terreno identificado.
2. Las líneas que forman los puntos de la habitación no deben cruzarse entre sí.

Como se muestra en la Figura AA-1, la instancia de la clase **TerrenoContainer** corrobora que su atributo *terreno* sea vacío. En caso de no serlo disparará una excepción instancia de la clase **TerrenoYaDefinidoException** y se notifica el error en pantalla.

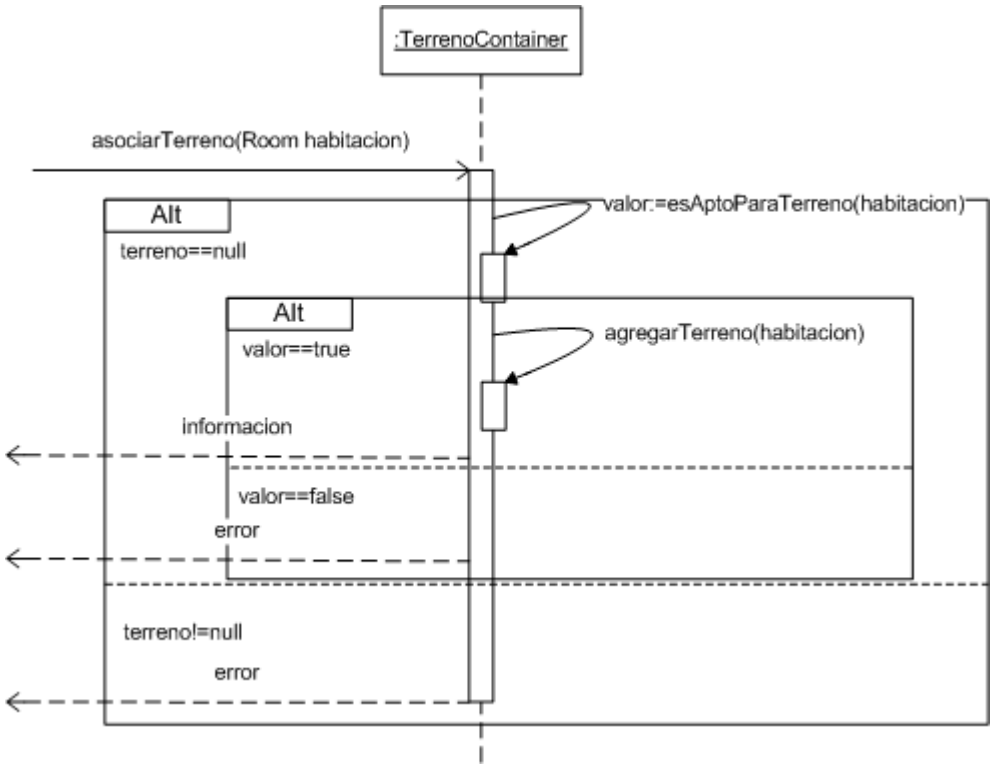


Figura AA-1: diagrama de secuencia del método asociarTerreno() de la clase TerrenoContainer

El método *esAptoParaTerreno(Room habitacion)* se encarga de corroborar que la instancia de la clase **Room** pasada como parámetro identifique correctamente un terreno. Para esto se utilizan las clases **Linea** y **Punto** del paquete **containers.utiles** y se procede de la siguiente manera:

- Se crea un conjunto de líneas con todos los puntos que forman la instancia de la clase **Room** seleccionada en el plano, de manera que para cada punto en la posición *i* de la instancia, se creará una línea con su punto inicial en el punto de la posición *i* y su punto final en el punto de la posición *i+1,* excepto en la última línea, la cual tendrá como punto final el punto de la posición 1. Esto se puede hacer debido a que la instancia de la clase **Room** ordena sus puntos de manera adecuada.
- Una vez creada las líneas, se recorre dos veces la colección verificando que ninguna de ellas se cruce con otra en puntos diferentes a los iniciales y finales.
- Si alguna de las líneas se cruzan de manera errónea, el terreno no es apto para dicho propósito y se dispara una excepción instancia de la clase **TerrenoMalFormadoException** y se notifica el error en pantalla. En caso contrario, la habitación cumple con el segundo requisito.

Habiendo verificado que la habitación es apta para representar un terreno, la instancia de la clase **TerrenoContainer** crea una instancia de clase **TerrenoE** con la habitación seleccionada en el plano y la almacena en su atributo *terreno* (Figura AA-2).

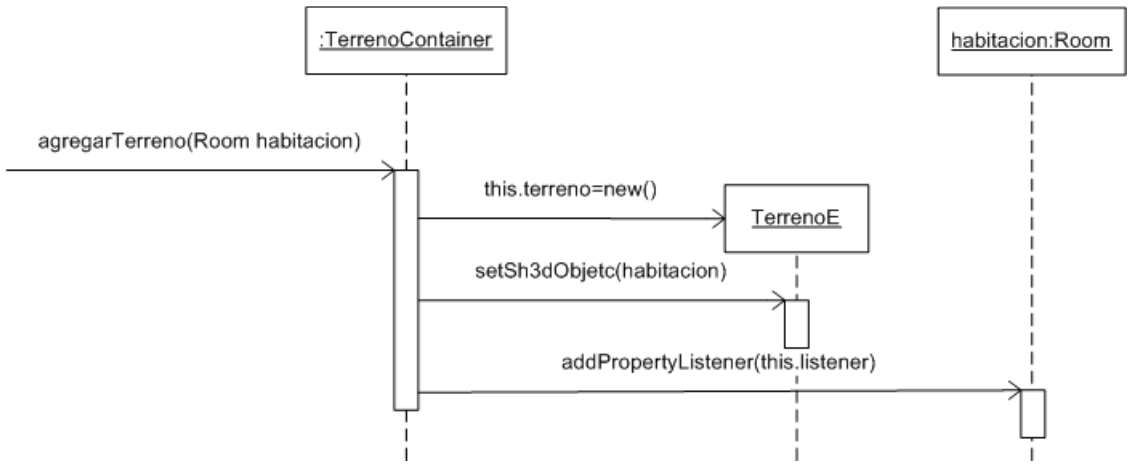


Figura AA-2: diagrama de secuencia del método agregarTerreno() de la clase TerrenoContainer

Explotación del método *asociarEspacio(List<Wall> paredes)* de la clase EspaciosContainer

El método *asociarEspacio(List<Wall> paredes)* (Figura AA-3) identifica como espacio al conjunto de instancias de la clase **Wall** pasadas como parámetro, siempre y cuando cumplan con los siguientes requisitos:

1. Las paredes deben formar, a partir de sus coordenadas de inicio y de fin, un polígono cerrado.
2. El conjunto de paredes no pueden haber sido identificadas como espacio previamente
3. Debe existir un terreno identificado.
4. El área del espacio debe estar totalmente incluido dentro del área que abarca el terreno.
5. El área del espacio no debe superponerse con el área de ningún otro espacio, excepto cuando la superposición significa una inclusión total, caso en cual los espacios no podrán compartir pared alguna.

Como se muestra en la Figura AA-3, el método *asociarEspacio(ArrayList<Wall> paredes)* invoca al método *generarEspacio(List<Wall> paredesSinProcesar, List<Wall> paredesProcesadas, Punto puntoIntersección) de la misma instancia* , el cual recibe como parámetros inicialmente la lista de paredes seleccionadas en el plano (*paredesSinProcesar*), una lista vacía (*paredesProcesadas*) y un punto nulo (*puntoIntersección*). El objetivo de este método es verificar que el conjunto de paredes forma un polígono cerrado.

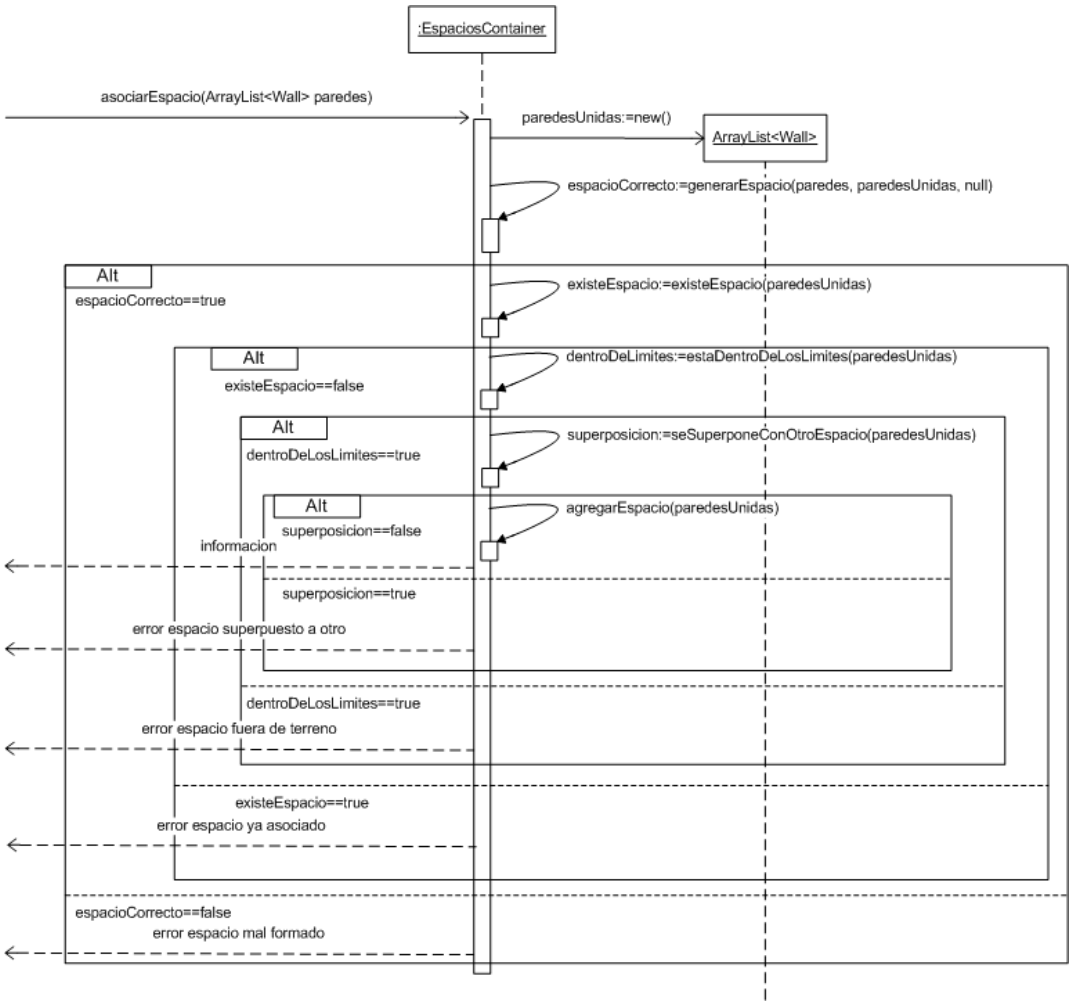


Figura AA-3: diagrama de secuencia del método asociarEspacio() de la clase EspaciosContainer

La idea del método es invocarse recursivamente de la siguiente manera, tal como se muestra en la Figura AA-4:

- Se invoca al método estático *ObtenerParedesUnion(List<Wall> paredesSinProcesar, Punto puntoIntersección)* de la clase **Utiles**. El objetivo del método es retornar el subconjunto de las paredes pasadas como parámetros que empiezan o terminan en el punto de intersección que se pasa también como parámetro. La primera invocación tendrá como punto de intersección un valor nulo, por lo que el método procede a retornar únicamente la primera pared de la colección.
- Si el método estático *obtenerParedesUnion(List<Wall> paredesSinProcesar, Punto puntoIntersección)* de la clase **Utiles** retorna una sola pared, se invoca al método estático *obtenerPuntoContrario(Wall pared, Punto punto)* de la misma clase con la pared elegida como primer parámetro y el punto de intersección como segundo. Dicho método retorna el punto de la pared (punto inicial o punto final) contrario al punto pasado como parámetro. En la primera invocación de la recursividad, al ser el punto pasado como parámetro nulo, se retorna el punto final de la pared.
- Se procede a eliminar del conjunto de paredes sin procesar la pared resultado, y se agrega la misma a la lista de paredes ya procesadas. Con estas dos listas y el punto obtenido en el método *obtenerPuntoContrario(Wall pared, Punto punto)*, se vuelve a invocar recursivamente al método *generarEspacio(List<Wall> paredesSinProcesar, List<Wall> paredesProcesadas, Punto puntoIntersección).*
- Así recursivamente hasta que el método estático *obtenerParedesUnion(List<Wall> paredesSinProcesar, Punto puntoIntersección)* de la clase **Utiles** retorne más o menos de una pared. Si el mismo retorna más de una pared, significa que hay dos o más paredes unidas a esa pared, por lo que el conjunto de paredes seleccionadas no cumplen el requisito 3. Si no se retorna ninguna pared, se corrobora que la lista de paredes sin procesar no tenga más elementos y que la ultima pared se una a la primera de la colección de paredes ya procesadas. Si esto ocurre, en la lista de paredes ya procesadas estarán las paredes ordenadas formando el espacio en cuestión, y el método retornará el valor Verdadero. Caso contrario, significa que hay al menos una pared que no tiene ninguna pared unida a alguno de sus extremos, lo que lleva a que el requisito de formar un polígono cerrado no se cumpla y el método retornará el valor False.

