UNIVERSIDAD CARLOS III DE MADRID

Desarrollo de un escenario basado en la herramienta Greenfoot para el apoyo de la enseñanza temprana de la Programación Orientada a Objetos

Proyecto Fin de Carrera

Escuela Politécnica Superior

Alumno: Rosa Romero Gómez NIA: 100060426
Tutor: Susana Montero
Titulación: Ingeniería Técnica en Informática de Gestión

Índice de Contenido

Índice de Figuras

AGRADECIMIENTOS

A todos mis "amigos" de la Universidad que han estado conmigo en estos 4 años inolvidables, ellos han hecho que se me pasen volando y que mire con nostalgia los comienzos. Lo mejor de todo, es que siguen estando ahí porque ya no son solo compañeros, son más que eso, son mis amigos.

"Hagas lo que hagas, no seas un ladrillo más en el muro"

1 Introducción

El objetivo de este capítulo es presentar el trabajo realizado y describir el contenido del documento. Con esta introducción se describe brevemente el problema planteado y la solución llevada a cabo, los objetivos que se pretenden cubrir con la solución propuesta y la estructura que posee el documento, presentado a continuación.

1.1 Planteamiento del problema

Este documento presenta un proyecto final de carrera orientado a la innovación docente en la enseñanza temprana de los paradigmas de la Programación Orientada a Objetos (a partir de ahora se utilizarán las siglas POO).

El proyecto nace de la necesidad del personal docente de la asignatura Programación del primer curso de Ingeniería de Telecomunicación de la Universidad Carlos III de Madrid, de proporcionar a sus estudiantes un mecanismo con el que reafirmen los principios de la POO aprendidos en la parte teórica de la asignatura.

Los conceptos sobre los que se trabaja principalmente son la creación de objetos, incidiendo en la distinción entre clase y objeto, invocación de métodos de un objeto, inspección del estado de un objeto e interacción con estos objetos.

Esta reafirmación de conceptos se realiza a través de la adaptación de las prácticas de la asignatura a un entorno gráfico, utilizando para ello la herramienta software Greenfoot [1], a la cual da soporte Sun Microsystems [2].

El sistema Greenfoot proporciona exactamente la manera que se busca de introducir la POO, aportando un sofisticado entorno interactivo que permite una creación de objetos sencilla y provee de una visualización del comportamiento e interacción directa entre objetos.

1.2 Objetivos

El objetivo de este proyecto es el desarrollo de una solución basada en la herramienta software Greenfoot para el apoyo de la enseñanza temprana en la POO. Los objetivos que se pretenden satisfacer son:

- Desarrollo de un escenario basado en la herramienta Greenfoot que pretende proveer al estudiante de una manera nueva de reafirmar los conceptos básicos que debe poseer de la POO:
    - Diferenciación entre clase y objeto.
    - Creación de objetos.
    - Invocación de métodos que posee una clase.
    - Inspección y edición del estado de un objeto.

- El estudiante no necesitará tener ningún conocimiento acerca de la herramienta Greenfoot, puesto que su código Java implementado en la herramienta jGrasp, es adaptado totalmente a Greenfoot sin que ellos tengan que realizar ningún cambio.

1.3 Estructura del documento

Este documento lo componen varios capítulos y se presenta la siguiente estructura de contenidos:

1. **Introducción:** se hace un resumen del propósito del documento, comentando los objetivos que se persiguen con el desarrollo de la aplicación en Greenfoot y los no perseguidos.

2. **Estado de la cuestión:** en este capítulo se comenta cual es la situación actual, análisis de la enseñanza de la POO y los problemas existentes en su aprendizaje, presentación y estudio comparativo de las diferentes herramientas útiles que se han desarrollado para la ayuda a la enseñanza y descripción de la herramienta Greenfoot como herramienta software que mejor se adapta a nuestra problemática.

3. **Gestión del proyecto:** se presenta una estimación del proyecto realizado y se presentan las conclusiones en cuanto a una estimación temprana y una estimación tras la finalización del proyecto.

4. **Planteamiento del problema y solución:** en este capítulo se realiza una descripción de la solución llevada a cabo. Para ello se presenta el contexto del problema que se pretende resolver y la solución definida con una explicación razonada del porqué de la presentación de esta solución.

4. **Análisis:** donde se definen y especifican los requisitos del cliente y como consecuencia, se identifican los requisitos funcionales, no funcionales, de dominio y de usabilidad del sistema.

5. **Diseño:** en este capítulo se especifica el diseño que satisface los requisitos identificados anteriormente. Se presenta la arquitectura del sistema, así como los diagramas de clases y de secuencia del mismo.

6. **Implementación:** se presentan en este capítulo los detalles de implementación del sistema.

7. **Pruebas:** en este capítulo se presenta el plan de pruebas del sistema que se va a llevar a cabo y los casos de prueba basados en los casos de uso del sistema.

8. **Conclusiones**: Para finalizar se comentan las conclusiones obtenidas como resultado de la elaboración del proyecto y se exponen los trabajos futuros que podrían desempeñarse en base a la solución presentada.

9. **Bibliografía:** En este apartado se enumeran las referencias de la literatura utilizada en todo el proyecto.

10. **Definiciones y acrónimos:** en este capítulo se encuentran las definiciones y acrónimos utilizados en el documento.

11. **Anexos**: Diagrama Gantt realizado en la gestión del proyecto.

2 Estado de la cuestión

En este capítulo se presentan los estudios previos que se han realizado en cuanto a la problemática que se nos presenta y se especifican todas las tecnologías estudiadas para la elaboración del proyecto.

2.1 Programación Orientada a Objetos en la enseñanza.

La enseñanza de programación ha cambiado considerablemente desde la introducción generalizada a la orientación a objetos en los cursos de programación. La aceptación de una enseñanza temprana de la orientación a objetos junto con la inclusión de puntos de vista provenientes de la ingeniería del software (como el uso de grandes proyectos, librerías de clases, mantenimiento de código, etc.) han dado lugar a una complicada infraestructura contra la que los nuevos estudiantes tienen que hacer frente.

Esto incluye frecuentemente múltiples archivos de código fuente, librerías de clases y entornos de configuración para hacer frente a dependencias de compilación. Como resultado, un número de herramientas para la enseñanza han llegado a convertirse en una popular forma para los estudiantes de enfrentarse a la creciente complejidad de la infraestructura. Estas herramientas ofrecen una gama que va, desde las librerías de clases para tareas especificas sobre marcos temáticos hasta entornos de programación.

2.1.1 Aproximación a la orientación a objetos.

La investigación de una primera aproximación a la orientación a objetos está basada en la premisa por la cual los estudiantes deberían estudiar programación orientada a objetos desde el comienzo, para evitar así un cambio de paradigmas de programación que ocurre en una aproximación tardía a la orientación a objetos. Sin embargo, esta primera aproximación puede resultar problemática, porque se requiere de los estudiantes conocimientos simultáneos tales como:

- Conceptos generales de computación tales como el de código fuente, compilación, ejecución...
- Paradigmas independientes de programación como son la asignación, invocación y declaración de procedimientos y funciones, paso de parámetros...
- Conceptos específicos de la orientación a objetos, tales como clase, objeto, constructor, accesos...

2.1.2 Problemas con la visualización de la orientación a objetos.

La dinámica de la ejecución de un programa orientado a objetos es bastante compleja cuando es enunciada completamente, pero estos detalles son esenciales para un correcto entendimiento de la POO por parte del estudiante. Enunciamos los principales escenarios que pueden ocurrir:

- **Creación de objetos:** la declaración de la clase es consultada para determinar los campos y así la memoria puede ser asignada a ese objeto. Después, bien por defecto o por inicializaciones explicitas de los campos, el constructor es ejecutado. Las instrucciones del constructor son de nuevo obtenidas de la declaración de la clase. Desde que los constructores son más frecuentemente usados para inicializar campos, el flujo de datos va desde los parámetros actuales a los parámetros formales y a continuación por asignación, hasta los campos del objeto.

- **Invocación de un método de un objeto:** la declaración de la clase es consultada para obtener la secuencia de instrucción del método. Los parámetros actuales son evaluados y pasados a parámetros formales. Las instrucciones son ejecutadas, donde se referencian identificadores son para identificar que son declarados en la clase, mientras los propios valores son campos del objeto.

Estos escenarios son esenciales para el entendimiento de la POO y deben ser concebidos por el estudiante. En la terminología del constructivismo, un modelo mental debe ser construido. Estos detalles de conocimiento deben ser explícitamente enseñados, porque hay demasiado trabajo personal y los estudiantes son muy propensos a la construcción de modelos no viables.

De particular importancia es la interacción entre las declaraciones estáticas de campos y métodos en las clases y el acceso dinámico y modificación de valores, en cada objeto de una clase. Algunos de los errores que se han encontrado en la enseñanza temprana de la orientación a objetos son los siguientes:

- La diferencia entre una clase y el objeto de una clase.

- La ejecución de un constructor como parte de la creación de un objeto.

- Las operaciones solo pueden ser invocadas por objetos.

- El estado de un objeto (los valores de sus campos) y el hecho de que una operación puede cambiar un estado.

- La conexión entre ejecución de una operación y su código fuente.

- Parámetros actuales vs. Parámetros formales.

- La relación de usos entre clases, en particular, el valor de un campo de un objeto puede ser un objeto.

2.1.3 Problemas del aprendizaje de Java.

Para la realización de las prácticas de la asignatura de Ingeniería de Telecomunicación se utiliza como lenguaje de programación, el lenguaje Java, por lo tanto se van a comentar brevemente algunos problemas encontrados en su aprendizaje.

Existen algunos aspectos a considerar en la enseñanza de Java, vamos a razonar la repercusión de esta tecnología para la enseñanza de la programación desde edades tempranas.

Debido a su popularidad en el contexto de aplicaciones Web y la facilidad con la que los principiantes pueden producir programas gráficos, Java se ha convertido en uno de los lenguajes más usados en los cursos de introducción a la programación.

El resultado de todo esto es un estudiante que sabe como unir un simple programa pero no sabe cómo funciona ese programa. Llegando más lejos, el temprano uso de las librerías Java y los entornos de desarrollo (en inglés, *"frameworks")* hacen imposible, para un estudiante, el desarrollo de la conciencia del coste de tiempo de ejecución, porque es extremadamente duro saber qué llamada a método ejecutará eventualmente.
Por lo tanto se llega a la conclusión de que se aprende Java a base de prueba y error, formando así codificadores no programadores. [3]

2.1.4 Java como primer lenguaje de programación.

Java puede ser utilizado como primer lenguaje de programación siempre y cuando antes de estudiar dicho lenguaje, se forme al alumno en el desarrollo de la lógica. Pues el principal problema empieza en el momento en que un futuro programador no es capaz de alcanzar una clara comprensión del problema a resolver (análisis), ni identificar los conceptos claves implicados (diseño) para finalmente expresar la solución en un programa (programación).

En caso de no promover la resolución de problemas y pensamiento en profundidad se produce un estilo de programación de ensayo y error e impide que los estudiantes adquieran la disciplina de la programación, convirtiéndose únicamente en codificadores. Los hábitos de los programadores están influenciados por el primer lenguaje que aprenden y es difícil, algunas veces, abandonar ciertos hábitos adquiridos al programar en un único lenguaje. Aunque existen críticas al uso de Java como primer o único lenguaje de programación, éste tiene un importante rol en la instrucción de los programadores.

A continuación se mencionan dos aspectos del lenguaje que deben formar parte del conocimiento de un programador real:

- Un entendimiento de la programación concurrente (por la cual, los hilos proveen de un modelo de bajo nivel).

- Reflexión, llamado así el entendimiento por el cual, un programa puede examinar su propio estado y determinar su propio comportamiento en un entorno dinámico de cambio. [4]

2.1.5 Problemas que presentan entornos de desarrollo existentes, en Java.

Como se citaba anteriormente, Java está haciéndose muy popular, tanto en el desarrollo de aplicaciones como para la enseñanza de la POO y por lo tanto, existen muchos entornos de programación.