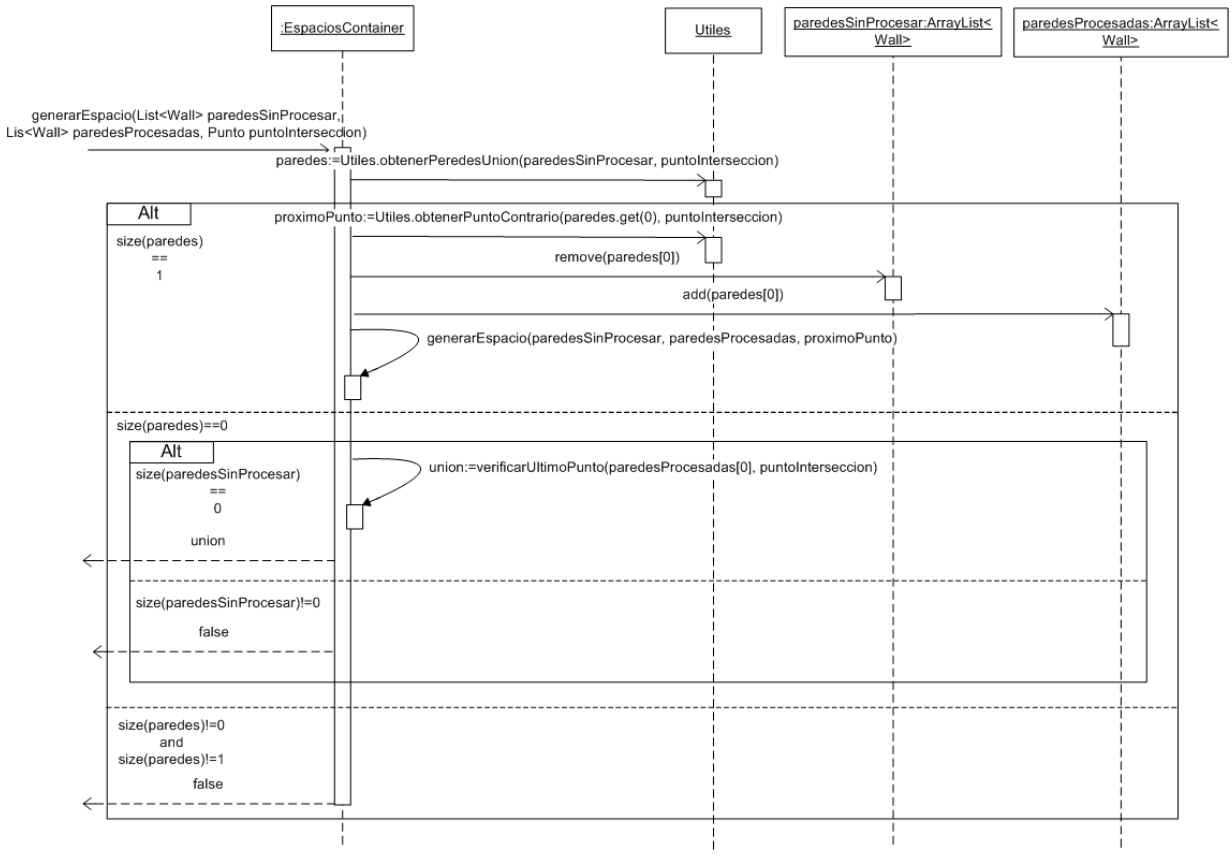


Figura AA-4: diagrama de secuencia del método generarEspacio() de la clase EspaciosContainer

Si el método *generarEspacio(List<Wall> paredesSinProcesar, List<Wall> paredesProcesadas, Punto puntoIntersección)* retorna False, se dispara una excepción instancia de la clase **EspacioMalFormadoException.** En caso contrario, el método ordenará las paredes, de manera tal que el inicio de una es el fin de la anterior en la colección, cerrándose el polígono entre la primera pared de la colección y la última.

En caso de que el espacio forme un polígono cerrado, el método *asociarEspacio(Room habitación),* verifica que la lista de espacios almacenados por la instancia de la clase **EspaciosContainer** no contenga un espacio formado por las mismas paredes que el conjunto de paredes seleccionadas en el plano. En caso de existir, se dispara una excepción instancia de la clase **EspacioAsociadoPreviamenteException.** En caso contrario se procede a verificar que todas las paredes estén dentro del terreno invocando al método *estaDentroDeLosLimites(List<Wall paredes)*, al cual se le pasa la lista de paredes ordenadas. El mismo verifica que el área que delimita las paredes este totalmente dentro del área del terreno, para lo cual debe existir primero un terreno identificado.

En caso de no cumplirse se dispara una excepción instancia de la clase **EspacioFueraDeTerrenoException.** En caso contrario, se procede a corroborar el último requisito invocando al método *seSuperponeConOtroEspacio(List<Wall> espacio)* de la misma clase **EspaciosContainer**, el cual verifica que el área que forman las paredes cumpla con los siguientes requisitos:

- Dos espacios solo pueden superponerse en caso de que el área de alguno de ellos esté totalmente incluida en el área del otro.
- Si se cumple lo anterior, los espacios en cuestión no deben compartir pared alguna.

Si esto no ocurre se dispara una excepción instancia de la clase **EspacioSuperpuestoException.** En caso contrario, el conjunto de paredes ordenadas forman un espacio apto, por lo que se procede a agregarlo al conjunto de espacios identificados, tal como se muestra en la Figura AA-5. Para esto la instancia de la clase **EspaciosContainer** crea una instancia de clase **EspacioE** con la lista de paredes ordenadas. El espacio creado se guarda en el atributo Espacios. En caso de existir un espacio previamente identificado que contenga al nuevo espacio, se procede a buscar los tags incluidos dentro del espacio padre antes de la nueva identificación que hayan quedado incluidos ahora dentro del nuevo espacio y se les cambia la referencia del espacio al que pertenecen.

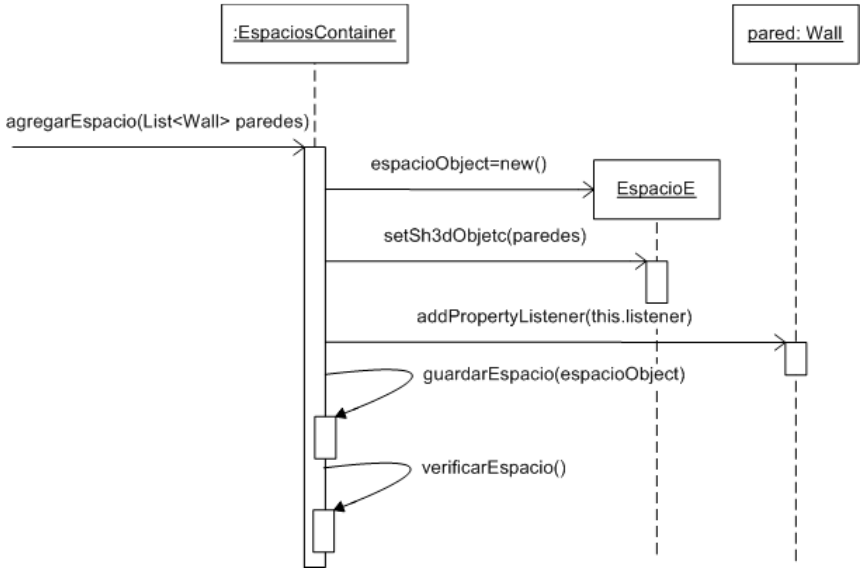


Figura AA-5: diagrama de secuencia del método agregarEspacio() de la clase EspaciosContainer

Explotación del método *desasociarEspacio(List<Wall> paredes)* de la clase EspaciosContainer

El método des*asociarEspacio(List<Wall> paredes)* (Figura AA-6) elimina de la lista de espacios identificados aquel espacio formado únicamente por conjunto de instancias de la clase **Wall** pasadas como parámetro, siempre y cuando exista el mismo.

El método en si verifica que exista en la lista de espacios identificados previamente una instancia de la clase **EspacioE** que esté formada únicamente por las paredes recibidas como parámetro en la invocación del método.

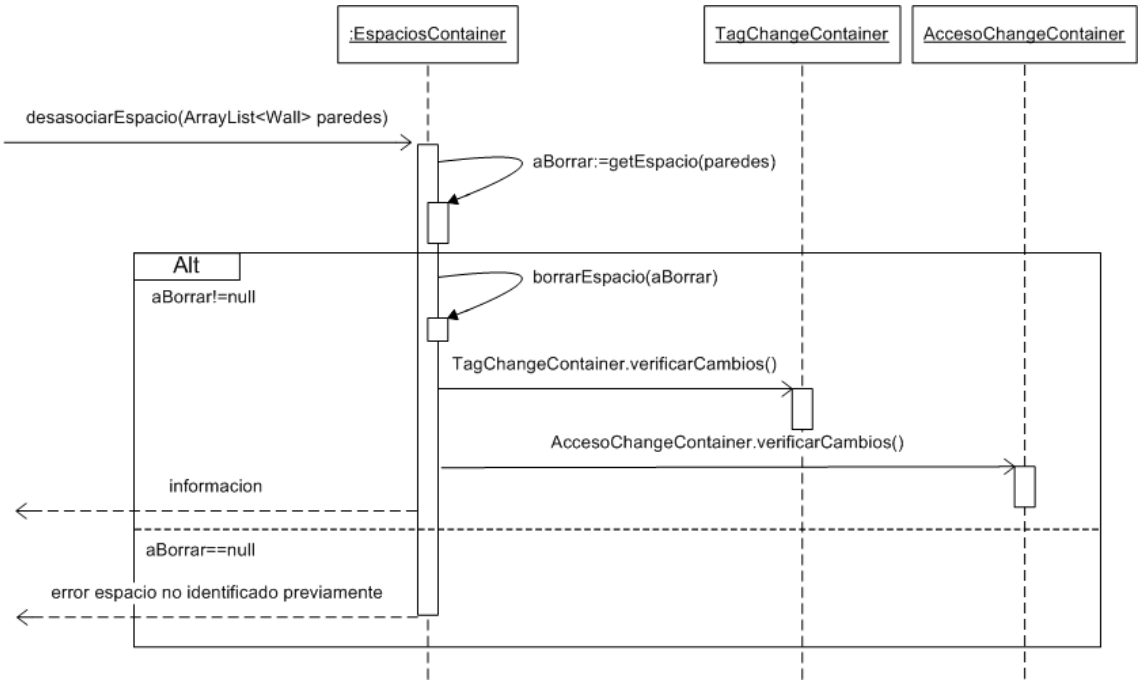


Figura AA-6: diagrama de secuencia del método desasociarEspacio() de la clase EspaciosContainer

Para esto, la instancia de la clase **EspaciosContainer** recorre la lista de espacios identificados buscando una instancia con dicho requisito. En caso de no existir, se dispara una excepción instancia de la clase **EspacioNoAsociadoException**. En caso contrario, se procede a eliminar el objeto de la lista de espacios identificados, invocando al método *borrarEspacio(EspacioE espacio)* (Figura AA-7), el cual procede de la siguiente manera:

- Se recuperan los objetos identificados como tags que pertenezcan al espacio.
- Se notifica a la clase que controla los cambios de los tags que los objetos recuperados en el punto anterior deben verificar su posición mediante la invocación al método estático *cambio(HomePieceOfFurniture tag)* de la clase **TagChangeContainer**.
- Se recuperan las paredes del espacio que solo pertenezcan al mismo, y no a otros.
- Se recuperan los objetos identificados como accesos que estén relacionados a alguna de las paredes obtenidas en el punto anterior.
- notifica a la clase que controla los cambios de los accesos que los objetos recuperados en el punto anterior deben verificar su posición mediante la invocación al método estático *cambio(HomeDoorOrWindow acceso)* de la clase **AccesoChangeContainer**.
- Se recupera la instancia de la clase **EspacioListener** que fue agregada a cada pared del mismo, y se procede a eliminar al mismo de su lista de listeners.
- Por último, se elimina el espacio de la lista.

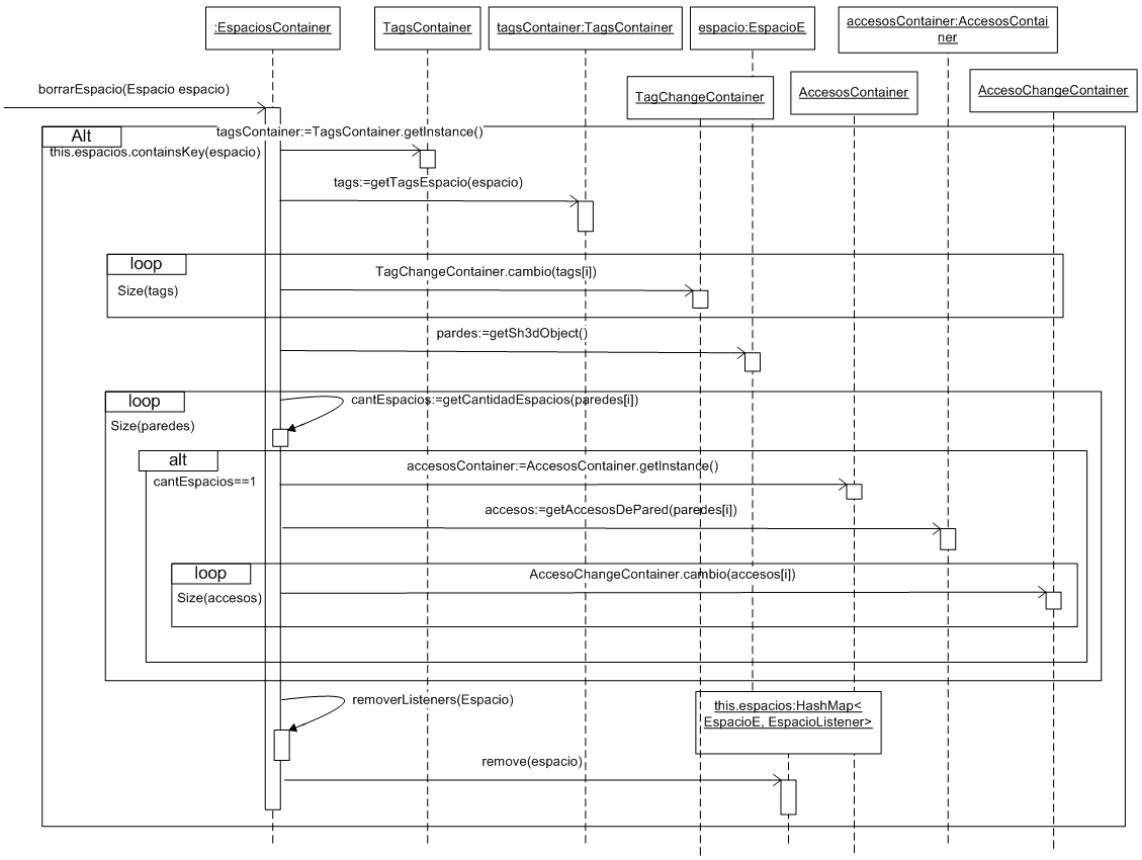


Figura AA-7: diagrama de secuencia del método borrarEspacio() de la clase EspaciosContainer

Una vez finalizado el método *borrarEspacio(Espacio espacio),* se retorna el control al método des*asociarEspacio(ArrayList<Wall> paredes),* al cual solo le quedará invocar al método *verificarCambios()* de las clases **TagChangeContainer** y **AccesoChangeContainers,** quienes se encargarán de verificar que los tags y los accesos ya identificados aún cumplen con sus respectivos requisitos. De ser necesario, actualizará las relaciones de los mismos. En el caso de los tag, pasarán a estar incluidos en el espacio padre, en caso de existir, del objeto eliminado. En caso contrario se elimina el tag. Los accesos serán borrados si y solo si, la pared en la que están puestos no pertenece a ningún espacio identificado luego del borrado.

Explotación del método *asociarAcceso(Object acceso)* de la clase AccesosContainer

El método *asociarAcceso(Object acceso)* (Figura AA-8) identifica como acceso al objeto pasado como parámetro, siempre y cuando cumpla con los siguientes requisitos:

1. El parámetro debe ser instancia de la clase **HomeDoorOrWindow.**
2. El parámetro no debe haber sido identificado como acceso previamente.
3. El parámetro debe estar posado sobre una pared perteneciente a un espacio identificado.

El método *asociarAcceso(Object acceso)* de la instancia de la clase **AccesosContainer** primero verifica que el parámetro sea instancia de la clase **HomeDoorOrWindow.** En caso de no serlo se informa el error y finaliza la ejecución del método. En caso contrario se procede a invocar al método *existeAsociacion(HomeDoorOrWindow acceso).*

Este método se encarga de verificar que el objeto pasado como parámetro no haya sido identificado previamente como acceso. Para esto recorre la lista de accesos identificados previamente (los cuales son guardados en el atributo *accesos* de la instancia de la clase **AccesosContainer**) y verifica que ninguno mantenga como referencia al objeto pasado como parámetro. En caso de que el objeto aparezca en la lista, el método retorna VERDADERO, en caso contrario FALSE.

En caso de que el objeto haya sido asociado previamente como acceso se dispara una excepción instancia de la clase **AccesoPreviamenteAsociadoException,** se informa del error y finaliza la ejecución del método.

En caso de que el objeto no haya sido identificado previamente como acceso se invoca al método *getParedPara(HomeDoorOrWindow acceso)* de la instancia de la clase **EspaciosContainer.** Este método recorre el conjunto de paredes que pertenecen a al menos un espacio previamente identificado buscando sobre cuál de ellas el acceso está posado. Para corroborar esto se verifica que el atributo *boundToWall* de la instancia de la clase **HomeDoorOrWindow** sea VERDADERO y que la forma del acceso permita identificar que el objeto está posado sobre alguna de las paredes (mediante el método estático *recuperarIntersecciones(Wall pared, HomePieceOfFurniture pieza)* de la clase **Utiles**).

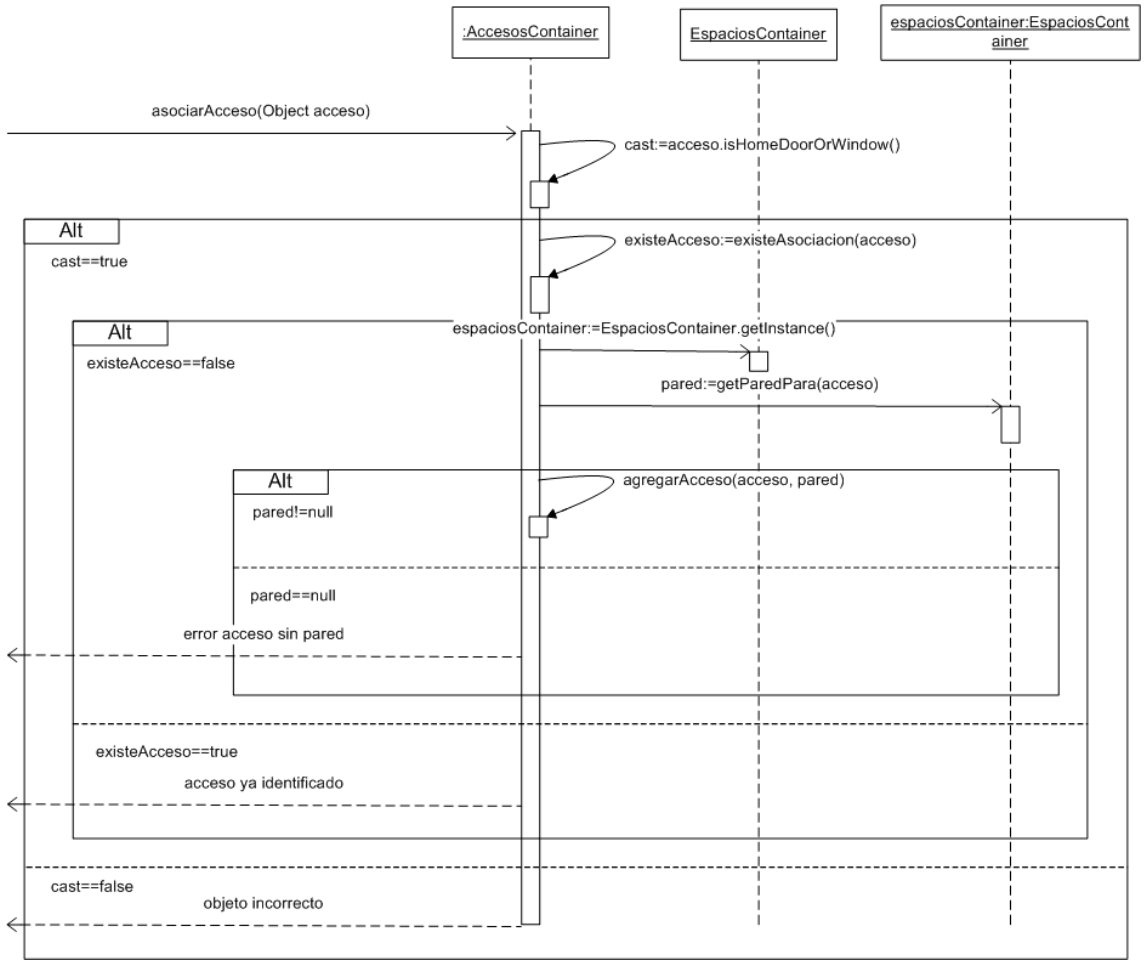


Figura AA-8: diagrama de secuencia del método asociarAcceso() de la clase AccesosContainer

El método *getParedPara(HomeDoorOrWindow acceso)* de la instancia de la clase **EspaciosContainer** retorna, en caso de existir, la pared a la que el acceso pertenece, y nulo en caso de que no esté asociado a ninguna pared. En caso de retornar este ultimo valor, se dispara una excepción instancia de la clase **AccesoSinParedException,** se informa del error y finaliza la ejecución del método. En caso de que si exista una pared para el acceso se procede a guardar el acceso, mediante el método *agregarAcceso(HomeDoorOrWindow acceso, Wall pared)* (Figura AA-9), por lo que se procede a agregarlo al conjunto de accesos previamente identificados. Para esto la instancia de la clase **AccesosContainer** crea una instancia de clase **AccesoE** la cual tendrá una referencia a la instancia de la clase **HomeDoorOrWindow** y otra a la instancia de la clase **Wall** sobre la que el objeto esta posado.

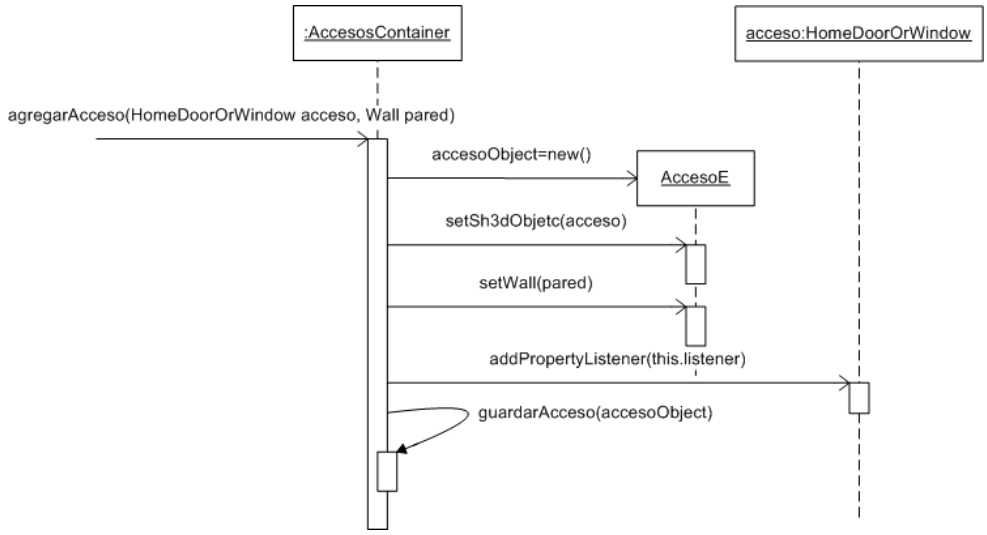


Figura AA-9: diagrama de secuencia del método agregarAcceso() de la clase AccesosContainer

Explotación del método *desasociarAcceso(Object acceso)* de la clase AccesosContainer

El método des*asociarAcceso(Object acceso)* (Figura AA-10) elimina de la lista de accesos identificados a aquel que haga referencia al objeto pasado como parámetro, siempre y cuando haya sido previamente identificado como tal.

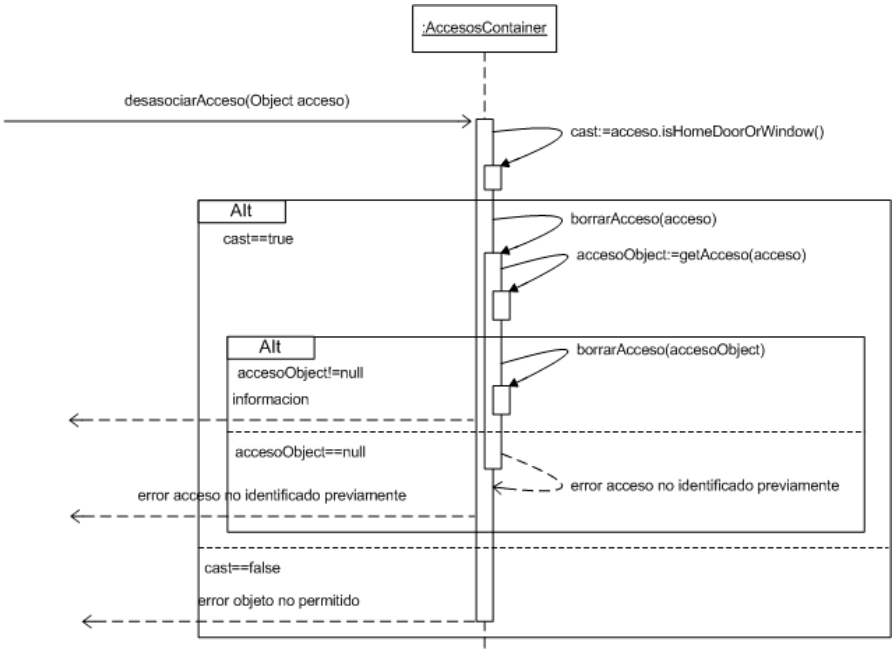


Figura AA-10: diagrama de secuencia del método desasociarAcceso() de la clase AccesosContainer

El método des*asociarAcceso(Object acceso)* de la instancia de la clase **AccesosContainer** comienza verificando que el objeto pasado como parámetro sea instancia de la clase **HomeDoorOrWindow.** En caso de no serlo se informa del error y finaliza la ejecución del método. En caso contrario se invoca al método *borrarAcceso(HomeDoorOrWindow acceso)* el cual se encarga de verificar que el objeto pasado como parámetro haya sido identificado previamente como acceso. Para esto recorre la lista de accesos identificados previamente (los cuales son guardados en el atributo *accesos* de la instancia de la clase **AccesosContainer**) y verifica que exista la instancia de la clase **AccesoE** que referencie la instancia de la clase **HomeDoorOrWindow** a borrar. En caso de que el objeto aparezca en la lista, se procede a borrarlo de la lista de accesos identificados. En caso contrario, se dispara una excepción instancia de la clase **AccesosNoAsociadoException,** se informa del error y finaliza la ejecución del método.

Explotación del método *asociarTag(Object tag)* de la clase TagsContainer

El método *asociarTag(Object tag)* (Figura AA-11) identifica como tag al objeto pasado como parámetro, siempre y cuando cumpla con los siguientes requisitos:

1. El objeto pasado como parámetro debe ser instancia de la clase **HomePieceOfFurniture.**
2. El objeto pasado como parámetro no puede ser instancia de la clase **HomeDoorOrWindow.**
3. El objeto pasado como parámetro no debe haber sido identificado como tags previamente.
4. El área que ocupa el objeto pasado como parámetro debe estar incluida en el área de un espacio previamente identificado.