Muchos aspectos en los entornos actuales existentes, causan una larga lista de problemas a la hora de enseñar a los estudiantes. Se enumeran, a continuación, los problemas existentes más comunes en estos entornos:

- **El entorno no es realmente orientado a objetos:** un *entorno para un lenguaje orientado a objetos* no lo convierte en un *entorno orientado a objetos*. El propio entorno debe refleja el paradigma del lenguaje. En particular, las abstracciones con las que trabajan los estudiantes deberían ser clases y objetos. En muchos entornos existentes, en lugar de eso, los estudiantes tienen que trabajar con ficheros. Están forzados a pensar sobre el sistema de ficheros del sistema operativo y en la estructura de los directorios. La creación de proyectos puede ser una pesadilla. Todo esto crea una sobrecarga que dificulta la enseñanza y distrae de los temas importantes. Los objetos como entidades de interacción, no se admiten en todos los entornos Java.

- **El entorno es insuficiente o exagerado:** muchos profesores no usan entornos integrados (usualmente porque tienen problemas para encontrar uno adecuado). En ese caso, los estudiantes trabajan desde línea de comandos y gastando un tiempo considerable en familiarizarse con Unix o DOS en lugar de aprender sobre programación. Muchas oportunidades se pierden para mejorar la enseñanza y el aprendizaje a través del uso de herramientas mejores. Otros entornos son desarrollados para usuarios más profesionales y presentan un conjunto abrumador de componentes y funcionalidades. Los estudiantes están perdidos en estos entornos y el efecto puede ser tan malo como no tener un entorno integrado. Otros entornos, son modificaciones de entornos no orientados a objetos y ofrecen abstracciones erróneas. Por lo tanto, herramientas minimalistas, herramientas complicadas o herramientas erróneas pueden causar problemas considerables.

- **El entorno se centra en la interfaz de usuario:** muchos entornos gráficos usan estos gráficos erróneamente. En particular, muchos entornos se centran en construir interfaces gráficas de usuario (el acrónimo en inglés es GUIs). Construir GUIs desde el comienzo transmite una imagen muy distorsionada de la programación y de la orientación a objetos. Los estudiantes gastan su tiempo en pulsar botones en lugar de pensar en cómo construir una aplicación.
  Al mismo tiempo, un uso mucho más beneficioso de los gráficos es

descuidado: visualización de la estructura de clases. La estructura de estos programas puede ser representada gráficamente de una manera que la hace mucho más fácil de entender. Pocos entornos existentes hacen buen uso de esto.

- **El entorno es caro:** el coste es un gran problema todavía en algunos entornos. Un efecto de esto es el problema que tienen los estudiantes para trabajar en casa, adquirir una licencia para el desarrollo en casa puede ser caro.[5]

2.1.6 Herramientas existentes para la ayuda a la enseñanza de la POO.

Mientras un número de herramientas útiles han sido desarrolladas, existe un área considerable para la mejora. Específicamente, la introducción a la orientación a objetos en niveles inferiores al universitario tiene un desarrollo muy reciente que puede beneficiarse de más apoyo de este tipo de herramientas. Estas herramientas están basadas en los conocimientos de los que se dispone.

Estas herramientas de apoyo se **basan en la animación para ayudar al estudiante a visualizar lo que un programa orientado a objetos está realizando en todo momento**. Los principios de diseño de animación deben corresponderse con completitud y continuidad.

"Completitud" significa que cada característica del programa debe ser visualizada, por ejemplo, un valor constante puede no aparecer de ningún sitio.

"Continuidad" quiere decir, que la animación debe establecer las relaciones entre el programa explicito y las acciones.

Qué herramienta se decide usar depende de lo que ya se sabe sobre programación y específicamente, sobre la programación en Java, en nuestro caso. Las herramientas de las que se dispone son diseñadas para enseñar a programar visualmente, en lugar de comenzar con un montón confuso de líneas de código. Estas aplicaciones demuestran las relaciones entre objetos (se aprende sobre los objetos con estas herramientas), y como hacer que estos objetos interactúen y realicen cosas. Cada herramienta de enseñanza ha sido diseñada para una audiencia específica. [6]

A continuación vamos a enumerar las ventajas y las desventajas existentes para las tres herramientas analizadas:

❖ **BlueJ** [7]

- **Ventajas**

✓ Hace énfasis en la relación entre el diagrama de clases y el código para facilitar la adquisición de conceptos de orientación a objetos.

- ✓ El entorno permite la interacción con clases y objetos mediante sus menús contextuales. Con el menú contextual de las clases podemos crear instancias que aparecerán en el "banco de objetos".

- ✓ Integración del editor de código fuente y los mensajes de error del compilador.

- **Desventajas**

- ✓ Dificultad de instalación e inestabilidad del producto.
  Hay que proporcionar al usuario las instrucciones de instalación y configuración, lo cual aunque sea un proceso sencillo ralentiza la puesta en marcha del aprendizaje del estudiante. [8]

- ✓ No proporciona ayuda al estudiante sobre la interpretación de los errores cometidos, ni en la solución de los mismos.[9]

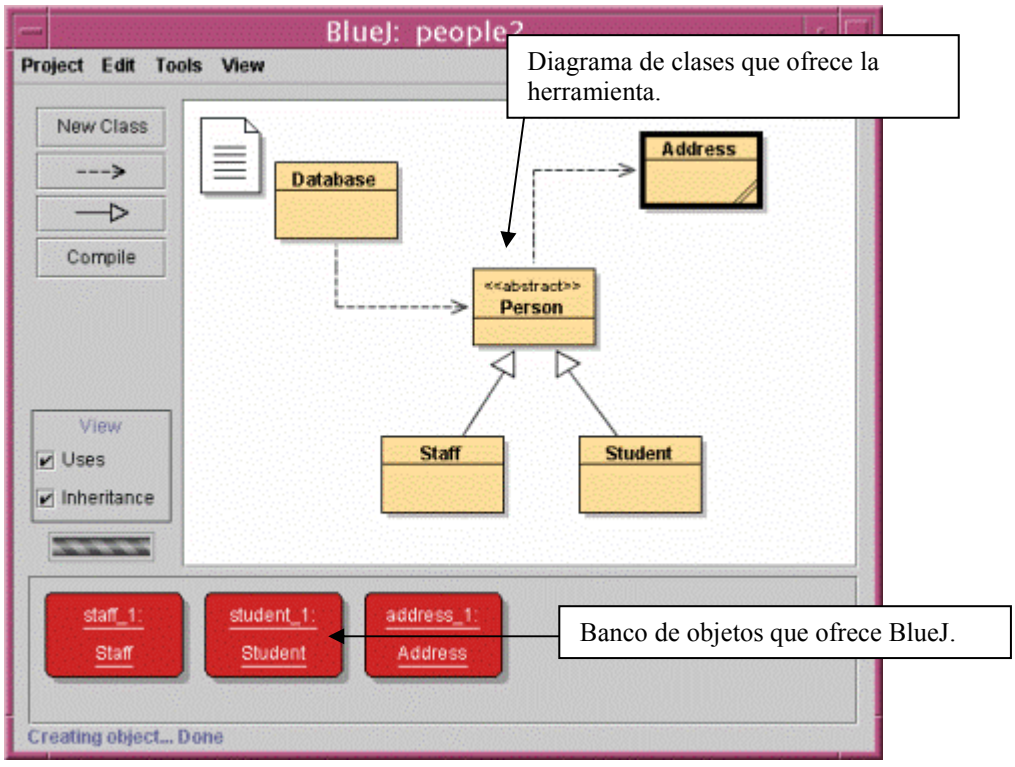


Figura 2.1.6.1: Interfaz gráfica de BlueJ

❖ **Jeliot 3 [10]**

- **Ventajas**

  ✓ Es una aplicación que permite visualizar cómo un programa en Java es interpretado.

  ✓ Muestra el funcionamiento en una pantalla como una animación continúa, que permite al estudiante seguir paso a paso la creación de variables, las llamadas a los métodos, etc.

- **Desventajas**

  ✓ No hay límite en el tamaño de los programas, sin embargo, todas las clases estarán en un archivo y debido al espacio limitado del fotograma de animación, la visualización de muchos objetos es uno de los problemas que presenta Jeliot 3.

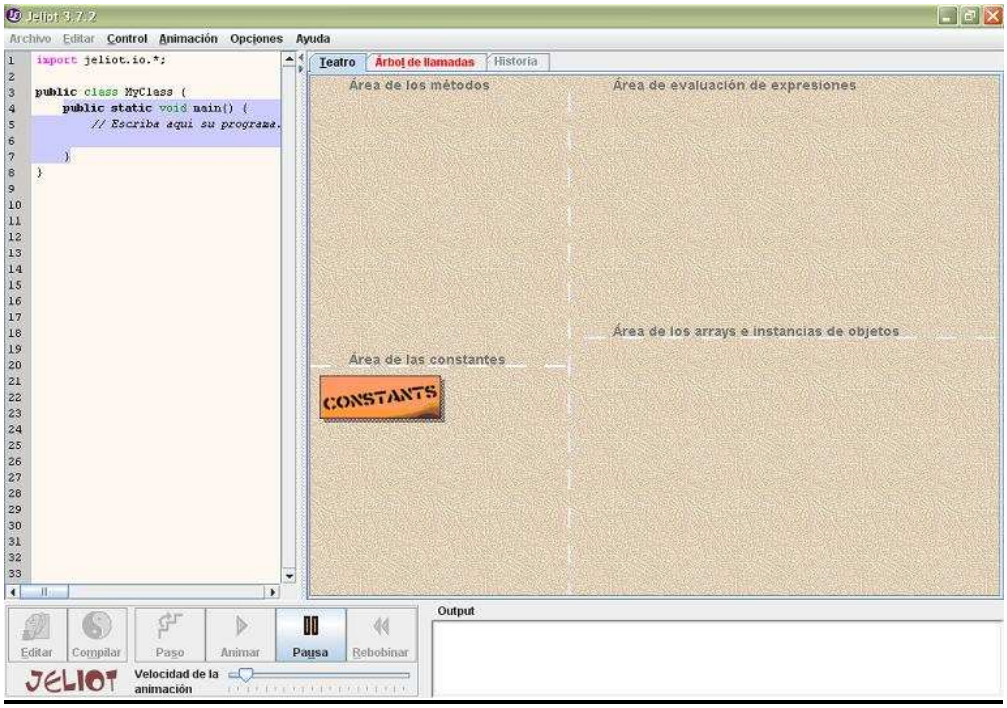


Figura 2.1.6.2: Interfaz gráfica de Jeliot 3.

❖ **Alice [11]**

- **Ventajas [12]**

  ✓ Programación incremental, se desarrolla un método cada vez y se va probando unitariamente.

  ✓ Una firme conciencia de los objetos, debido al fuerte entorno visual existente.

  ✓ El concepto de método se define como una petición a un objeto para que haga una actividad determinada. Para que el objeto realice la actividad, se le debe enviar un mensaje.

  ✓ Fuerte sentido de la herencia entre clases.

  ✓ Los estudiantes pueden desarrollar objetos individualmente y luego combinarlos para conseguir proyectos de grupo.

- **Desventajas**

  ✓ Se necesita demasiado tiempo en enseñar al alumno a desenvolverse en el entorno.

  ✓ Los conceptos aprendidos a veces son muy específicos del entorno, haciéndolos menos universales y restando así su aplicabilidad.