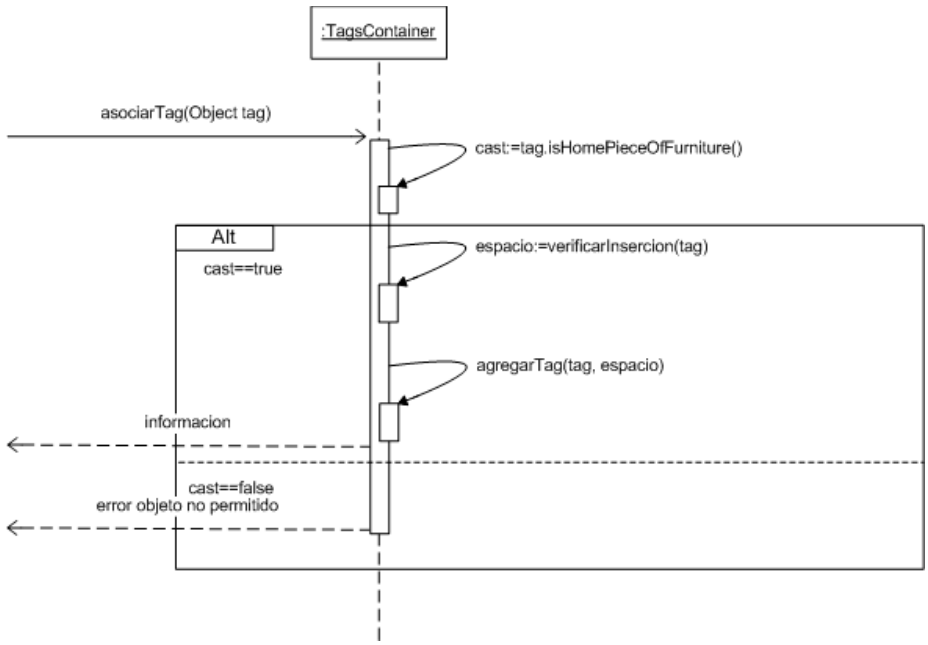


Figura AA-11: diagrama de secuencia del método asociarTag() de la clase TagsContainer

El método *asociarTag(Object tag)* de la instancia de la clase **TagsContainer** primero verifica que el objeto pasado como parámetro sea instancia de la clase **HomePieceOfFurniture.** En caso de no serlo, se informa del error y finaliza la ejecución del método. En caso contrario se procede a invocar al método *verificarInserción(HomePieceOfFurniture tag) (*Figura AA-12). El método *verificarInserción(HomePieceOfFurniture tag)* se encarga de verificar que el objeto pasado como parámetro cumpla los requisitos 2, 3 y 4.

Para verificar que el tag no haya sido identificado previamente, invoca al método *existeAsociacion(HomePieceOfFurniture tag),* el cual recorre la lista de tags identificados previamente (los cuales son guardados en el atributo *tags* de la instancia de la clase **TagsContainer**) y verifica que ninguno haga referencia al objeto pasado como parámetro. En caso de que el objeto haya sido asociado previamente como tag, el método retornará verdadero, y en caso contrario falso.

Una vez que el método *verificarInserción(HomePieceOfFurniture tag)* retoma el control de la ejecución, verifica el valor resultado. Cuando el resultado es verdadero se dispara una excepción instancia de la clase **TagPreviamenteAsociadoException,** se informa del error y finaliza la ejecución del método. En caso contrario continua con la ejecución del método verificando que el objeto no sea instancia de la clase **HomeDoorOrWindow.** Para esto, invoca al método *isDoorOrWindow()* sobre el mismo*,* el cual retorna verdadero en caso positivo y falso en caso contrario. Una vez que el método *verificarInserción(HomePieceOfFurniture tag)* retoma el control de la ejecución, verifica el valor resultado.

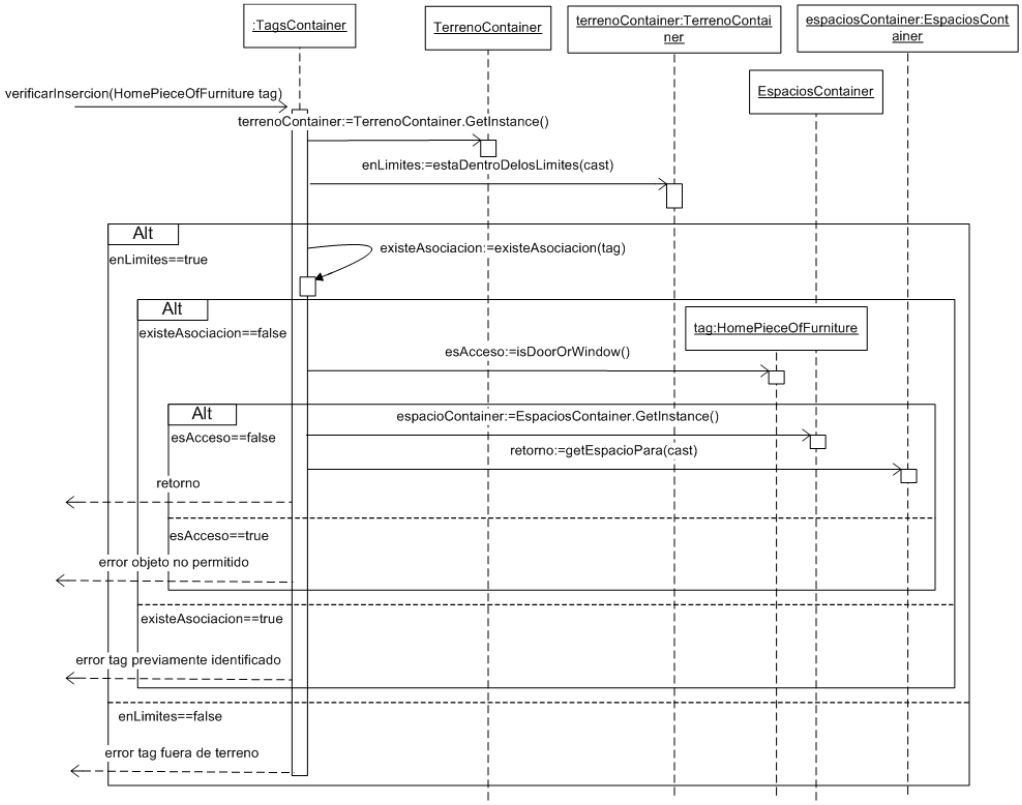


Figura AA-12: diagrama de secuencia del método verificarInserción() de la clase TagsContainer

En caso de ser verdadero, se dispara una excepción instancia de la clase **TagNoPermitidoException,** se informa el error y finaliza la ejecución del método. En caso contrario continua con la ejecución del método verificando que el tag se encuentre incluido dentro del área de un espacio previamente identificado. Para esto se invoca al método *getEspacioPara(HomePieceOfFurniture pieza)* de la instancia de la clase **EspaciosContainer**. Este método retornará el espacio de menor área previamente identificado en el que el tag esté totalmente incluido. En caso de que el tag se encuentre superpuesto sobre una pared, el método dispara una excepción instancia de la clase **TagSobreParedException,** se informa el error y finaliza la ejecución del método. Lo mismo pasa en caso de que el tag no se encuentre incluido en ningún espacio, excepto que la excepción que se dispara es instancia de la clase **TagSinEspacioExcepcion.**

En caso de existir un espacio que contenga al tag, se procede a agregarlo al conjunto de tags previamente identificados. Para esto la instancia de la clase **TagsContainer** crea una instancia de clase **TagE** referenciando a la instancia de la clase **HomePiecceOfFurniture** y a la instancia de la clase **EspacioE** que identifica al espacio que lo incluye.

Explotación del método *desasociarTag(Object tag)* de la clase TagsContainer

El método des*asociarAcceso(Object acceso)* (Figura AA-13) elimina de la lista de tags identificados a aquel que haga referencia al objeto pasado como parámetro, siempre y cuando haya sido previamente identificado como tal.

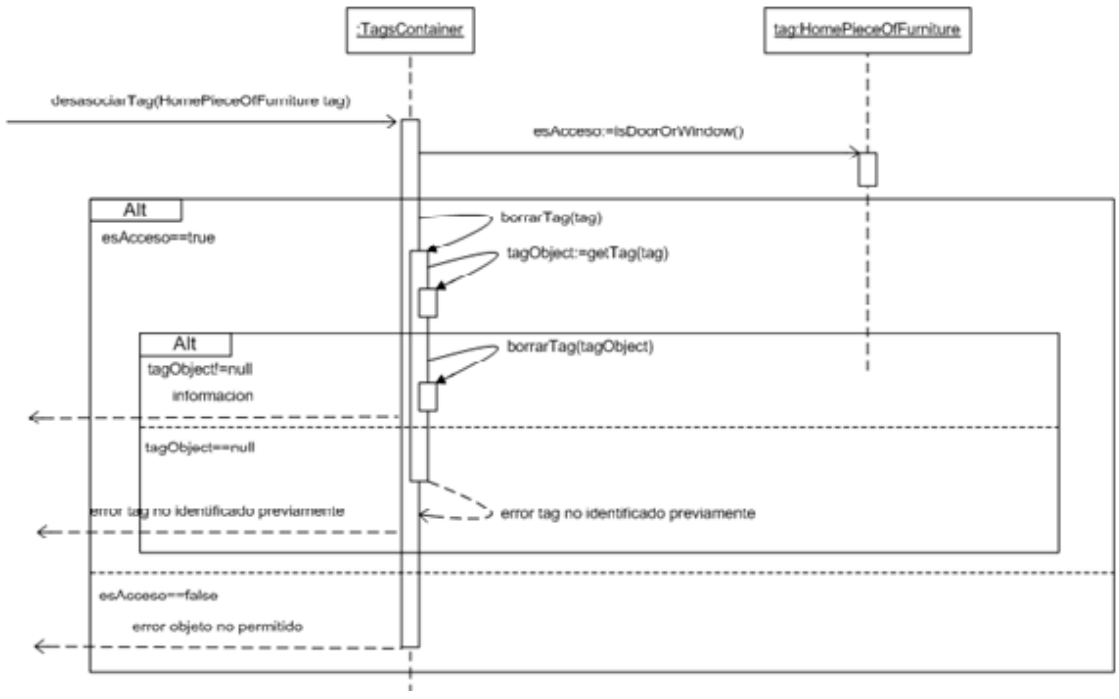


Figura AA-13: diagrama de secuencia del método desasociarTag() de la clase TagsContainer

El método des*asociarTag(Object tag)* de la instancia de la clase **TagsContainer** primero verifica que el objeto pasado como parámetro sea instancia de la clase **HomePieceOfFurniture.** En caso negativo, se informa del error y finaliza la ejecución del método.

En caso contrario se procede a invocar al método *borrarTag(HomePieceOfFurniture tag)*, el cual se encarga de verificar que el objeto pasado como parámetro haya sido identificado previamente como tag. Para esto recorre la lista de tags identificados previamente y verifica que exista la instancia de la clase **TagE** que referencie al objeto pasado como parámetro mediante el atributo sh3dObject. En caso positivo se borra la instancia de la clase **TagE** de la lista de tags identificados. En caso contrario, se dispara una excepción instancia de la clase **TagNoAsociadoException** y finaliza la ejecución del método.

Explotación del método *asociarObstaculosAutomaticamente()* de la clase ObstaculosContainer

Como se explica en el capítulo 3, debido a que los obstáculos no poseen ninguna característica especial en nuestro modelo final, se identifican automáticamente al momento de la exportación. Para que un elemento sea identificado automáticamente como obstáculo, debe cumplir los siguientes requisitos:

1. ser instancia de la clase **HomePieceOfFurniture.**
2. no pueden ser instancia de la clase **HomeDoorOrWindow.**
3. no deben haber sido identificados como tags previamente.
4. deben estar incluidos en el área de un espacio previamente identificado.

Para lograr esto, la exportación utiliza el método *asociarObstaculosAutomaticamente() (*Figura AA-14) de la instancia de la clase **ObstaculosContainer.** El método retorna VERDADERO en caso de que todos los elementos que cumplen los requisitos 1, 2 y 3 cumplen el requisito 4, y FALSO en caso contrario.