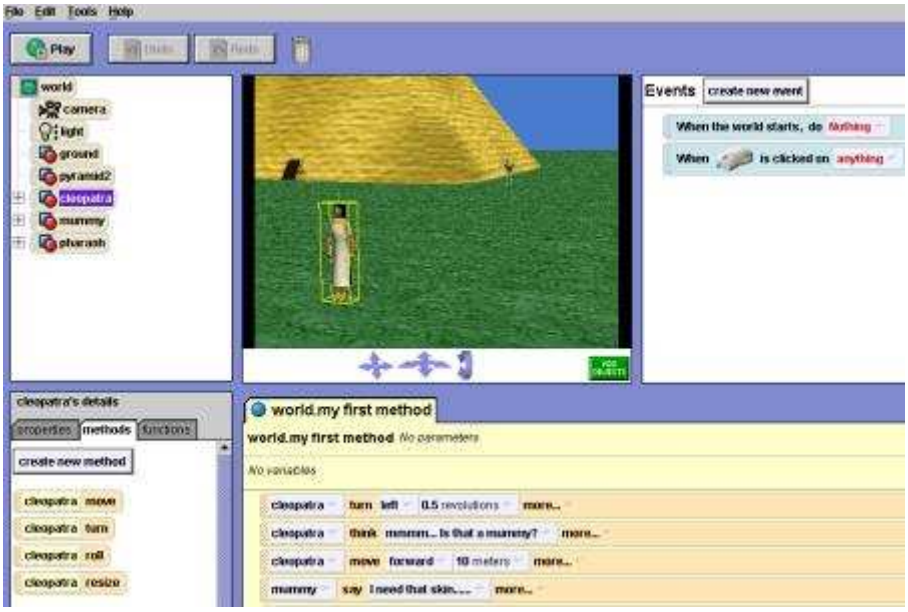


Figura 2.1.6.3: Interfaz gráfica de Alice.

2.2 ¿Qué es Greenfoot?

Greenfoot es una herramienta software diseñada para proporcionar a los principiantes cierta experiencia en la programación orientada a objetos. Apoya el desarrollo de aplicaciones gráficas en el lenguaje de programación Java. Esta herramienta fue diseñada e implementada en la Universidad de Kent (Inglaterra) y por la Universidad Deakin (Melbourne, Australia).

El diseño de Greenfoot está inspirado originalmente considerando la combinación de características de dos de los más populares tipos de entornos de enseñanza: **"microworlds"** (micromundos), como "*Karel te Robot"* [13] y **entornos de interacción directa** como el BlueJ. Uno de los puntos fuertes de los "micromundos" es la visualización excelente de los objetos, su estado y su comportamiento. Pero carece por otro lado, de un medio para la interacción directa de los objetos.

La herramienta BlueJ provee de un fuerte entorno para la interacción directa de objetos, pero carece de la visualización de objetos en detalle comparada con la de "*Karel te Robot*". "Carel te Robot" existe como un marco o estructura Java y BlueJ es un entorno Java, es posible ejecutar Carel con BlueJ para conseguir los beneficios de ambos. Cuando se hace esto, muchos objetos son representados dos veces (una vez en BlueJ y otra en Carel te Robot) y diferentes aspectos de visualización o interacción son extendidos a distintos puntos de vista de un mismo objeto. Esta confusión es bastante problemática para los nuevos estudiantes.

Greenfoot combina las funcionalidades sin los problemas de representación. Otro problema de los "microworlds" es que están típicamente restringidos a escenarios únicos, esto puede crear problemas en tareas de adaptación para intereses específicos de diversos grupos destinatarios. El sistema Greenfoot intenta resolver estos problemas aportando un sofisticado "*meta-framework*" (meta-estructura) interactivo. Esta estructura hace más fácil la creación de variados "micromundos" con diferentes escenarios, mientras provee de un soporte incorporado de visualización del comportamiento e interacción directa entre objetos. [14]

2.3 Objetivos de diseño de la herramienta Greenfoot.

El objetivo principal del entorno Greenfoot podría resumirse como "adecuado para la enseñanza de la orientación a objetos a nivel escolar". Se debe apuntar que el centrarse en los primeros niveles no excluye el uso de la herramienta a nivel universitario.

- **Experimentación y retroalimentación visual (visual feedback).**

Se intenta que el sistema sea altamente visual e interactivo. Los usuarios deben ser capaces de experimentar con instanciaciones de conceptos directamente, vía interfaz de usuario y adquirir un entendimiento de conceptos importantes a través de la retroalimentación visual.

Se espera que con esto se contribuya satisfactoriamente al reto de atraer a estudiantes sin ningún interés importante por la programación.

Sorprendentemente un balance entre simplicidad y riqueza de la herramienta es importante. Si el programa no es simple en su uso, los usuarios podrían perder el interés y el uso de esta herramienta podría no ser productivo.

- **Flexibilidad en los escenarios.**

Para atraer el interés de los estudiantes, el sistema necesita ser capaz de poseer ejemplos adecuados a su edad, género y características personales y culturales, además de otros factores individuales. Para el sistema de diseño, esto significa que Greenfoot, debe proveer de una gran variedad de diferentes actividades y escenarios. Creando flexibilidad en escenarios permite variar la complejidad y así el material va siendo aprendido según su nivel de dificultad.

- **Clara enseñanza de conceptos de la orientación a objetos.**

El primer foco debe ser el desarrollo del entendimiento en los estudiantes de los conceptos usados en la programación orientada a objetos. Con el uso de la herramienta Greenfoot los estudiantes deben familiarizarse con los conceptos fundamentales de la orientación a objetos como las clases, objetos, invocación de métodos y conceptos imperativos de programación.

- **Fácil desarrollo de escenarios y ejercicios.**

Parte del gran objetivo de proveer de un buen soporte para profesores es que éste debe aportar un fácil desarrollo de escenarios y ejercicios. Greenfoot debe intentar realizar el desarrollo de un ejercicio y de un escenario lo suficientemente fácil para que muchos profesores puedan desarrollar sus propias versiones. Esto significa que el escenario a nivel de usuario debe estar separado de la estructura general de implementación.

- **Apoyo para la migración a otros entornos.**

El sistema Greenfoot debe ser diseñado para que los conceptos y conocimientos aprendidos puedan ser transferidos fácilmente a otros entornos como Blues que puede ser usado como el siguiente entorno de desarrollo.

- **Generación automática de documentación.**

Greenfoot posee integrada la herramienta Jabado que es una utilidad provista por Sun Microsystems para la generación de documentación de APIs en formato formato HTML a partir de código fuente Java. Esto proporciona también una buena costumbre al programar, que es comentar los programas para que sean entendibles por cualquier persona que no es el desarrollador del código. Jabado es el estándar de la industria para documentar clases de Java. [15]

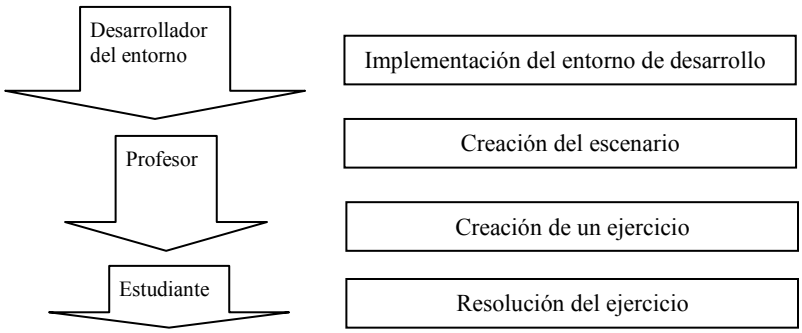


Figura 2.3: Actividades en Greenfoot.

3 Gestión del proyecto

En este capítulo se describe la gestión del proyecto presentado en este documento. Se realiza una estimación del esfuerzo del proyecto. Esta estimación es realizada antes del comienzo de las fases que lo componen, se utiliza para valorar la viabilidad del proyecto realizado.

3.1 Plan de trabajo

A continuación mostramos las fechas y duración de las tareas planificadas para el desarrollo del proyecto. Este plan de trabajo incluye una columna con las fechas del desarrollo real del proyecto que es el desarrollo que se incluye en el Gantt incluido en el Anexo

Tareas	Planificado		Re-Planificado		Real		Tiempo	Observaciones
	Desde	Hasta	Desde	Hasta	Desde	Hasta	dedicado	
Estudio preliminar								
Estado de la cuestión	06/10/08	22/10/08			06/10/08	22/10/08	2h/día	Revisar tecnologías, métodos, herramientas existentes y trabajos anteriores.
Proyecto dirigido	3/11/08	6/02/09	3/11/08	6/05/09	3/11/08	6/05/09	3h/día	Aprendizaje de tecnología Java y desarrollo en la herramienta Greenfoot de varios escenarios.
Planteamiento del problema	18/05/09	18/05/09			18/05/09	18/05/09	3h/día	Breve descripción del problema.
Esbozo de la solución	20/05/09	27/05/09			20/05/09	27/05/09	2h/día	Breve descripción de la solución.
Desarrollo								
Definición de requisitos	01/06/09	05/06/09			01/06/09	05/06/09	3h/día	Definición de los requisitos del sistema.
Especificación de requisitos	08/06/09	15/06/09			08/06/09	15/06/09	3h/día	Análisis de los requisitos y definición del modelo conceptual.
Diseño	17/06/09	30/06/09			17/06/09	30/06/09	2h/día	Diseño del sistema: arquitectura.
Implementación	01/07/09	30/07/09			01/07/09	30/07/09	5h/día	Programación de la solución
Pruebas	3/08/09	7/08/09			3/08/09	7/08/09	1h/día	Fase de Pruebas.
Memoria								Redacción de la memoria del proyecto.
Introducción	29/06/09	3/07/09			29/06/09	3/07/09	3h/día	Introducción al proyecto: Objetivos.
Estado de la cuestión	6/07/09	15/07/09			06/07/09	08/07/09		Redacción del trabajo realizado en el estudio preliminar.
Planteamiento del problema.	16/07/09	20/07/09	20/07/09	24/07/09	20/07/09	24/07/09	3h/día	Extensión del trabajo realizado en el estudio preliminar.
Gestión del proyecto	20/07/09	20/07/09			20/07/09	20/07/09	3h/día	Estimación del esfuerzo del proyecto.
Solución: Descripción/Fases de desarrollo.	28/07/09	21/08/09	28/07/09	24/08/09	28/07/09	24/08/09	4h/día	Explicar la solución desarrollada.
Pruebas	26/08/09	04/09/09			26/08/09	04/09/09	4h/día	La evaluación dependerá del tipo de solución.
Conclusiones	07/09/09	07/09/09			07/09/09	07/07/09	3h/día	Conclusiones personales sobre la solución desarrollada.
Anexos	08/09/09	11/09/09			08/09/09	11/09/09	3h/día	Documentos que constan como anexos del proyecto.
Presentación								Elaboración de la presentación

Figura 3.1: Planificación del proyecto.

Se comentan las tareas contempladas en la planificación:

1. **Estudio preliminar:**
   a. Estado de la cuestión: se alcanza una visión preliminar del proyecto a desarrollar.
   b. Proyecto dirigido: se realizan una serie de escenarios desarrollados en la herramienta Greenfoot que proveen a la autora del proyecto, de unos conocimientos preliminares para una mejor implementación de la solución propuesta.
   c. Planteamiento del problema: mediante la petición inicial del cliente, se plantea el problema a resolver.
   d. Esbozo de la solución: Se plantea una solución a rasgos generales.

2. **Desarrollo:**
   a. Definición de requisitos: se definen los requisitos de usuario y del sistema, estos requisitos son extraídos a partir de las reuniones del cliente y de la documentación existente de las prácticas de la asignatura.

   b. Especificación de requisitos: se realiza un análisis detallado de requisitos y una definición del modelo conceptual a partir de la respuesta del cliente.

   c. Diseño: Se estudian las distintas maneras para desarrollar la aplicación, se mejoran los prototipos a mostrar y se realizan los diagramas necesarios.

   d. Programación de la solución: cuando se ha definido el diseño y se sabe la arquitectura que se va a desarrollar, pasamos a la fase de implementación de la solución.

   e. Pruebas: Para este sistema se han usado pruebas basadas en el cumplimiento de los casos de uso, por lo tanto en esta fase se van verificando los requisitos del sistema a cumplir y se realizan los correspondientes diagramas.

3. **Memoria:** en esta fase se desarrolla el documento actual. Como se puede observar hay replanificaciones ya que se presentan modificaciones que obligan a actualizaciones en la memoria. Los capítulos de la memoria se engloban en los siguientes apartados:

   a. Introducción: Se plantea la introducción al proyecto, los objetivos que se desean cumplir y la estructura del documento.

b. Estado de la cuestión: el estudio realizado en el estudio preliminar es plasmado en este capítulo.

c. Gestión del proyecto: Se describe el esfuerzo estimado del proyecto.

d. Planteamiento del problema: Aquí se describe el problema que se presenta y al que hay que darle solución.

d. Solución: Se plantea la solución que se ha escogido justificadamente, así como las fases de análisis y diseño de la solución y descripción de la implementación de esta solución.

e. Pruebas: Se describe el plan de pruebas a seguir y se plasman las pruebas realizadas, así como los objetivos que se persiguen con ellas.

g. Conclusiones: Se plasman las conclusiones del autor en cuánto al aporte que le ha supuesto el proyecto y lo que ha supuesto su realización.

h. Anexos: Capítulo que engloba los anexos del proyecto.

3.2 Diagrama Gantt

El diagrama Gantt asociado a las tareas descritas, se incluye en el capítulo de Anexos del proyecto. Este diagrama Gantt recoge la duración real del proyecto. Las tareas asociadas se corresponden con las descritas en la figura *Figura 3.1: Planificación del proyecto* y se engloban en tres grupos:

1. Estudio preliminar, definido por las subtareas:
    a. Estado de la cuestión
    b. Proyecto dirigido.
    c. Planteamiento del problema
    d. Esbozo de la solución

2. Desarrollo, está definido por las subtareas:
    a. Definición de requisitos
    b. Especificación de requisitos
    c. Diseño
    d. Implementación
    e. Pruebas

3. Memoria, definida por las subtareas:
    a. Introducción
    b. Estado de la cuestión
    c. Gestión del proyecto
    d. Planteamiento del problema
    e. Solución
    f. Plan de pruebas

g. Conclusiones
h. Anexos

Diagrama de Gantt

	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	**ESTUDIO PRELIMINAR**	**168 días?**	**lun 06/10/08**	**mié 27/05/09**	
2	Inicio de proyecto	1 día?	lun 06/10/08	lun 06/10/08	
3	Estado de la cuestión	13 días?	mar 07/10/08	jue 23/10/08	2
4	Proyecto dirigido	133 días?	lun 03/11/08	mié 06/05/09	3
5	Planteamiento del problema	1 día?	lun 18/05/09	lun 18/05/09	4
6	Esbozo de la solución	6 días?	mié 20/05/09	mié 27/05/09	5
7	Fin del estudio preliminar	0 días	mié 27/05/09	mié 27/05/09	6
8	**DESARROLLO**	**50 días?**	**lun 01/06/09**	**vie 07/08/09**	
9	Definición de requisitos	5 días?	lun 01/06/09	vie 05/06/09	7
10	Especificación de requisitos	6 días?	lun 08/06/09	lun 15/06/09	9
11	Diseño	10 días?	mié 17/06/09	mar 30/06/09	10
12	Implementación	22 días?	mié 01/07/09	jue 30/07/09	11
13	Fin de implementación	0 días	jue 30/07/09	jue 30/07/09	12
14	Pruebas	5 días?	lun 03/08/09	vie 07/08/09	13
15	**MEMORIA**	**53 días?**	**mié 01/07/09**	**vie 11/09/09**	
16	Introducción	5 días?	mié 01/07/09	mar 07/07/09	12CC
17	Estado de la cuestión	8 días?	mié 08/07/09	vie 17/07/09	16
18	Gestión del proyecto	1 día?	lun 20/07/09	lun 20/07/09	17
19	Planteamiento del problema	3 días?	mar 21/07/09	jue 23/07/09	18
20	Solución	19 días?	mar 28/07/09	vie 21/08/09	19
21	Plan de Pruebas	8 días?	mié 26/08/09	vie 04/09/09	19;14;20
22	Conclusiones	1 día?	lun 07/09/09	lun 07/09/09	21
23	Anéxos	1 día?	vie 11/09/09	vie 11/09/09	22
24	Fin de la memoria	0 días	vie 11/09/09	vie 11/09/09	23

Figura 3.2: Tareas Gantt.

3.3 Análisis económico

Para estimar el coste de la aplicación desarrollada se han utilizado los criterios que influyen en cualquier proyecto de consultoría, como es el tiempo de desarrollo, los costes derivados del consumo de recursos y la infraestructura utilizada para el desarrollo. También hemos de tener en cuenta aspectos como la formación necesaria y mantenimiento de la aplicación.

El resumen del presupuesto que vamos a mostrar a continuación, es un **presupuesto ficticio** ya que estima que el proyecto va a ser realizado por analistas, jefes de proyecto, etc. Pero en realidad, al ser un proyecto académico ha sido realizado por la autora del presente documento. La principal diferencia con el presupuesto que mostraremos y el real es el coste de las horas realizadas por los empleados, más adelante se comentará este aspecto.

Mostraremos la estimación de costes de la aplicación teniendo en cuenta los costes directos de personal, costes directos materiales y los costes indirectos del proyecto.

3.3.1 Desglose por actividades del proyecto

Se han puesto las horas totales de cada una de las fases del proyecto teniendo en cuenta el tiempo empleado en su desarrollo. Esta tabla se basa en las tareas que componen el diagrama Gantt y en la tabla sobre la planificación del proyecto.

DURACIÓN PROYECTO	
Actividades	**Tiempo real**
Planificación	26h (Estado de la cuestión = 13 días*2h/día)
Estudio Viabilidad Sistema	15h (Planteamiento del problema+Esbozo de la solución=((1 día*3h/día) + (6 días*2h/día))
Análisis y diseño	53 h (Definición + Especificación de requisitos+ diseño=((5 días*3h/día) + (6 días*3h/día)+(10 días*2h/día))
Implementación	110 h Implementación=(22 días*5 h/día)
Documentación	159 h ((5 días*3h/día) + (8días*2h/día) +(3días*3h/día)+(19 días*4 h/día)+(8 días*4 h/día) + (1 día*5h/día)+(1día*3h/día)+(1 días*3 h/día))
Pruebas	5 h (5 días*1h/día)
Implantación	(Estimación)1 h
Total horas/Empleado	**369 h**

Figura 3.3.1: Tabla de duración de actividades del proyecto.

3.3.2 Salarios por categoría

Para conocer el gasto de la empresa en relación al trabajo de sus empleados o personal que conforma la misma es necesario conocer el salario de los trabajadores. Para saber los salarios hemos tomado como referencia una empresa que se dedica a desarrollo de proyectos software.

Los salarios medios se definen en la siguiente tabla:

Costes salarios de personal			
Cargo	**Sueldo Bruto/Año**	**Sueldo Bruto/Mes**	**Coste Hora/Empleado**
Jefe de proyecto	42.000 €	3.500 €	21,875 €
Analista	30.000 €	2.500 €	15,625 €
Programador	20.000 €	1666,66 €	11,45 €
Gestor de Pruebas	20.000 €	1666,66 €	11,45 €

Figura 3.3.2.1: Tabla de salarios puesto/hora.

Mostramos otra tabla con los porcentajes que van a influir en el coste total del proyecto. Esta tabla se toma como referencia de la misma empresa de la que proviene la tabla de costes de los salarios.

ACTIVIDAD	PORCENTAJE
Costes directos	-
Costes indirectos	10 %
Riesgo	15 %
Beneficios	40 %
IVA	16 %

Figura 3.3.2.2: Tabla de porcentajes.

A continuación, una vez conocidos los salarios, estimamos la dedicación de los empleados en cada una de las fases definidas en el proyecto. De esta manera, obtendremos el coste del personal por cada fase.

Coste salarios de personal/Fase					
Cargo / Fase	Jefe de proyecto	Analista	Programador	Gestor de pruebas	TOTAL
Planificación	26 h				568,65 € (26 * 21,875)
Estudio de viabilidad del sistema	15h				328,125 €
Análisis y diseño	26,5 (53h/2)	26,5 (53h/2)			1053,375 €
Implementación		36,667 (110h/3)	73,44 (110h - 110h/3)		1413,801875 €
Documentación	53 (159h/3)	106 (159-159h/3)			2815,625 €
Pruebas				5 h	57,25 €
Implantación		1 h			15,625 €
TOTAL	120,5*21,875= 2635,9375 1 €	170,167*15,625= 2658,859 €	73,44*11,45= 840,88 €	5*11,45=57,25€	**6236,826875 €**

Figura 3.3.2.3 Tabla de costes personal/fase del proyecto.

3.3.3 Gastos de personal imputables al proyecto

A partir de la tabla anterior es posible sacar la relación entre costes de los empleados del proyecto y las horas que van a dedicar:

Gastos horas/empleados		
Empleado	**Horas**	**Coste**
Jefe de proyecto	120,5h	120,5*21,875= 2635,9375 1 €
Analista	170,167h	170,167*15,625= 2658,859 €
Programador	73,44h	73,44*11,45= 840,88 €
Gestor de Pruebas	5h	5*11,45=57,25€
TOTAL	369,107 h	**6236,826875 €**

Figura 3.3.3: Tabla de horas/empleados.

3.3.4 Gastos directos materiales

Para la realización de este proyecto se han empleado los siguientes medios materiales:

Gastos materiales		
Material	**Cantidad**	**Coste**
Ordenador de Sobremesa PC Intel Pentium Core 2 DUO 2 Gb RAM, 160 GB de disco duro.	1	699 €
Sistema Operativo Windows XP Proffesional	1	505 €
TOTAL	369,107 h	**1204 €**

Figura 3.3.4: Tabla de gastos materiales del proyecto.

Además de estos materiales mostrados en la tabla necesitamos otras herramientas que son gratuitas y por eso no las mencionamos anteriormente:

- Licencias software: la herramienta Greenfoot y el entorno de desarrollo Java (JDK) son software de libre distribución (por lo tanto, el coste asociado es de 0 €).

3.3.5 Gastos indirectos

Los gastos indirectos suponen el 10% de los gastos totales en el proyecto, ya que tomamos como referencia los porcentajes que se indican en la *Figura 3.3.2.2: Tabla de porcentajes.*

En la siguiente tabla se especifican todos los componentes de los gastos indirectos y una estimación de los posibles costes:

Gastos indirectos	
Gasto	**Coste**
Limpieza de oficina	100 € (Incluido en los costes indirectos)
Electricidad	200 €(Incluido en los costes indirectos)
Agua	100 €(Incluido en los costes indirectos)
Amortizaciones	100 €(Incluido en los costes indirectos)
Alquiler Local	700 €(Incluido en los costes indirectos)
Teléfono y ADSL	200 €(Incluido en los costes indirectos)

Figura 3.3.5: Tabla de gastos indirectos.

3.3.6 Gastos directos

En esta tabla que se muestra a continuación se especifican todos los gastos obtenidos anteriormente para la realización del resumen del presupuesto.

Gastos directos	
Gasto	**Coste**
Empleados	6236,826875 €
Gastos materiales	1205 €
TOTAL	**7441,826875 €**

Figura 3.3.6: Tabla de gastos directos.

3.3.7 Resumen del presupuesto

El presupuesto final se elabora teniendo en cuenta el presupuesto del proyecto (gatos directos) junto con el beneficio de la empresa del 40%, los gastos indirectos (electricidad, ADSL, utilización del local…) y unos posibles riesgos del 15%. Al precio final se le debe añadir el 16% de IVA.