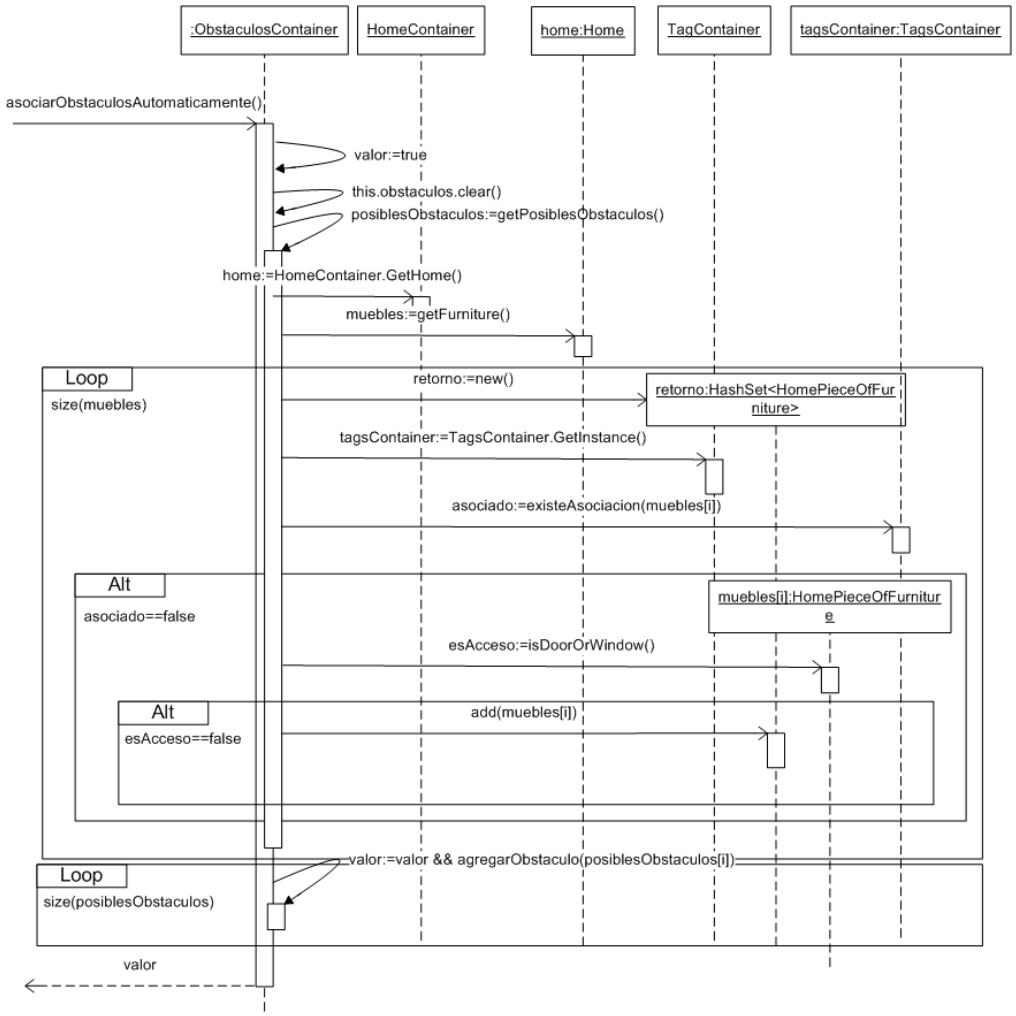


Figura AA-14: diagrama de secuencia del método asociarObstaculosAutomaticamente() de la clase ObstaculosContainer

Como se aprecia en la Figura AA-15, lo primero que hará el método será vaciar los elementos guardados en el atributo *obstáculos de la instancia.* Luego busca elementos en el plano que cumplan con los primeros tres requisitos. Para esto recupera de la instancia de la clase **Home** todos los elementos del plano que no hayan sido identificado previamente como tag (pasando al método *existeAsociacion(HomePieceOfFurniture piece)* de la instancia de la clase **TagsContainer** uno por uno los objetos y guardando los que retornan el valor FALSE) y que no sean instancia de la clase **HomeDoorOrWindow** (mediante el método *isDoorOrWindow()* aplicado sobre el objeto).

Una vez recuperados los objetos que cumplen los primeros tres requisitos, por cada uno de ellos se invoca al método *agregarObstaculo(HomePieceOfFurniture piece)* de la instancia de la clase **ObstaculosContainer** (la explotación y explicación de este método se puedo ver en el ANEXO PROTOTIPO) el cual determinará si el elemento se identifica como obstáculo.

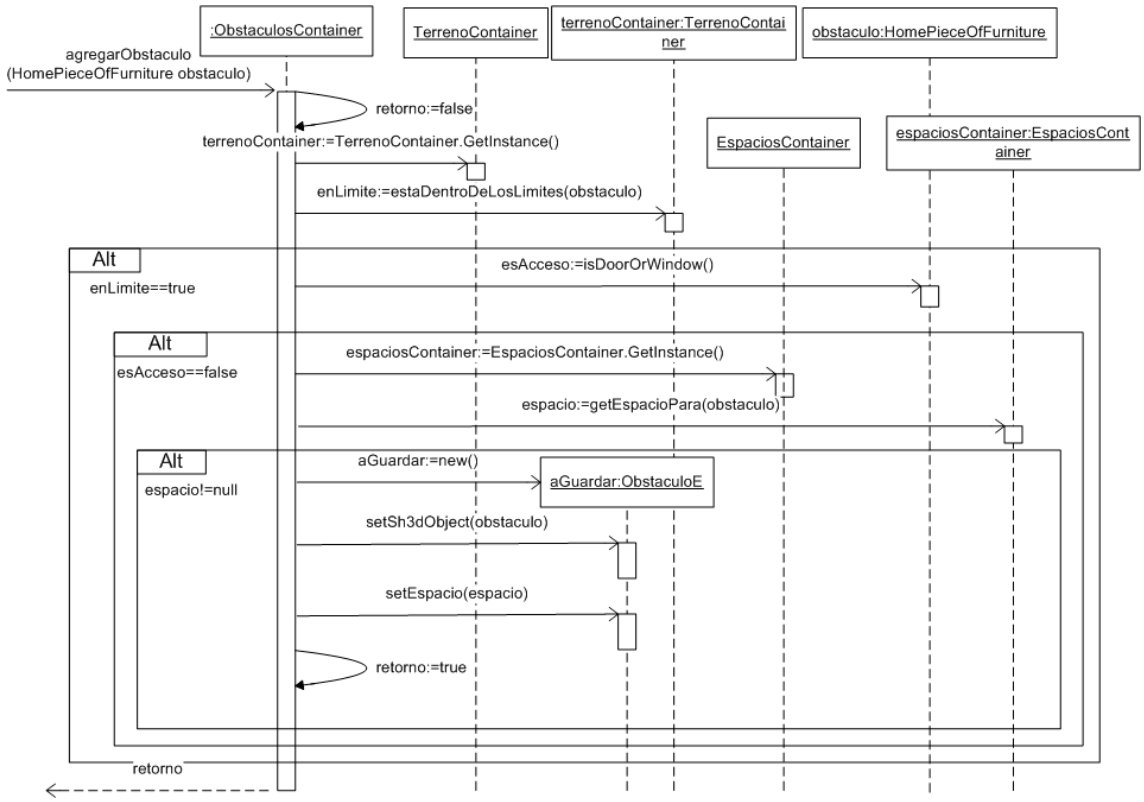


Figura AA-15: diagrama de secuencia del método agregarObstaculo(HomePieceOfFurniture obstaculo) de la clase ObstaculosContainer

Una vez recuperados los objetos que cumplen los primeros tres requisitos, por cada uno de ellos se invoca al método *agregarObstaculo(HomePieceOfFurniture piece)* de la instancia de la clase **ObstaculosContainer** (la explotación y explicación de este método se puedo ver en el ANEXO PROTOTIPO) el cual retorna VERDADERO y agrega una instancia de la clase **ObstaculoE** en el atributo *obstaculos* en caso de que exista un espacio previamente identificado que incluya totalmente al objeto (si hay más de un espacio que lo contenga, será el de menor área de entre todos los identificados, y el objeto lo almacenará en su atributo *espacio).*

Explotación del método estático *generarParedes()* de la clase Exporter

Para crear las instancias de la clase **Pared**, se procede a invocar al método estático *generarParedes(List<Wall> walls, HashMap<Wall, Pared> paredesExportadas),* tal como se muestra en las Figuras AA-16 y AA-17*,* el cual recibe como parámetro la lista de paredes que componen a al menos una instancia de la clase **EspacioE,** y devuelve un map relacionando la instancia de la clase **Wall** (modelo SH3D) y la instancia de la clase **Pared** (modelo Final) creada a partir de ella.

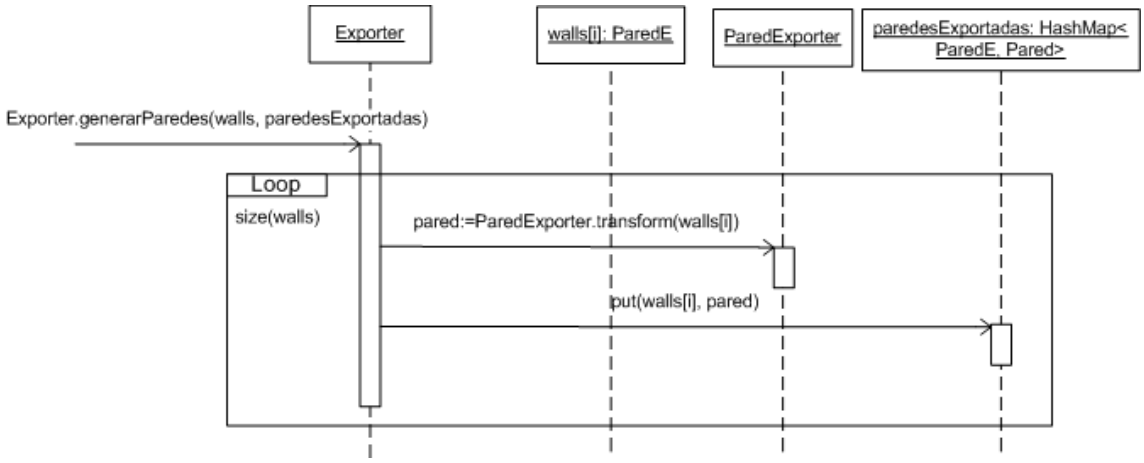


Figura AA-16: diagrama de secuencia del método estático generarParedes() de la clase Exporter

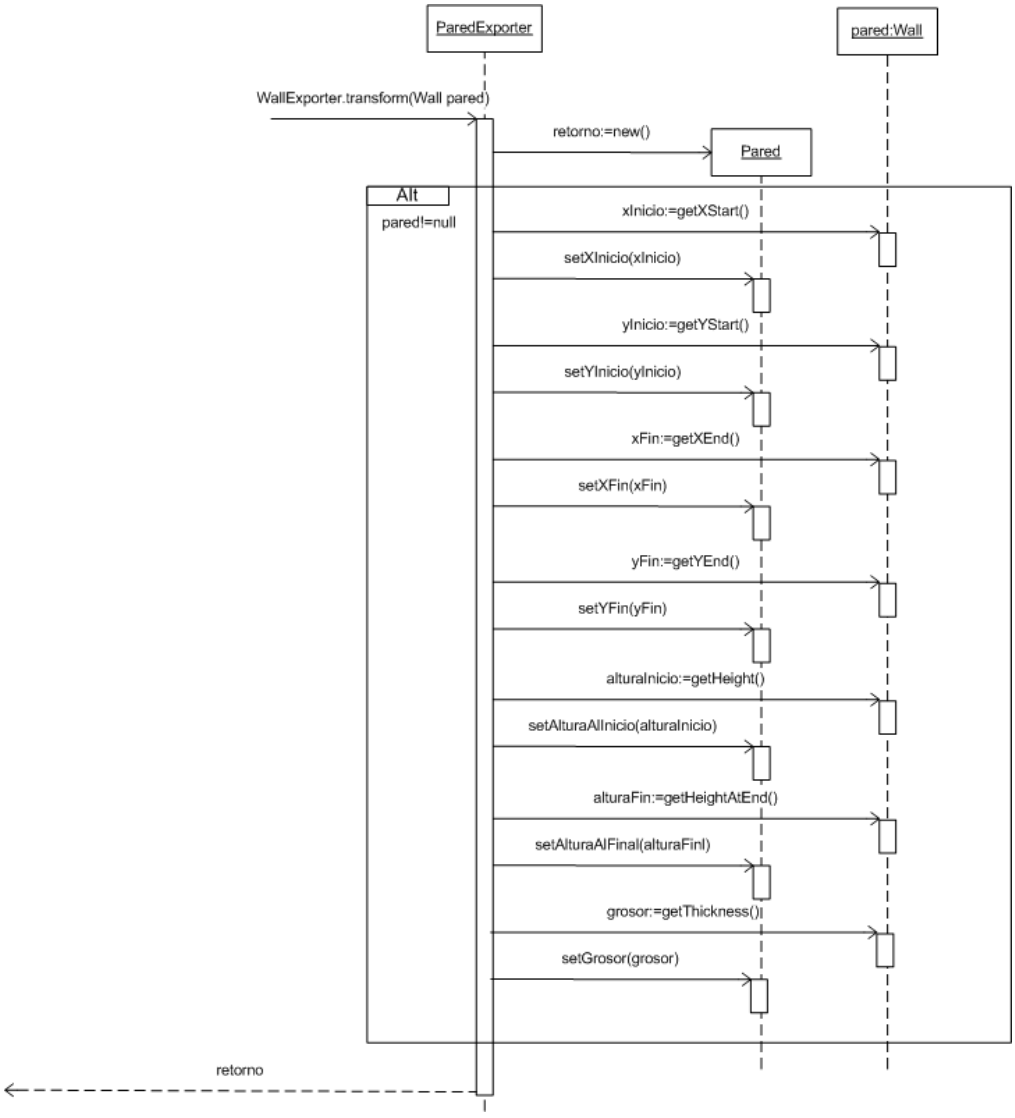


Figura AA-17: diagrama de secuencia del método estático transform() de la clase ParedExporter

Explotación del método estático *generarObstaculos()* de la clase Exporter

Para crear las instancias de la clase **Obstaculo**, se procede a invocar al método estático *generaObstaculos(List<ObstaculoE> obstaculos, HashMap<ObstaculoE, Obstaculo> obstaculosExportados),* tal como se muestra en las Figuras AA-18 y AA-19*,* el cual recibe como parámetro la lista de instancias de la clase **ObstaculoE,** y devuelve un map relacionando cada una de ellas con la respectiva instancia de la clase **Obstaculo** creada a partir de la misma.