PRESUPUESTO			
Actividad	**Porcentaje**	**Subtotal**	**Total**
Costes directos	-	7441,826875 €	7441,826875 €
Costes Indirectos	10%	744,1826875 € (10% de 7441,826875)	8186,0095625 €
Riesgo	15%	1227,901434375 € (15 % de 8186,0095625)	9413,910996875 €
Beneficios	40%	3765,56439875 (40 % de 9413,910996875)	13179,475395625 €
IVA	16%	2108,7160633 (16 % de 13179,475395625)	15288,191458925 €
TOTAL			**15288,2 €**

Figura 3.3.7: Tabla de resumen de presupuesto del proyecto.

El presupuesto total asciende a un TOTAL de **15288,2 € IVA INC.**

3.3.8 Resumen del presupuesto real

El presupuesto que se muestra a continuación consiste en tomar que todas las tareas que intervienen en la elaboración del proyecto sólo las realiza una persona, que es la autora del proyecto y el coste por hora será el de un programador. Por lo tanto obtendremos un presupuesto más económico, ya que el coste por hora del programador es más bajo que el del analista o el del jefe de proyecto.

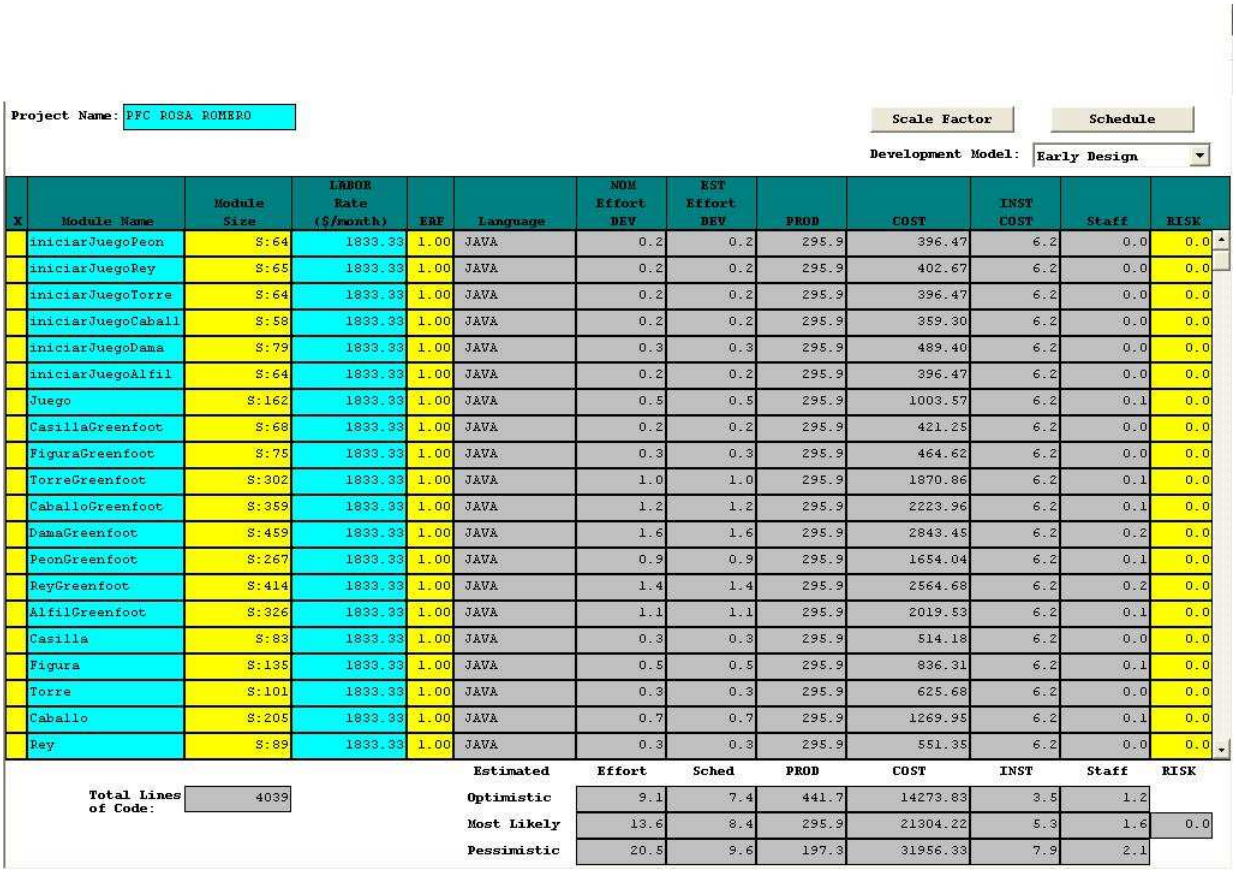
3.4 Esfuerzo del proyecto

A continuación se expone la estimación de las líneas de código codificadas en el desarrollo del proyecto fin de carrera.

Para realizar esta estimación se ha utilizado el modelo COCOMO II [15]. Este modelo permite estimaciones de coste, de esfuerzo y de tiempo cuando se planea el desarrollo de un nuevo software. COCOMO II es la ultima extensión del original COCOMO (COCOMO 81) que fue publicado en 1981.

Este modelo ofrece tres submodelos: modelo de Composición de Aplicaciones, modelo de Diseño Anticipado y modelo de Post-Arquitectura. Para la estimación de este proyecto se ha utilizado el modelo de Diseño Anticipado puesto que este modelo se utiliza para obtener estimaciones aproximadas del coste de un proyecto antes de que esté terminada por completo su arquitectura. Está basado en este caso, en KSLOC (líneas de código fuente). De esta manera se realiza una comparación entre lo estimado en fases tempranas de desarrollo de la solución y entre la solución final propuesta.

Para realizar esta estimación se utiliza una versión de la solución realizada en fases tempranas de implementación, posee módulos eliminados en la solución final y no posee algunos que si que están en la solución.



Project Name: PFC ROSA ROMERO

Scale Factor | Schedule

Development Model: Early Design

X	Module Name	Module Size	LABOR Rate ($/month)	ERF	Language	NOM Effort DEV	EST Effort DEV	PROD	COST	INST COST	Staff	RISK
	iniciarJuegoPeon	S:64	1833.33	1.00	JAVA	0.2	0.2	295.9	396.47	6.2	0.0	0.0
	iniciarJuegoRey	S:65	1833.33	1.00	JAVA	0.2	0.2	295.9	402.67	6.2	0.0	0.0
	iniciarJuegoTorre	S:64	1833.33	1.00	JAVA	0.2	0.2	295.9	396.47	6.2	0.0	0.0
	iniciarJuegoCaball	S:58	1833.33	1.00	JAVA	0.2	0.2	295.9	359.30	6.2	0.0	0.0
	iniciarJuegoDama	S:79	1833.33	1.00	JAVA	0.3	0.3	295.9	489.40	6.2	0.0	0.0
	iniciarJuegoAlfil	S:64	1833.33	1.00	JAVA	0.2	0.2	295.9	396.47	6.2	0.0	0.0
	Juego	S:162	1833.33	1.00	JAVA	0.5	0.5	295.9	1003.57	6.2	0.1	0.0
	CasillaGreenfoot	S:68	1833.33	1.00	JAVA	0.2	0.2	295.9	421.25	6.2	0.0	0.0
	FiguraGreenfoot	S:75	1833.33	1.00	JAVA	0.3	0.3	295.9	464.62	6.2	0.0	0.0
	TorreGreenfoot	S:302	1833.33	1.00	JAVA	1.0	1.0	295.9	1870.86	6.2	0.1	0.0
	CaballoGreenfoot	S:359	1833.33	1.00	JAVA	1.2	1.2	295.9	2223.96	6.2	0.1	0.0
	DamaGreenfoot	S:459	1833.33	1.00	JAVA	1.6	1.6	295.9	2843.45	6.2	0.2	0.0
	PeonGreenfoot	S:267	1833.33	1.00	JAVA	0.9	0.9	295.9	1654.04	6.2	0.1	0.0
	ReyGreenfoot	S:414	1833.33	1.00	JAVA	1.4	1.4	295.9	2564.68	6.2	0.2	0.0
	AlfilGreenfoot	S:326	1833.33	1.00	JAVA	1.1	1.1	295.9	2019.53	6.2	0.1	0.0
	Casilla	S:83	1833.33	1.00	JAVA	0.3	0.3	295.9	514.18	6.2	0.0	0.0
	Figura	S:135	1833.33	1.00	JAVA	0.5	0.5	295.9	836.31	6.2	0.1	0.0
	Torre	S:101	1833.33	1.00	JAVA	0.3	0.3	295.9	625.68	6.2	0.0	0.0
	Caballo	S:205	1833.33	1.00	JAVA	0.7	0.7	295.9	1269.95	6.2	0.1	0.0
	Rey	S:89	1833.33	1.00	JAVA	0.3	0.3	295.9	551.35	6.2	0.0	0.0

	Estimated	Effort	Sched	PROD	COST	INST	Staff	RISK
Total Lines of Code: 4039	Optimistic	9.1	7.4	441.7	14273.83	3.5	1.2	
	Most Likely	13.6	8.4	295.9	21304.22	5.3	1.6	0.0
	Pessimistic	20.5	9.6	197.3	31956.33	7.9	2.1	

Figura 3.4: Estimación COCOMO II del proyecto.

Como se observa en la figura anterior, se tiene un **total de líneas de código de 4039**. Observando la estimación del tiempo de duración más probable, se obtiene que el proyecto dura **8,4 meses** con un coste de **17506,75 dólares** (moneda usada en COCOMO II).

3.5 Conclusiones sobre el esfuerzo del proyecto

Tomando en consideración la estimación realizada, la duración del proyecto estimada en las fases tempranas de realización del proyecto es menor que la real, lo cual es bastante común en el desarrollo de proyectos reales debido a replanificaciones que siempre surgen.

En cuanto al coste estimado, el presupuesto realizado presentado en los apartados anteriores, se encuentra en el intervalo razonable estimado por COCOMO II. Por lo tanto se podría considerar que el esfuerzo del proyecto realizado es razonable.

4 Planteamiento del problema y la solución

En este capítulo se comenta la solución que se ha llevado a cabo para solventar la problemática que nos planteaba el cliente.

4.1 Contexto del problema

Para poner en situación al lector, comenzaremos enunciando la problemática que se presenta y a continuación seguiremos con la solución propuesta.

Los estudiantes de la asignatura de Programación desarrollan sus prácticas en el entorno de desarrollo jGrasp [16]. Sería necesario por tanto, introducir superficialmente algunos aspectos de esta herramienta.

jGrasp es un entorno de desarrollo integrado, creado específicamente para aportar visualizaciones que mejoren la comprensión del software. jGrasp está implementado en Java y se ejecuta en todas las plataformas con una Máquina Virtual de Java (JVM, en inglés *Java Virtual Machine*). Está configurado para trabajar con muchos compiladores proveyendo de un modo "*señala y clickea*" de compilación y ejecución. Es el último IDE de GRASP (acrónimo de *Graphical Representations of Algorithms, Structures and Processes*), grupo de investigación de la Universidad de Auburn.

jGrasp actualmente permite la generación automática de tres importantes visualizaciones software:

-**Estructuras de control**: para visualización de código.
-**Diagramas de clase UML**: para la visualización de la arquitectura.
-**Visores**: provee vistas dinámicas para primitivas y objetos, incluyendo las estructuras tradicionales como listas enlazadas y árboles binarios.

También aporta un innovador banco de objetos y un depurador que están estrechamente integrados con estas visualizaciones. [17]

Tras una breve descripción de la herramienta, continuamos con la descripción del problema.

A lo largo de los años se ha ido observando que estos estudiantes realizan sus prácticas mecánicamente. Toman los enunciados de tales prácticas y copian el código que se les ofrece, sin entender realmente conceptos clave de la POO, como la diferencia entre clase y objeto. También se observa la presencia de una programación que se lleva a cabo mediante ensayo y error.