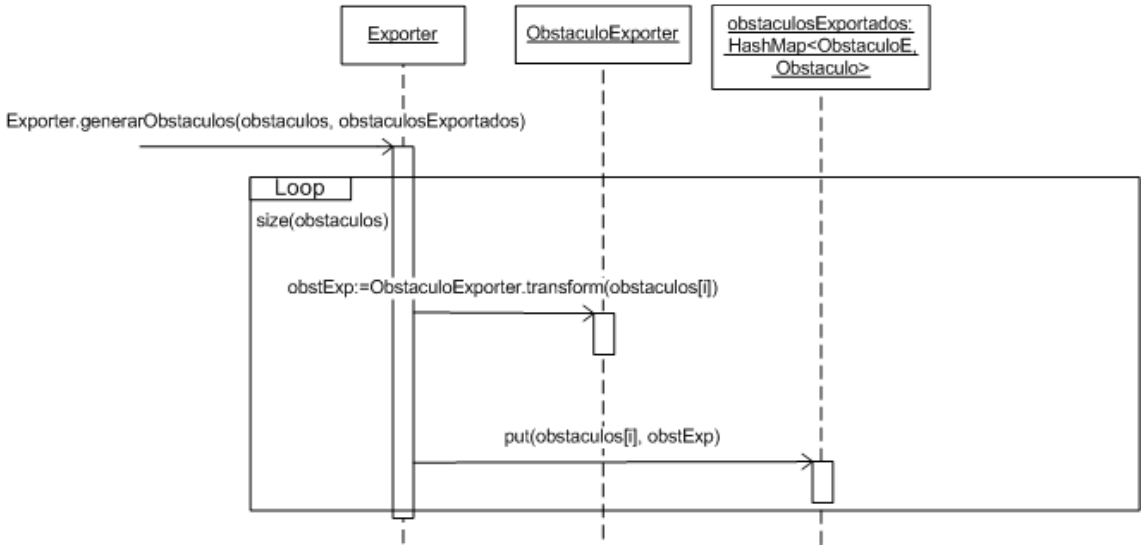


Figura AA-18: diagrama de secuencia del método estático generarObstaculos() de la clase Exporter

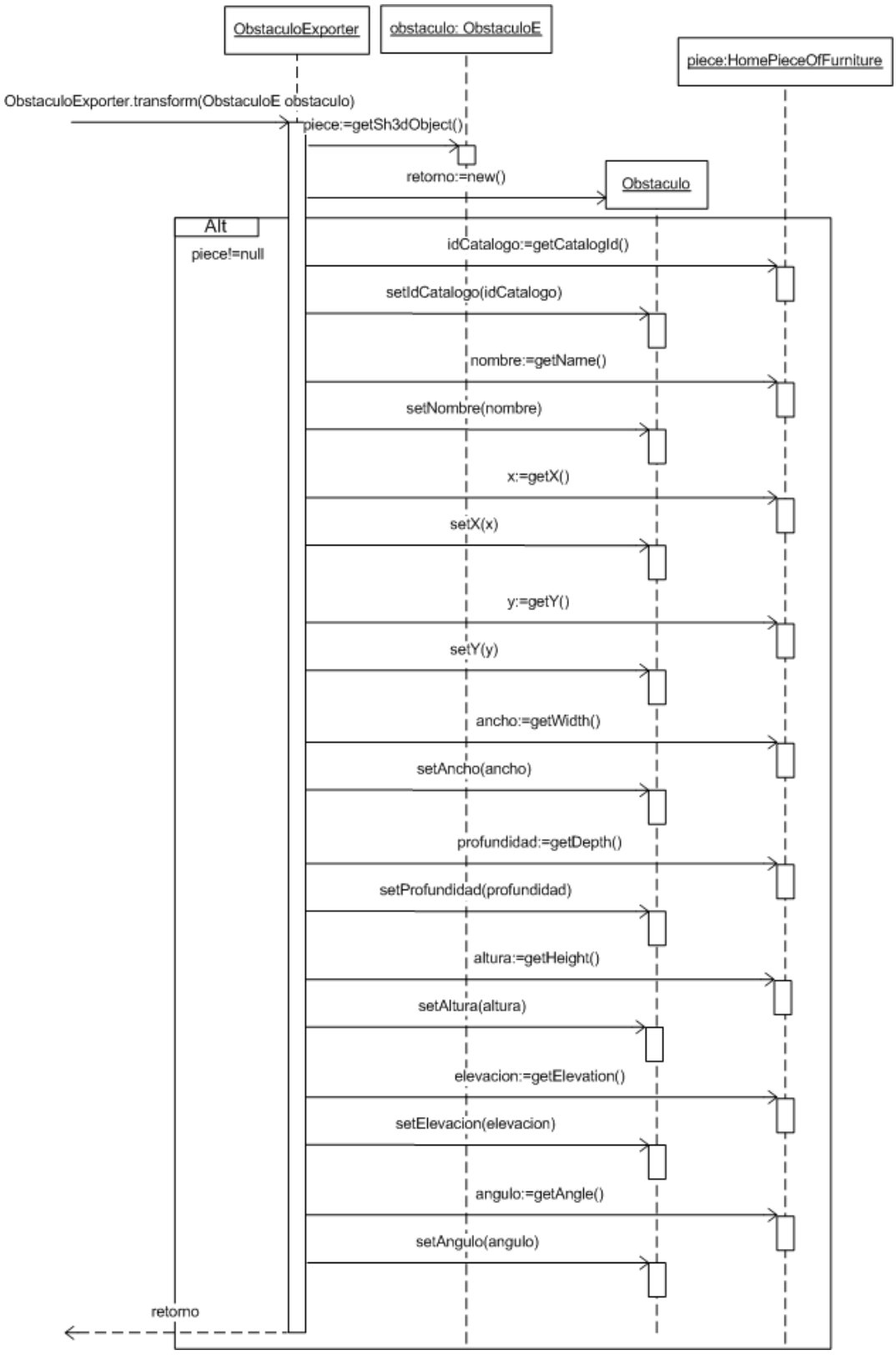


Figura AA-19: diagrama de secuencia del método estático transform() de la clase ObstaculoExporter

Explotación del método estático *generarTags()* de la clase Exporter

Para crear las instancias de la clase **Tag**, se procede a invocar al método estático *generaTags(List<TagE> tags, HashMap<TagE, Tag> tagsExportados),* tal como se muestra en las Figuras AA-20 y AA-21*,* el cual recibe como parámetro la lista de instancias de la clase **TagE,** y devuelve un map relacionando cada una de ellas con la respectiva instancia de la clase **Tag** creada a partir de la misma.

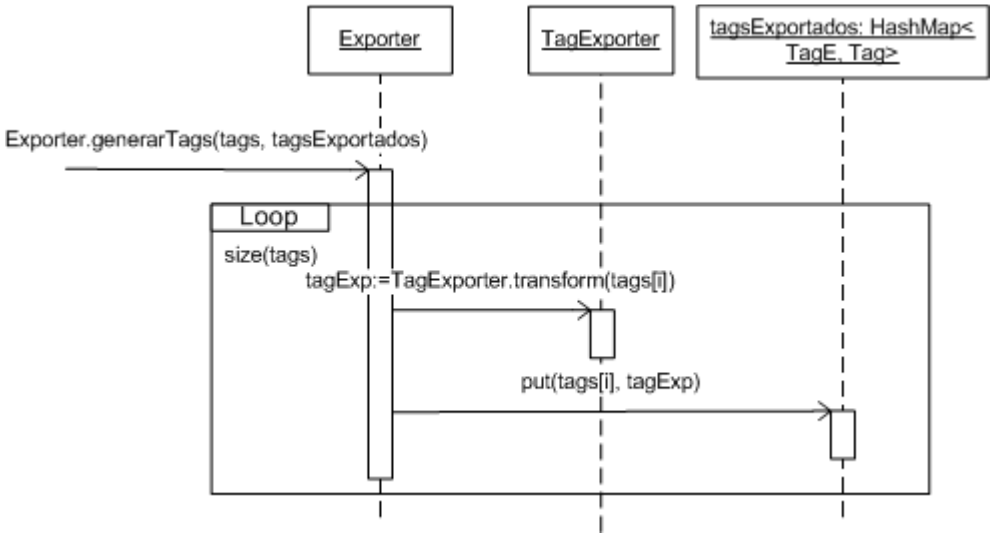


Figura AA-20: diagrama de secuencia del método estático generarTags() de la clase Exporter

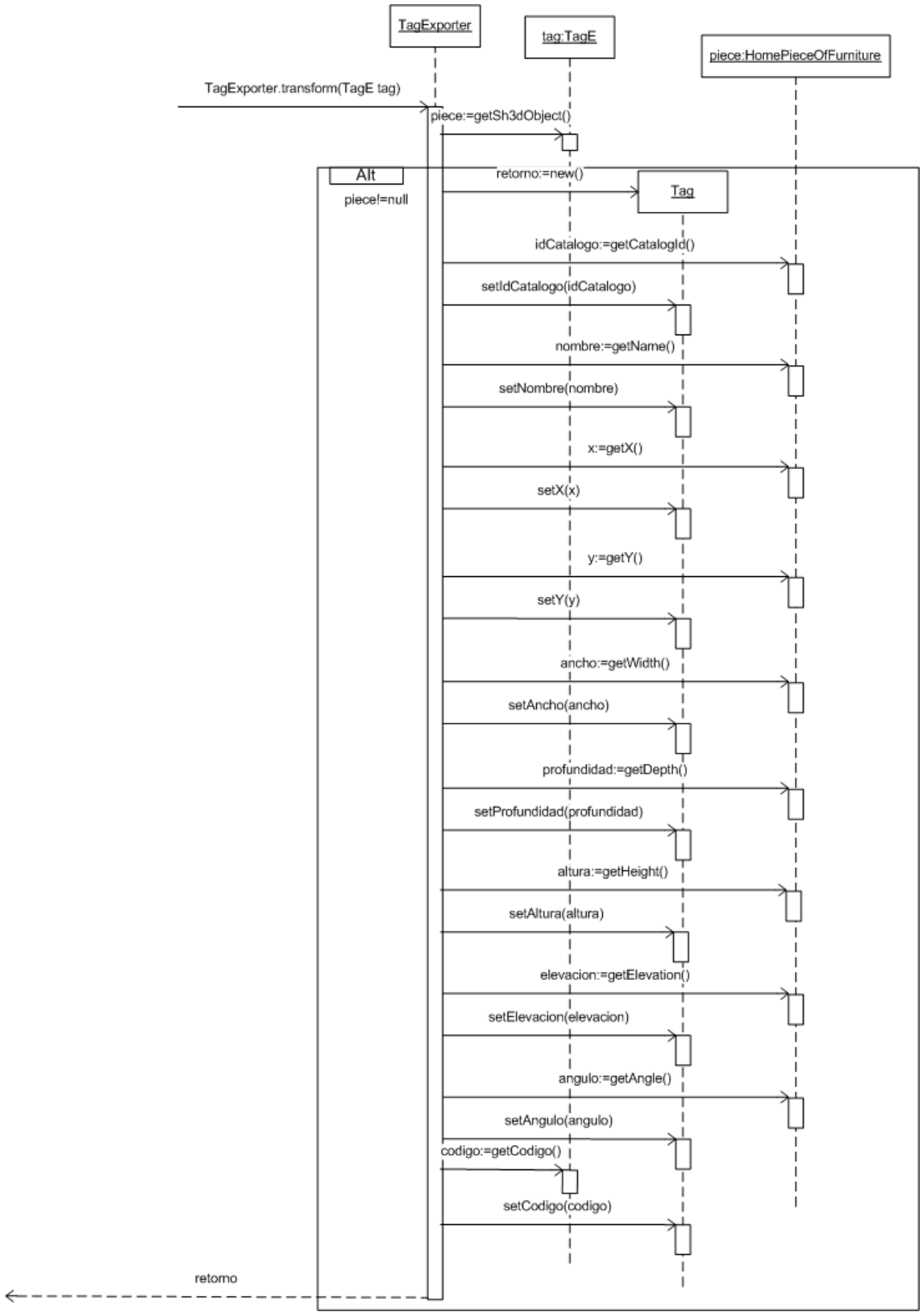


Figura AA-21: diagrama de secuencia del método estático transform() de la clase TagExporter

Explotación del método estático *generarAccesos()* de la clase Exporter

Para crear las instancias de la clase **Acceso**, se procede a invocar al método estático *generaAccesos(List<AccesoE> accesos, HashMap<AccesoE, Acceso> accesosExportados),* tal como se muestra en las Figuras AA-22 y AA-23*,* el cual recibe como parámetro la lista de instancias de la clase **AccesoE,** y devuelve un map relacionando cada una de ellas con la respectiva instancia de la clase **Acceso** creada a partir de la misma.

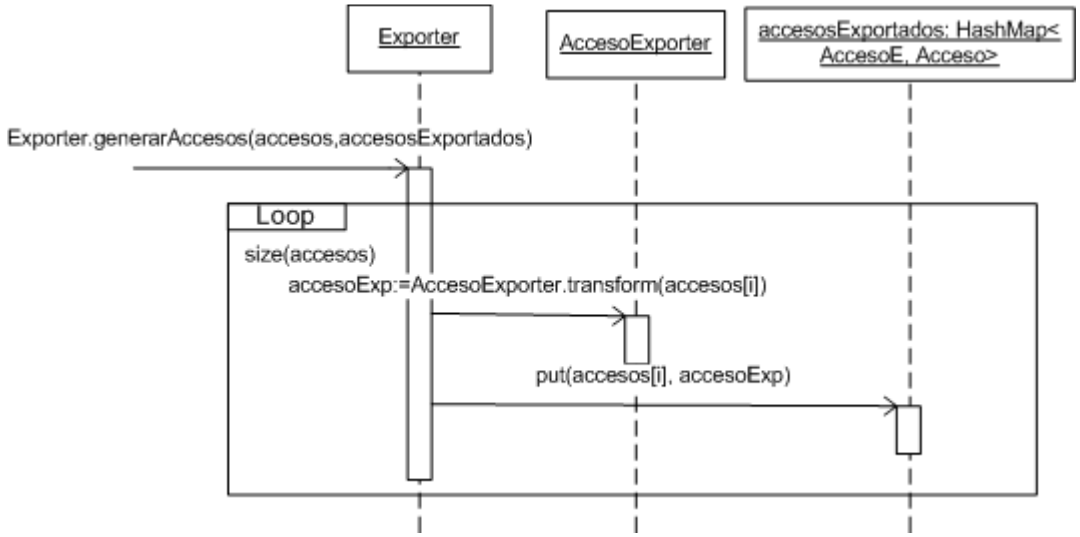


Figura AA-22: diagrama de secuencia del método estático generarAccesoss() de la clase Exporter