El estudiante observa en la salida por pantalla del entorno de desarrollo, que se ejecuta el programa, creándose una serie de objetos e invocándose una serie de métodos. Pero los estudiantes, siguen sin poder entender que han definido una clase de la que se instancian objetos. Para ellos son simples líneas de texto que aparecen como resultado.

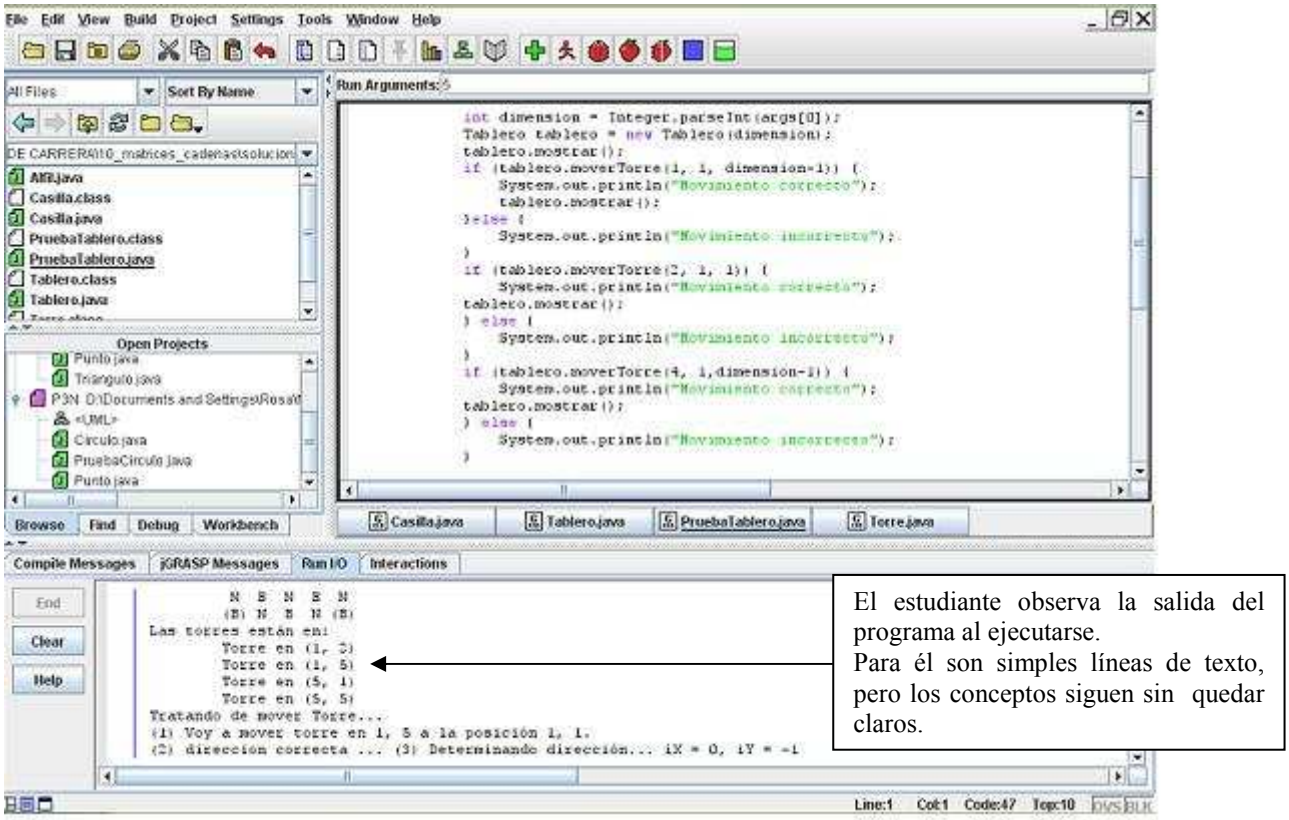


Figura 4.1: Vista de la herramienta jGrasp.

Con el problema que se nos presenta, el cliente, en este caso, el equipo docente de la asignatura de Programación de Ingeniería de Telecomunicación, realiza una petición de requisitos que debe poseer el mecanismo a ofrecer.

La práctica a adaptar al nuevo mecanismo será la correspondiente a la creación de un tablero de ajedrez e implementación del movimiento de las figuras de ajedrez. El cliente requiere que los estudiantes sigan utilizando la herramienta jGrasp para la definición de clases necesarias para el desarrollo de la práctica. Solicita, por tanto, que el mecanismo de solución permita al estudiante la utilización de su código previamente desarrollado y que éste sea fácilmente adaptable a la nueva situación.

El estudiante debe poder visualizar los objetos creados, pertenecientes a las clases definidas, así como poder interactuar con ellos e inspeccionar su estado, todo esto siempre de manera visual.

4.2 Definición de la solución

En el estado de la cuestión se presentaban algunas de las herramientas existentes de enseñanza de POO.

Para la elección de la herramienta adecuada hacia el desarrollo de la aplicación, se realizó un estudio comparativo, en base a los objetivos de la solución propuesta y requisitos del cliente, de las herramientas citadas anteriormente: **BlueJ, Jeliot 3, Greenfoot y Alice.**

4.2.1 Estudio comparativo para la elección de la herramienta adecuada.

A continuación se presentan una serie de tablas en las que se comparan los requisitos necesarios que debe poseer la herramienta que dé soporte a la solución y las posibilidades que ofrece cada herramienta. Los requisitos que se exponen en las tablas se extraen de las reuniones con el cliente y son los elegidos para escoger la herramienta adecuada.

- **Usabilidad de la herramienta:** este requisito se define como la facilidad de uso y aprendizaje rápido de la herramienta. No se pretende formar al usuario en el uso de la herramienta, ni perder el tiempo en enseñar al usuario a desenvolverse en la misma. También se pretende que la herramienta sea de fácil acceso e instalación.

- **Visualización de clases y objetos:** la herramienta debe poseer una interfaz gráfica atractiva que permita una visualización de las clases que componen el sistema y las instancias que se crean de esas clases. De esta manera se trabaja sobre el concepto de diferenciación entre clase y objeto.

- **Creación e interacción con los objetos:** como se citaba anteriormente, la herramienta debe permitir la creación de objetos que se visualicen y de los cuáles se puedan invocar sus métodos y se observen los cambios que se producen tras realizarse una acción.

- **Fácil adaptación del código desarrollado por el estudiante:** la herramienta puede poseer restricciones en cuanto al desarrollo de los escenarios, pero debe tener la opción de poder integrar código puramente Java para colaborar en la construcción del escenario.

Con estos requisitos propuestos se va a realizar una comparación entre herramientas y la elegida debe ser la que mejor se adapte a las necesidades propuestas.

- **BlueJ:**

BlueJ	
Requisitos necesarios	**Posibilidades de la herramienta**
Usabilidad de la herramienta	• La herramienta despliega una interfaz de usuario muy intuitiva, nada más abrir un programa. Como ejemplo se puede nombrar que la creación de un objeto se realiza de la primera manera que se le puede ocurrir al usuario, pulsando el ratón sobre la clase que desea. • Estudios iníciales acerca de la herramienta remarcaban dificultades en la instalación de la herramienta, pero en sus versiones posteriores no se remarcó esta dificultad. • Es muy accesible, libre de cargo para uso educativo y solo conlleva una descarga de internet de pequeño volumen.
Visualización de clases y objetos.	• La herramienta provee de un diagrama de clases que representa las clases que componen el sistema y la relación entre ellas. • La herramienta provee de una excelente visualización de clases e instancias y permite la inspección del estado de las instancias de estas clases.
Creación e interacción con los objetos.	La herramienta permite la creación y manipulación de instancias de clases que los estudiantes desarrollan o clases provistas por la API de Java.
Fácil adaptación del código desarrollado por el estudiante.	La herramienta permite la total adaptación de un código Java desarrollado fuera del entorno BlueJ. [18]

Figura 4.2.1.1: Tabla de posibilidades que ofrece la herramienta BlueJ.

A continuación, se ofrece una imagen de la herramienta, para que se puedan observar las características visualmente:

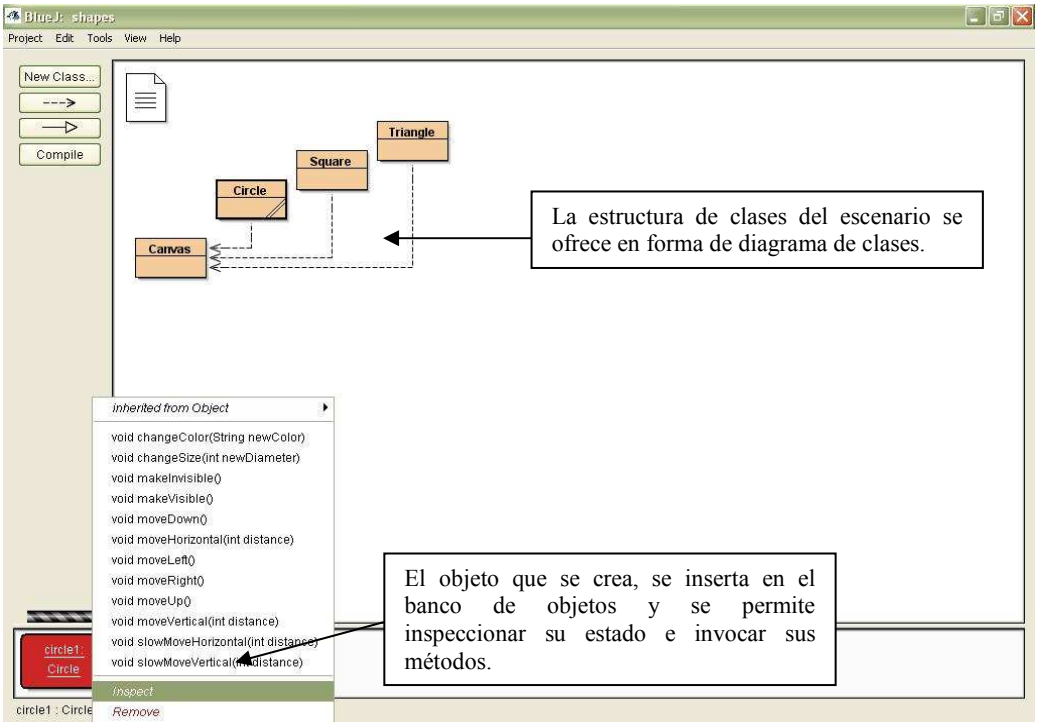


Figura 4.2.1.2: Vista de la interfaz gráfica que ofrece BlueJ.

- **Jeliot 3:**

Jeliot 3	
Requisitos necesarios	**Posibilidades de la herramienta**
Usabilidad de la herramienta	• La herramienta despliega una interfaz de usuario atractiva, con sus partes bien definidas, el código se encuentra a la izquierda del lienzo de representación, para que se puedan seguir las instrucciones del programa fácilmente. • Fácil instalación de la herramienta. • Es muy accesible, libre de cargo para uso educativo y solo conlleva una descarga de internet de pequeño volumen.
Visualización de clases y objetos.	• La herramienta no provee de una estructura de visualización de las clases que componen el sistema. • La herramienta va mostrando la creación de las variables y llamadas a métodos que se producen en la ejecución de un programa. Necesita que la clase creada contenga el método *public static void main (String[] args)*.
Creación e interacción con los objetos.	No se observa visualmente la creación de la instancia de una clase y no se puede interactuar con los objetos ni inspeccionar su estado.
Fácil adaptación del código desarrollado por el estudiante.	La herramienta permite la total adaptación de un código Java desarrollado fuera del entorno Jeliot 3.

Figura 4.2.1.3: Tabla de posibilidades que ofrece la herramienta Jeliot 3.