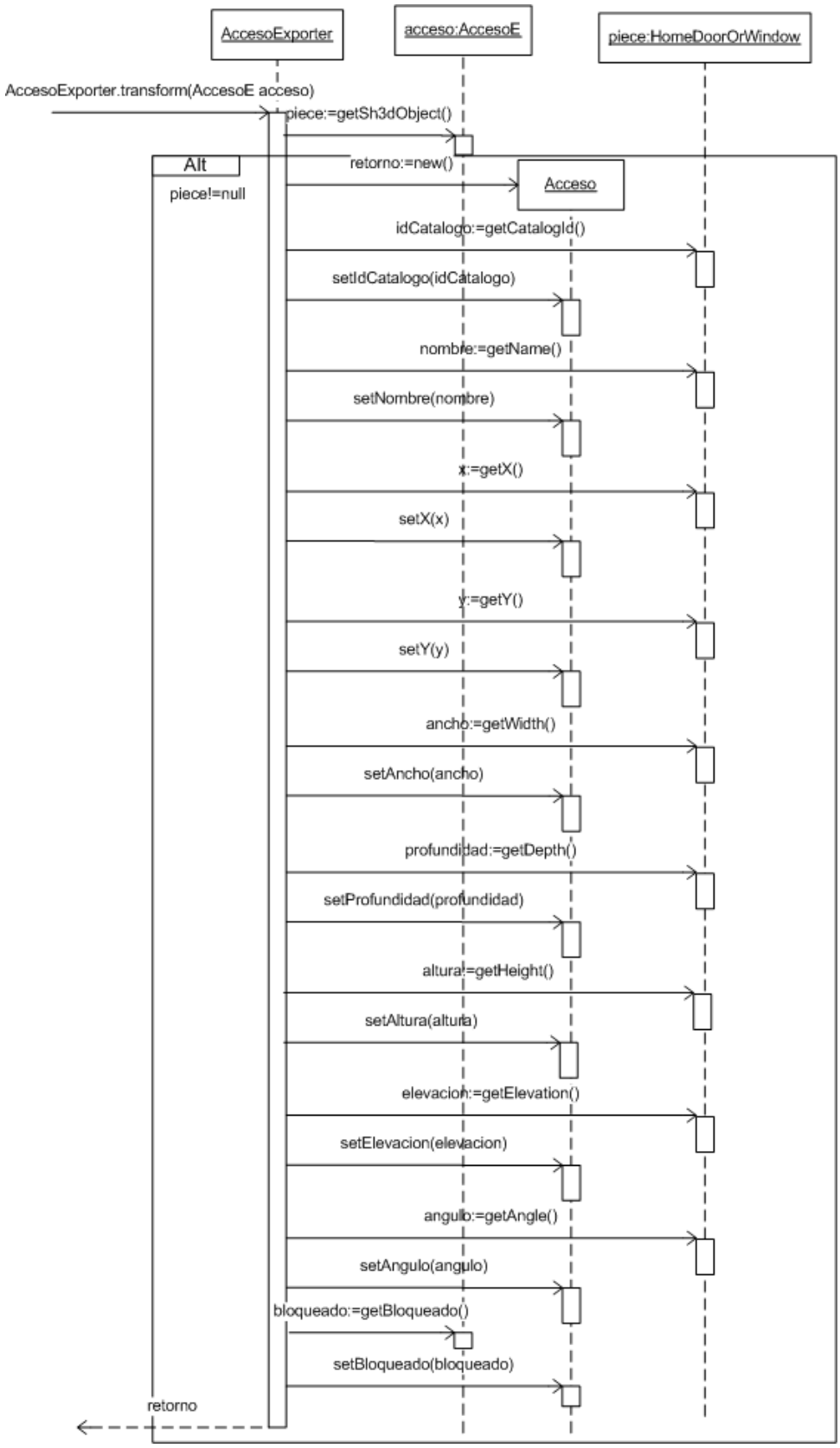


Figura AA-23: diagrama de secuencia del método estático transform() de la clase AccesoExporter

ANEXO A.2. ALGUNOS COMPORTAMIENTOS EXTRAS

Debido a que las instanciaciones de las clases de nuestro modelo extendido a partir de las instancias del modelo SH3D se hacen en cualquier momento, y la modificación del plano puede llevar a que las mismas dejen de cumplir con sus respectivos requisitos, se creó un mecanismo de almacenamiento de las instancias del modelo extendido (restringido a las posibilidades del software y su mecanismo de plugins), junto con un mecanismo de listeners que permite reconocer los cambios de las instancias del modelo SH3D para verificar que los requisitos del modelo extendido se cumplan.

Almacenamiento de las instancias del modelo extendido

Cada vez que se crea una instancia del modelo extendido se debe almacenar. Con este objetivo existe un conjunto de clases, que permiten recuperar a través de un método estático (*getInstance()*) una instancia de sí misma (patrón singleton), las cuales almacenarán las asociaciones que se irán realizando con el correr de la ejecución del software. Además, cada una de estas instancias, son las encargadas de verificar que las identificaciones requeridas cumplen con los requisitos necesarios para realizarse.

Para esto creamos el paquete **containers.** Cada una almacena un conjunto de identificaciones de la siguiente manera:

- La instancia de la clase **containers.TerrenoContainer** almacena la instancia de la clase **TerrenoE.** Esto se debe a que solo puede identificarse uno y solo un terreno.
- La instancia de la clase **containers.EspaciosContainer** almacena las instancias de la clase **EspacioE.**
- La instancia de la clase **containers.AccesosContainer** almacena las instancias de la clase **AccesoE.**
- La instancia de la clase **containers.TagsContainer** almacena las instancias de la clase **TagE.**

Existe además en el paquete una clase llamada **HomeContainer.** La misma contiene la instancia de la clase **Home** (del modelo de SH3D), la cual es guardada en el momento que se carga el plugin (a través de la clase **plugins.CargadorActions** y su método *getActions()*. Esta clase, difiere de las otras en que no almacena ninguna instancia del modelo extendido, sino que solo es utilizada para verificar que los objetos del plano que se asociaron sigan existiendo en un momento dado.

Control de cambios

Debido a que todas las identificaciones se pueden realizar al mismo tiempo que el plano se va dibujando en el software, debe existir un método que, cuando algo cambie en el plano, se verifiquen que los objetos identificados como parte del modelo extendido sigan cumpliendo

los requisitos. Para esto decidimos aplicar un mecanismo de listeners sobre las instancias del modelo SH3D que formen parte de las instancias del modelo extendido.

Si bien SH3D provee un mecanismo con estos fines, el mismo no es suficiente para satisfacer nuestras necesidades. El mecanismo utilizado por el software agrega a los objetos de SH3D listeners que escuchan determinados atributos (a través del método *addPropertyChangeListener(PropertyChangeListener listener)),* los cuales se limitan por cada clase a partir de su atributo *property*, que no es más que un tipo de datos enumerativo. Este atributo restringe las propiedades del objeto que pueden ser escuchadas.

Debido a que nuestra aplicación requiere monitorear la posición de los objetos asociados, y dicha posición, SH3D la representa, como mínimo, mediante 2 propiedades distintas, el uso de este mecanismo sin una buena metodología provocaría que un cambio de posición de un objeto ejecute más de una vez la acción propia del listener, lo que en nuestro caso sería totalmente ineficiente. Por ejemplo, si una pared se mueve, se dispara el listener 4 veces (las instancias de la clase Wall tienen 4 atributos de posición), lo que llevaría a verificar que el espacio cumpla los requisitos 4 veces. Esto conlleva a que usar el mecanismo de listeners de SH3D no es viable para la eficiencia de nuestra aplicación.

Para solucionar este problema, ampliamos el mecanismo existente a partir de un conjunto de paquetes (vale aclarar que está fuertemente restringido por el mecanismo de plugins y de listeners utilizado por SH3D). Las clases principales de dicho mecanismo se encuentran en los paquetes **listeners** y **listeners.changeContainers.** El paquete **listeners** contiene las siguientes clases:

- **TerrenoListener**
- **EspacioListener**
- **AccesoListener**
- **TagListener**
- **HomeListener**

Mientras que el paquete **listeners.changeContainers** contiene las clases:

- **TerrenoChangeContainer**
- **EspacioChangeContainer**
- **AccesoChangeContainer**
- **TagChangeContainer**

Cada clase en el paquete **listener** implementa 2 interfaces (excepto **HomeListener** que solo implementa una) con el propósito de actuar sobre cambios en el plano:

- **java.beans.PropertyChangeListener,** la cual requiere la implementación del método *propertyChange(ChangeEvent ev).* Mediante este método cada listener escucha los cambios de ciertos atributos de los objetos monitorizados.
- **com.eteks.sweethome3d.model.CollectionListener<T>,** la cual requiere la implementación del método *collectionChanged(CollectionEvent<T> ev).* Mediante este método, escuchamos cuando a una colección se le agrega o se le elimina un objeto.

La idea principal es la siguiente:

- Cada clase en el paquete **listeners.changeContainers** tiene como atributo un conjunto sin repetir de instancias del modelo SH3D que cambiaron en una acción ejecutada en el software.
- A través de la interface **java.beans.PropertyChangeListener** monitorizamos los objetos prestándole atención a los atributos de posicionamiento (en la clase **Room** la property es *POINT;* en la clase **Wall** las properties son *X_START, Y_START, X_END* e *Y_END;* y en la clase **HomePieceOfFurniture** las properties son *X e Y).*
- Si el listener es desencadenado por alguno de estos atributos, el objeto se agrega al conjunto de objetos a verificar de su respectivo ChangeContainer.
- Y por último se ejecuta el listener de la instancia de la clase **Home,** el cual escucha el atributo *MODIFIED* y ejecuta el método *VerificarCambios()* de todos los ChangeContainers, los cuales dependiendo si tiene objetos a verificar, verificará los requisitos.

A continuación explicaremos el momento de instanciación de cada listener y las acciones que cada uno desencadena:

Listeners.TagListener

Esta clase se instancia cuando se instancia la clase **containers.TagsContainer** y se almacena en su atributo *listener.* Su objetivo es el de controlar los cambios que se realizan sobre objetos identificados como Tags.

El listener controla dos clases de objetos:

- La instancia de la clase **Home,** agregándose como listener a través del método *addFurnitureListener.* Se agrega como listener a la colección de muebles debido a que un tag es un objeto de la clase **HomePieceOfFurniture** y SH3D almacena las instancias de esa clase en dicha colección.
- Instancias de la clase **HomePieceOfFurniture** que son identificadas como tags, agregándose como listener a través del método *addPropertyChangeListener.*

A la instancia de la clase **Home** se la comienza a monitorear al momento de la instanciación y se mantendrá durante toda la ejecución de la aplicación. El objetivo es controlar la colección de muebles de la instancia, provocando a que cada vez que un mueble se elimine o se agregue, SH3D ejecute el método *collectionChanged* del listener*,* el cual, dependiendo si el objeto fue previamente identificado como tag, se agregue a la lista de tags cambiados de la clase **TagChangeContainer**.

Por otro lado, a la instancia de la clase **HomePieceOfFurniture** se la comienza a monitorear al momento en que se identifica dicha instancia como Tag, y durará lo que dure la identificación. El objetivo es controlar los cambios en los atributos del objeto, provocando a que cada vez que cambie alguno de sus atributos, SH3D ejecute el método *PropertyChange* del listener*,* el cual, dependiendo si el atributo implica posicionamiento (propiedades *X* e *Y en este caso)*, agregará el objeto modificado a la lista de tags cambiados de la clase **TagChangeContainer**.

Listeners.AccesoListener

Esta clase se instancia cuando se instancia la clase **containers.AccesosContainer** y se almacena en su atributo *listener.* Su objetivo es el de controlar los cambios que se realizan sobre instancias del modelo SH3D identificados como Accesos.

El listener controla dos clases de objeto:

- La instancia de la clase **Home,** agregándose como listener a través del método *addFurnitureListener.* Se agrega como listener a la colección de muebles debido a que un acceso es un objeto de la clase **HomeDoorOrWindow** y SH3D almacena las instancias de esa clase en dicha colección.
- Instancias de la clase **HomeDoorOrWindow** que son identificadas como acceso, agregándose como listener a través del método *addPropertyChangeListener* de dicha instancia.

A la instancia de la clase **Home** se la comienza a monitorear al momento de la instanciación y se mantendrá durante toda la ejecución de la aplicación. El objetivo es controlar la colección de muebles de la instancia, provocando a que cada vez que un mueble se elimine o se agregue, SH3D ejecute el método *collectionChanged* del listener*,* el cual, dependiendo si el objeto fue previamente identificado como acceso, se agregue a la lista de accesos cambiados de la clase **AccesoChangeContainer**.

Por otro lado, a la instancia de la clase **HomeDoorOrWindow** se la comienza a monitorear al momento en que se identifica una puerta o ventana como acceso, y durará lo que dure la identificación. El objetivo es controlar los cambios en los atributos del objeto identificado como Acceso, provocando a que cada vez que cambie alguno de sus atributos, Sweet Home 3D ejecuta el método *PropertyChange* del listener*,* el cual, dependiendo si el atributo implica posicionamiento (propiedades *X* e *Y en este caso),* agregará el objeto modificado a la lista de accesos cambiados de la clase **AccesoChangeContainer**.

Listeners.EspacioListener

Esta clase se instancia cada vez que se agrega un nuevo espacio a la instancia de la clase **EspaciosContainer,** y se almacena relacionando el listener con el espacio que se creó*.* Su objetivo es el de controlar los cambios que se realizan sobre las paredes que forman el espacio identificado.

El listener controla dos clases de objeto:

- La instancia de la clase **Home,** agregándose como listener a través del método *addWallsListener.* Se agrega como listener a la colección de paredes debido a que un espacio está formado por un conjunto de paredes y SH3D almacena las instancias de esa clase en dicha colección.
- Instancias de la clase **Wall** que forman parte de al menos un espacio, agregándose como listener a través del método *addPropertyChangeListener* de dicha instancia.

A la instancia de la clase **Home** se la comienza a monitorear al momento de la instanciación del listener y se mantendrá mientras dure la identificación del espacio. El objetivo es controlar la colección de paredes de la instancia, provocando a que cada vez que una pared se elimine o se agregue, SH3D ejecute el método *collectionChanged* del listener, el cual, dependiendo si el objeto fue previamente identificado como pared en al menos un espacio, se agregue a la lista de paredes cambiadas de la clase **EspacioChangeContainer**.

A las instancias de la clase **Wall** que forman el espacio identificado se las comienza a monitorear al momento de la instanciación del listener y se mantendrá mientras dure la asociación. El objetivo es controlar los cambios en los atributos de cada una de las paredes del espacio, provocando a que cada vez que cambie alguno de sus atributos, SH3D ejecute el método *PropertyChange* del listener, el cual, dependiendo si el atributo implica posicionamiento (propiedades *X_START, Y_START, X_END e Y_END en este caso),* agregará la pared modificada a la lista de paredes cambiadas de la clase **EspacioChangeContainer**.

Listeners.TerrenoListener

Esta clase se instancia cuando se instancia la clase **containers.TerrenoContainer** y se almacena en su atributo *listener.* Su objetivo es el de controlar los cambios que se realizan sobre la habitación identificada como Terreno.

El listener controla dos clases de objeto:

- La instancia de la clase **Home,** agregándose como listener a través del método *addRoomsListener.* Se agrega como listener a la colección de habitaciones debido a que un Terreno es un objeto de la clase **Room** y SH3D almacena las instancias de esa clase en dicha colección.
- Instancia de la clase **Room** que es identificada como terreno en nuestro modelo, agregándose como listener a través del método *addPropertyChangeListener* de dicha instancia.

A la instancia de la clase **Home** se la comienza a monitorear al momento de la instanciación y se mantendrá durante toda la ejecución de la aplicación. El objetivo es controlar la colección de habitaciones de la instancia, provocando a que cada vez que una habitación se elimine o se agregue, SH3D ejecute el método *collectionChanged* del listener, el cual, dependiendo si el objeto fue previamente identificado como terreno, se agregue al mismo como habitación cambiada de la clase **TerrenoChangeContainer**.

Por otro lado, a la instancia de la clase **Room** se la comienza a monitorear al momento en que se la identifica como Terreno, y durará lo que dure la identificación. El objetivo es controlar los cambios en los atributos del objeto identificado como Terreno, provocando a que cada vez que cambie alguno de sus atributos, SH3D ejecute el método *PropertyChange* del listener, el cual, dependiendo si el atributo implica posicionamiento (propiedad *POINTS en este caso),* agregará al mismo como habitación cambiada de la clase **TerrenoChangeContainer**.

Listeners.HomeListener

Esta clase se instancia una única vez al momento en que se cargan el plugin. Su principal objetivo es controlar la propiedad MODIFIED de la instancia de la clase **Home.** Dicha propiedad es la que se modifica cuando el plano cambia, y no será nuevamente modificada hasta que se guarde el mismo. SH3D utiliza este atributo para controlar las modificaciones del plano y dependiendo de eso, notificar al usuario que el archivo no fue guardado. Debido a que necesitamos corroborar los cambios en el plano cada vez que el mismo sea modificado, decidimos intervenir dicha funcionalidad con el fin que nos interesa.

Esta clase, a diferencia de los demás listeners, solo implementa la interface **java.beans.PropertyChangeListener** con el objetivo de monitorear únicamente la propiedad anteriormente nombrada. Nuestro listener se agrega como tal a la instancia de la clase **Home** mediante el método *addPropertyChangeListener(Home.Property.MODIFIED), new HomeListener())* en el método *getActions() de la* clase **plugins.CargadorActions** (método que se invoca al momento de iniciar la aplicación para cargar el plugin). A partir de ese momento, cada vez que el atributo *modified* cambie, se invocará el método *propertyChange* del listener.

Debido a que el listener es cargado antes que cualquier otro listener, este método será invocado luego de todos los demás que explicamos, garantizando que cualquier objeto modificado haya sido guardado en su respectivo ChangeContainer (como se explicó anteriormente). Su función es invocar al método *VerificarCambios()* de todos los ChangeContainers.

ANEXO B - EJEMPLO

ANEXO B.1. CÓDIGO XML RESULTANTE DE LA EXPORTACIÓN DE NUESTRO MODELO

El siguiente es el código XML que resulta de la exportación del modelo generado por nuestra herramienta.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_45" class="java.beans.XMLDecoder">
 <object class="exportacion.Exportacion">
  <void property="terreno">
   <object class="modelo.Terreno">
    <void property="espacios">
     <void method="add">
      <object class="modelo.EspacioSimple">
       <void property="paredes">
        <void method="add">
         <object id="Pared0" class="modelo.Pared">
          <void property="alturaAlInicio">
           <float>250.0</float>
          </void>
          <void property="grosor">
           <float>7.5</float>
          </void>
          <void property="xFinal">
           <float>454.49997</float>
          </void>
          <void property="xInicio">
           <float>454.5</float>
          </void>
          <void property="yFinal">
           <float>121.278656</float>
          </void>
          <void property="yInicio">
           <float>516.2787</float>
          </void>
         </object>
        </void>
        <void method="add">
         <object id="Pared1" class="modelo.Pared">
          <void property="accesos">
           <void method="add">
            <object class="modelo.Acceso">
             <void property="altura">
              <float>208.5</float>
             </void>
             <void property="ancho">
              <float>91.5</float>
             </void>
             <void property="nombre">
              <string>Puerta</string>
             </void>
             <void property="pared">
```

```
      <object idref="Pared1"/>
     </void>
     <void property="profundidad">
      <float>14.500251</float>
     </void>
     <void property="x">
      <float>667.25</float>
     </void>
     <void property="y">
      <float>123.77864</float>
     </void>
    </object>
   </void>
  </void>
  <void property="alturaAlInicio">
   <float>250.0</float>
  </void>
  <void property="grosor">
   <float>7.5</float>
  </void>
  <void property="xFinal">
   <float>919.5</float>
  </void>
  <void property="xInicio">
   <float>454.5</float>
  </void>
  <void property="yFinal">
   <float>121.278656</float>
  </void>
  <void property="yInicio">
   <float>121.278656</float>
  </void>
 </object>
</void>
<void method="add">
 <object class="modelo.Pared">
  <void property="alturaAlInicio">
   <float>250.0</float>
  </void>
  <void property="grosor">
   <float>7.5</float>
  </void>
  <void property="xFinal">
   <float>919.5</float>
  </void>
  <void property="xInicio">
   <float>919.5</float>
  </void>
  <void property="yFinal">
   <float>516.2787</float>
  </void>
  <void property="yInicio">
   <float>121.278656</float>
  </void>
 </object>
</void>
```

```
   <void method="add">
    <object id="Pared2" class="modelo.Pared">
     <void property="alturaAlInicio">
      <float>250.0</float>
     </void>
     <void property="grosor">
      <float>7.5</float>
     </void>
     <void property="xFinal">
      <float>454.5</float>
     </void>
     <void property="xInicio">
      <float>919.5</float>
     </void>
     <void property="yFinal">
      <float>516.2787</float>
     </void>
     <void property="yInicio">
      <float>516.2787</float>
     </void>
    </object>
   </void>
  </void>
  <void property="tags">
   <void method="add">
    <object class="modelo.Tag">
     <void property="altura">
      <float>5.0</float>
     </void>
     <void property="ancho">
      <float>5.0</float>
     </void>
     <void property="codigo">
      <string></string>
     </void>
     <void property="nombre">
      <string>Tag</string>
     </void>
     <void property="profundidad">
      <float>0.1</float>
     </void>
     <void property="x">
      <float>519.0</float>
     </void>
     <void property="y">
      <float>333.6136</float>
     </void>
    </object>
   </void>
  </void>
 </object>
</void>
<void method="add">
 <object class="modelo.EspacioSimple">
  <void property="obstaculos">
   <void method="add">
```

```
 <object class="modelo.Obstaculo">
  <void property="altura">
   <float>90.0</float>
  </void>
  <void property="ancho">
   <float>40.0</float>
  </void>
  <void property="nombre">
   <string>Silla</string>
  </void>
  <void property="profundidad">
   <float>42.0</float>
  </void>
  <void property="x">
   <float>542.5</float>
  </void>
  <void property="y">
   <float>789.5592</float>
  </void>
 </object>
</void>
<void method="add">
 <object class="modelo.Obstaculo">
  <void property="altura">
   <float>90.0</float>
  </void>
  <void property="ancho">
   <float>40.0</float>
  </void>
  <void property="nombre">
   <string>Silla</string>
  </void>
  <void property="profundidad">
   <float>42.0</float>
  </void>
  <void property="x">
   <float>536.5</float>
  </void>
  <void property="y">
   <float>594.5592</float>
  </void>
 </object>
</void>
<void method="add">
 <object class="modelo.Obstaculo">
  <void property="altura">
   <float>90.0</float>
  </void>
  <void property="ancho">
   <float>40.0</float>
  </void>
  <void property="nombre">
   <string>Silla</string>
  </void>
  <void property="profundidad">
   <float>42.0</float>
```

```
  </void>
  <void property="x">
   <float>629.5</float>
  </void>
  <void property="y">
   <float>780.5592</float>
  </void>
 </object>
</void>
<void method="add">
 <object class="modelo.Obstaculo">
  <void property="altura">
   <float>74.0</float>
  </void>
  <void property="ancho">
   <float>170.0</float>
  </void>
  <void property="idCatalogo">
   <string>Sleipnir1#rectangularTable</string>
  </void>
  <void property="nombre">
   <string>Mesa Rectangular</string>
  </void>
  <void property="profundidad">
   <float>90.0</float>
  </void>
  <void property="x">
   <float>574.5</float>
  </void>
  <void property="y">
   <float>690.5592</float>
  </void>
 </object>
</void>
<void method="add">
 <object class="modelo.Obstaculo">
  <void property="altura">
   <float>90.0</float>
  </void>
  <void property="ancho">
   <float>40.0</float>
  </void>
  <void property="nombre">
   <string>Silla</string>
  </void>
  <void property="profundidad">
   <float>42.0</float>
  </void>
  <void property="x">
   <float>641.5</float>
  </void>
  <void property="y">
   <float>594.5592</float>
  </void>
 </object>
</void>
```

```
</void>
<void property="paredes">
 <void method="add">
  <object id="Pared3" class="modelo.Pared">
   <void property="accesos">
    <void method="add">
     <object class="modelo.Acceso">
      <void property="altura">
       <float>208.5</float>
      </void>
      <void property="ancho">
       <float>91.5</float>
      </void>
      <void property="angulo">
       <float>4.712389</float>
      </void>
      <void property="nombre">
       <string>Puerta</string>
      </void>
      <void property="pared">
       <object idref="Pared3"/>
      </void>
      <void property="profundidad">
       <float>14.500251</float>
      </void>
      <void property="x">
       <float>131.99998</float>
      </void>
      <void property="y">
       <float>394.30917</float>
      </void>
     </object>
    </void>
   </void>
   <void property="alturaAlInicio">
    <float>250.0</float>
   </void>
   <void property="grosor">
    <float>7.5</float>
   </void>
   <void property="xFinal">
    <float>129.5</float>
   </void>
   <void property="xInicio">
    <float>129.5</float>
   </void>
   <void property="yFinal">
    <float>916.2787</float>
   </void>
   <void property="yInicio">
    <float>121.278656</float>
   </void>
  </object>
 </void>
 <void method="add">
  <object class="modelo.Pared">
```

```
  <void property="alturaAlInicio">
   <float>250.0</float>
  </void>
  <void property="grosor">
   <float>7.5</float>
  </void>
  <void property="xFinal">
   <float>919.5</float>
  </void>
  <void property="xInicio">
   <float>129.5</float>
  </void>
  <void property="yFinal">
   <float>916.2787</float>
  </void>
  <void property="yInicio">
   <float>916.2787</float>
  </void>
 </object>
</void>
<void method="add">
 <object class="modelo.Pared">
  <void property="alturaAlInicio">
   <float>250.0</float>
  </void>
  <void property="grosor">
   <float>7.5</float>
  </void>
  <void property="xFinal">
   <float>919.5</float>
  </void>
  <void property="xInicio">
   <float>919.5</float>
  </void>
  <void property="yFinal">
   <float>516.2787</float>
  </void>
  <void property="yInicio">
   <float>916.2787</float>
  </void>
 </object>
</void>
<void method="add">
 <object idref="Pared2"/>
</void>
<void method="add">
 <object idref="Pared0"/>
</void>
<void method="add">
 <object class="modelo.Pared">
  <void property="alturaAlInicio">
   <float>250.0</float>
  </void>
  <void property="grosor">
   <float>7.5</float>
  </void>
```

```
      <void property="xFinal">
       <float>129.5</float>
      </void>
      <void property="xInicio">
       <float>454.49997</float>
      </void>
      <void property="yFinal">
       <float>121.278656</float>
      </void>
      <void property="yInicio">
       <float>121.278656</float>
      </void>
     </object>
    </void>
   </void>
  </object>
 </void>
</void>
<void property="points">
 <array class="[F" length="4">
  <void index="0">
   <array class="float" length="2">
    <void index="0">
     <float>9.5</float>
    </void>
    <void index="1">
     <float>13.278656</float>
    </void>
   </array>
  </void>
  <void index="1">
   <array class="float" length="2">
    <void index="0">
     <float>9.5</float>
    </void>
    <void index="1">
     <float>1013.2787</float>
    </void>
   </array>
  </void>
  <void index="2">
   <array class="float" length="2">
    <void index="0">
     <float>1009.5</float>
    </void>
    <void index="1">
     <float>1013.2787</float>
    </void>
   </array>
  </void>
  <void index="3">
   <array class="float" length="2">
    <void index="0">
     <float>1009.5</float>
    </void>
    <void index="1">
```

```
          <float>13.278656</float>
         </void>
        </array>
       </void>
      </array>
     </void>
    </object>
   </void>
  </object>
</java>
```