Como en la herramienta BlueJ, se ofrece una imagen de la herramienta, para que se puedan observar las características visualmente:

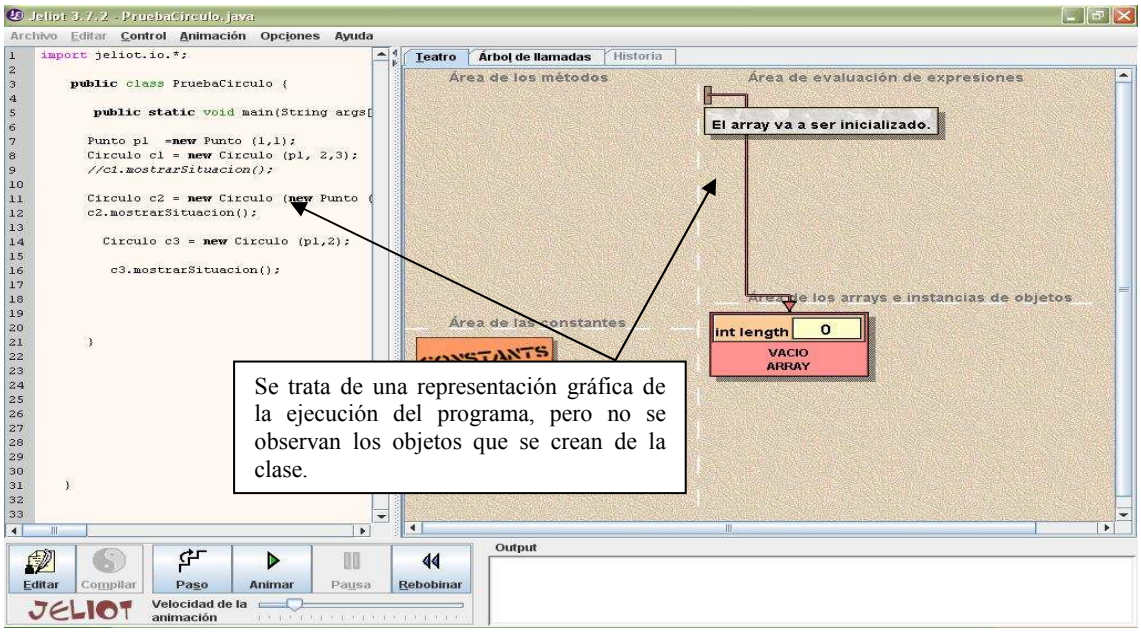


Figura 4.2.1.4: Vista de la interfaz gráfica ofrecida por Jeliot 3.

- **Alice:**

Alice	
Requisitos necesarios	**Posibilidades de la herramienta**
Usabilidad de la herramienta	• Se necesitan unas nociones básicas para comenzar a usar esta herramienta, la interfaz gráfica de usuario posee numerosas secciones que al principio del uso de esta herramienta pueden confundir al mismo. • Fácil instalación de la herramienta. • Es muy accesible, libre de cargo para uso educativo y solo conlleva una descarga de internet de pequeño volumen.
Visualización de clases y objetos.	• La herramienta provee de una estructura de visualización de las clases que componen el sistema. • Los objetos se pueden visualizar en un entorno 3D que se corresponde con el mundo en el que "viven".
Creación e interacción con los objetos.	La herramienta se centra en afirmar el concepto de "Programa", se comparan las instrucciones de un programa con las secuencias de un guión gráfico, con el objetivo de servir de guía para entender la historia.
Fácil adaptación del código desarrollado por el estudiante.	La herramienta posee su propia sintáxis en pseudocódigo. El programa debe ser desarrollado en el editor de la herramienta para su visualización.

Figura 4.2.1.5: Tabla de posibilidades que ofrece la herramienta Alice.

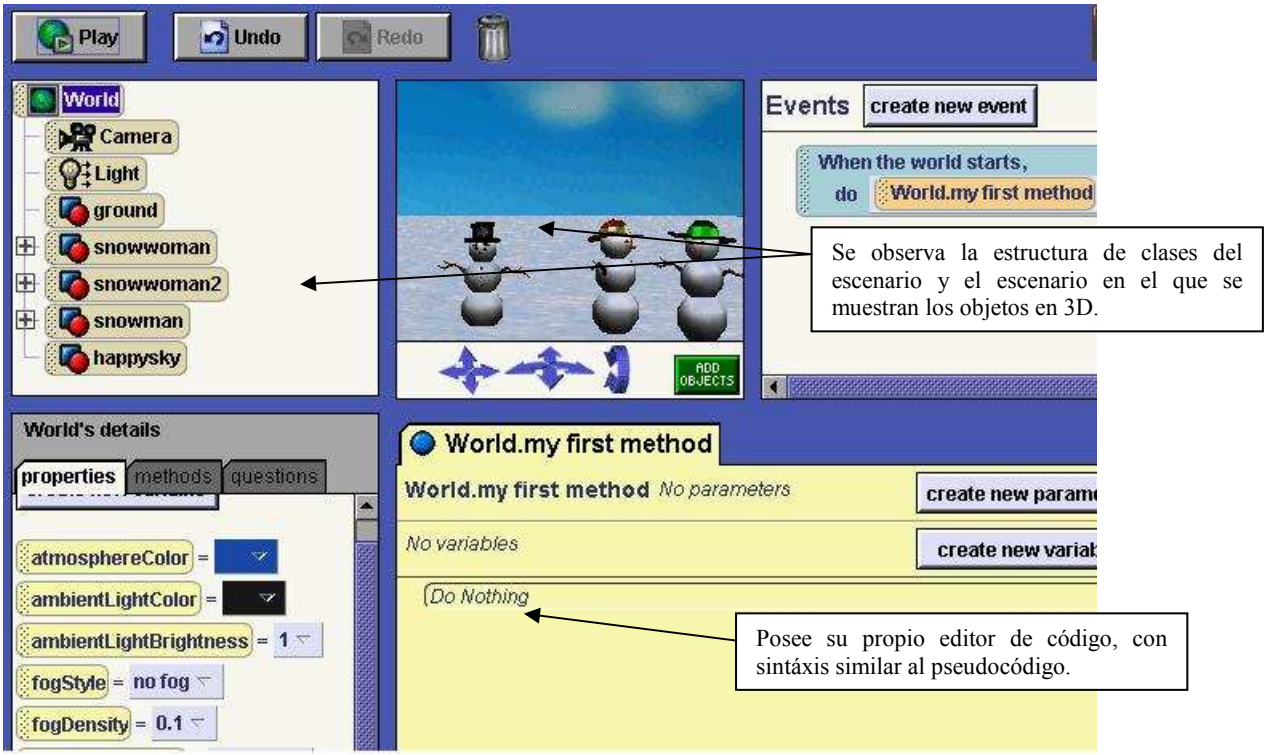


Figura 4.2.1.6: Imagen de las partes de la interfaz de usuario de la herramienta Alice.

- **Greenfoot:**

Greenfoot	
Requisitos necesarios	**Posibilidades de la herramienta**
Usabilidad de la herramienta	• Fácil instalación de la herramienta. • Es muy accesible, libre de cargo para uso educativo y solo conlleva una descarga de internet de pequeño volumen.
Visualización de clases y objetos.	• La herramienta provee de una estructura de visualización de las clases que componen el sistema. • Se permite asignar a los objetos que se crean, imágenes que resultan intuitivas para el usuario. El escenario desarrollado posee un aspecto cercano a la realidad de lo que se pretende desarrollar. Ej. Si se tiene una clase denominada FiguradeAjedrez, a esa clase se le puede asignar la imagen de una figura de ajedrez.
Creación e interacción con los objetos.	Se permite la creación e interacción con los objetos, inspeccionando su estado e invocando sus métodos.
Fácil adaptación del código desarrollado por el estudiante.	La herramienta permite la portabilidad de clases desarrolladas en otros entornos.

Figura 4.2.1.7: Tabla de posibilidades que ofrece la herramienta Greenfoot.

Se muestra una imagen de la herramienta Greenfoot para poder observar sus características:

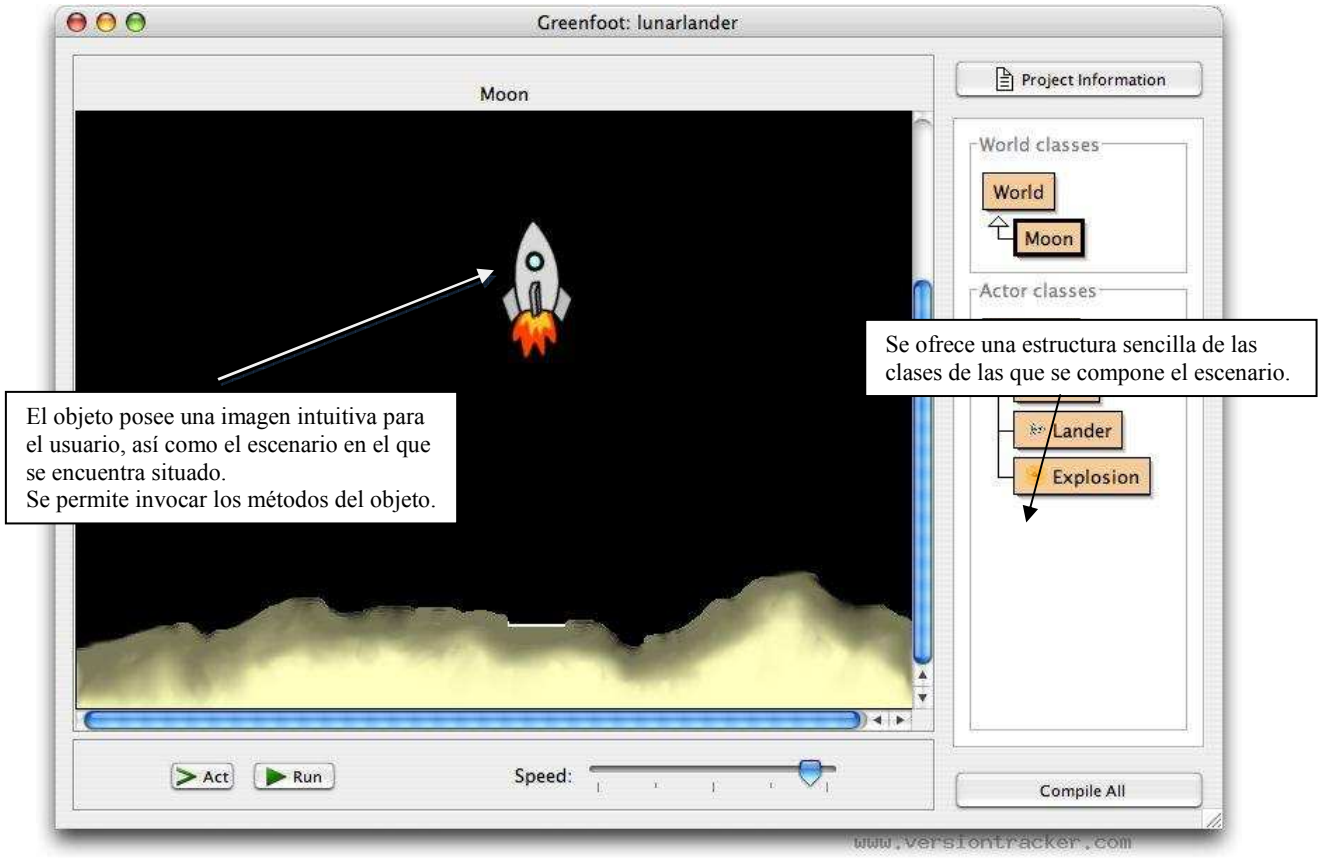


Figura 4.2.1.8: Interfaz gráfica que ofrece la herramienta Greenfoot.

4.2.2 Elección de la herramienta soporte, para el desarrollo de la solución.

Después de realizar el estudio comparativo de estas tres herramientas, la decisión de utilizar la herramienta Greenfoot, como herramienta soporte para el desarrollo de nuestra solución, está basada en las siguientes justificaciones:

- **La usabilidad del sistema es muy alta.** Es una herramienta de fácil instalación y está libre de cargo para uso educativo, la descarga de la herramienta se facilita en la página de la misma. Aunque la herramienta es distribuida en inglés, se trata de un vocabulario básico, que no impide a un usuario con bajo nivel de este idioma, disfrutar del uso de la herramienta.

- **El código fuente está perfectamente vinculado a la visualización de las instancias de clases.** Esta visualización ayudará a los estudiantes a establecer la relación entre el código fuente que ellos han escrito y el resultado que se produce. Se ayuda al entendimiento de conceptos abstractos como son el estado de un objeto y su identidad. Estudiantes que se inician en la POO pueden crear una representación visual de una clase, instanciar la clase, inspeccionar su estado e

invocar sus métodos, sin tener para ello, que escribir absolutamente nada de código.

Esta justificación anterior, sería válida también para la herramienta BlueJ, pero el problema observado en esta herramienta es que la estructura de clases que ofrece el sistema, se corresponde con diagramas UML. Estos diagramas describen la relación entre las clases pero no se corresponden con los objetivos fijados para un estudiante que se inicia en la POO. Por ello, se pretende simplificar lo máximo posible la experiencia del alumno en su introducción a la POO y se considera más adecuada la representación que ofrece Greenfoot.

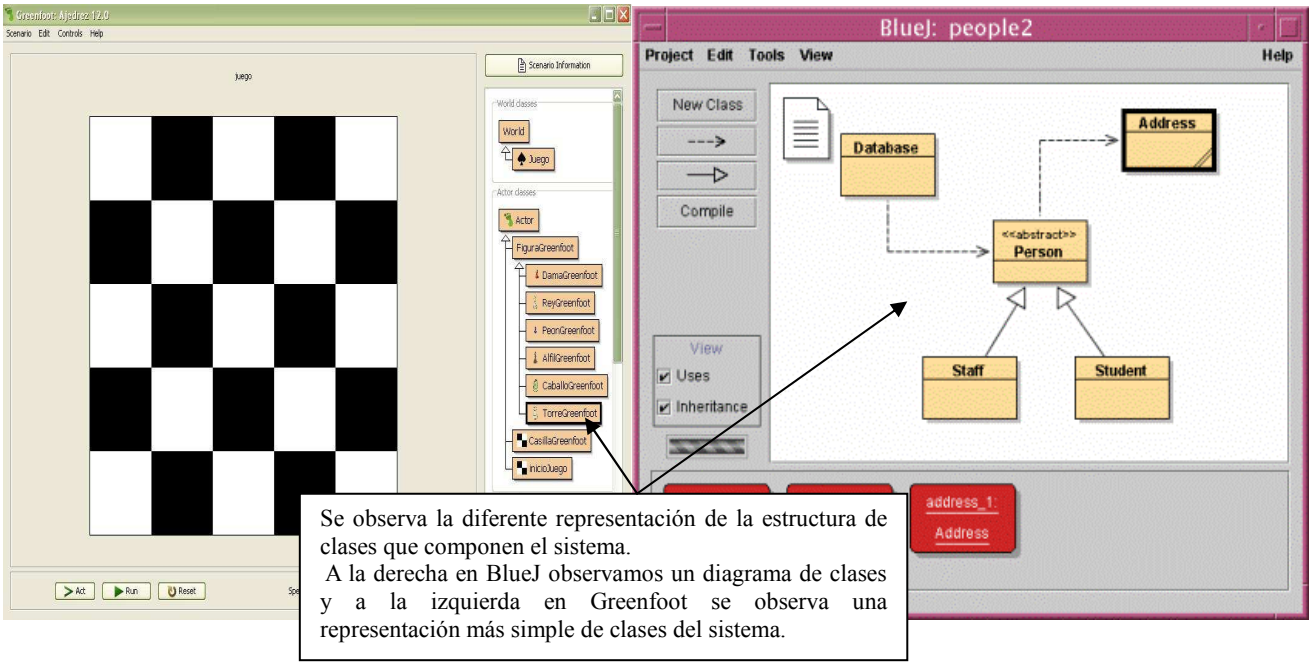


Figura 4.2.2: Comparativa de interfaces entre BlueJ y Greenfoot.

- **La visualización de las instancias de clases de un escenario desarrollado, es muy intuitiva**. Greenfoot permite asignar una imagen a las clases que componen el sistema, por lo tanto al instanciar el objeto de la clase se observa una imagen muy cercana a la realidad. Los objetos se sitúan en el "mundo" al que pertenecen. Se permite crear un programa con una apariencia atractiva para el usuario.

Esta última característica justifica también la elección de Greenfoot como herramienta soporte, sitúa a los objetos en un escenario en el que se interactúa con ellos, mientras que BlueJ, los sitúa en un banco de objetos sin distinción entre clases. Esta disposición de los objetos podría confundir a los estudiantes sobre el contexto en el que son creados los objetos.

- Como última justificación para la elección de la herramienta Greenfoot, se afirma que esta herramienta **ofrece una total portabilidad de código desde otra herramienta hasta Greenfoot**. El usuario puede desarrollar su código Java y utilizarlo después como clases de apoyo para la creación de programas en esta herramienta.

4.2.3 Planteamiento de la solución

Con todos los argumentos anteriores, se justifica la elección de la herramienta Greenfoot para el desarrollo de la solución. Una vez elegido el soporte en el que se desarrollará la aplicación requerida, nos centramos ahora en *cómo* se desarrollará esa solución.

Como se citaba anteriormente, el cliente solicitaba como requisito básico que los estudiantes pudieran utilizar su código y se realizará la portabilidad al nuevo escenario, por lo tanto en la solución planteada, las clases de los estudiantes son completamente independientes de las clases necesarias en Greenfoot para que la visualización de los objetos sea posible. Las clases desarrolladas por el alumno servirán de apoyo para las clases del juego, que utilizan la API propia de Greenfoot y que invocan métodos de estas primeras clases.

Para proveer de una mayor variedad de escenarios en la solución, se presentan seis modos de juego, tantos como figuras distintas de ajedrez existen. Con esto se pretende dejar a elección del personal docente que tipo de figura y movimiento de ésta, se desarrolla para las prácticas de la asignatura.

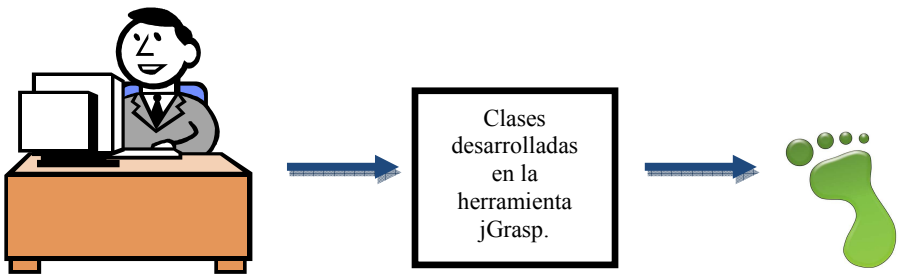


Figura 4.2.3: Figura que representa la solución planteada.

Las peculiaridades de implementación se incluirán en el capítulo *7. Implementación.*

5 Análisis

El propósito de la definición de requisitos es especificar lo que desea el usuario y producir un conjunto de requisitos software tan completo, constante y correcto como sea posible.

Los requisitos definen "*qué*" debe hacer el producto. Son una referencia para verificar el diseño y el producto. Los aspectos relativos al "*cómo*" no se incluyen, excepto aquellos que restringen las capacidades del software.

5.1 Definición de requisitos

El personal docente de la asignatura de Programación de la Universidad Carlos III de Madrid nos plantea la necesidad de obtener un método más adecuado que el actual para que los alumnos sean capaces de visualizar lo que realizan en las prácticas.

Para abordar la petición, se realizan reuniones con el cliente, que en este caso es la tutora del proyecto. En estas reuniones se acuerda la herramienta a utilizar para desarrollar la aplicación y como documentos base para la especificación de requisitos, se toman los enunciados de las prácticas de la asignatura.

La práctica que se acuerda adaptar para la solución, se trata de la correspondiente a la implementación de un tablero de ajedrez y el movimiento de sus piezas. El tablero de ajedrez se crea añadiendo casillas blancas y negras de manera alternada y no se permiten las casillas de igual color, ni horizontal ni verticalmente. Las piezas son creadas y añadidas al tablero y se solicitan ciertos movimientos sobre ellas. Se trata de comprobar la validez de estos movimientos. La herencia entre clases no se considera para esta práctica, al exceder los objetivos de la asignatura.

Para que la recogida de requisitos se realice de forma clara, sencilla y estructurada se ha definido una plantilla con las siguientes características:

IDENTIFICADOR:		
Nombre:		
Descripción:		
Prioridad: □ Alta □ Media □ Baja		**Necesidad:** □ Esencial □ Opcional □ Conveniente
Estabilidad: □ Si □ No		
Prerrequisito:		
Fuente:		

Donde:

- **Identificador:** Identifica de forma unívoca cada uno de los requisitos. Dicho identificador sigue la siguiente nomenclatura.

RS-Tipo-Número

Dónde:

- Tipo: puede tomar los valores:
    - un requisito del Sistema puede tomar los siguientes valores:
        - U de usuario.
        - F de funcional.
        - NF de no funcional.
        - D de dominio.
        - U de usabilidad.

- Número: será un número de tres cifras que empezará por 001 y se irá incrementando en una unidad por cada nuevo requisito añadido.

Ej.: RSF001 (Requisito Funcional 001)

- **Nombre:** Expresa en pocas palabras un resumen del requisito.

- **Descripción:** Breve comentario textual acerca de qué requisito se trata.

- **Identificador:** Indica la prioridad en el desarrollo del requisito. Puede tomar los valores de Alta, Media o Baja.

- **Necesidad:** Indica el nivel de necesidad del requisito dentro del sistema final. Puede tomar los valores de Esencial, Opcional o Conveniente. Un requisito será Esencial cuando el cliente no acepte ninguna negociación, Conveniente cuando el requisito se pueda negociar u Opcional cuando su implementación no es imprescindible.

- **Estabilidad:** indica la posibilidad de que el requisito cambie a lo largo del desarrollo de la aplicación. Puede tomar los valores "Si", o "No", cuando el requisito puede variar en función de las sucesivas etapas del proyecto.

- **Prerrequisito:** Indica si necesita algún requisito para poderse realizar.

- **Fuente**: Indica el origen a partir del cual se ha obtenido el documento. Cuando se trata de un documento externo se hace referencia al documento.

5.1.1 Requisitos funcionales

Se definen aquí los requisitos propios del sistema, los servicios que el sistema solución debe proporcionar.

IDENTIFICADOR: RSF001		
Nombre:	**Creación del objeto tablero de ajedrez.**	
Descripción:	Los usuarios pueden crear un objeto tablero de juego y visualizarlo.	
Prioridad: X Alta □ Media □ Baja	**Necesidad: X** Esencial □ Opcional □ Conveniente	
Estabilidad: X Si □ No		
Prerrequisito:	**Ninguno.**	
Fuente:	**Reuniones con el cliente.**	

IDENTIFICADOR: RSF002		
Nombre:	**Creación de un objeto que se corresponde con una figura de ajedrez.**	
Descripción:	Los usuarios pueden crear un objeto figura de ajedrez y visualizarla. Los objetos que pueden crear son los siguientes: -Torre. -Alfil. -Rey. -Dama. -Peón. -Caballo.	
Prioridad: X Alta □ Media □ Baja	**Necesidad: X** Esencial □ Opcional □ Conveniente	
Estabilidad: X Si □ No		
Prerrequisito:	**Ninguno.**	
Fuente:	**Reuniones con el cliente.**	

IDENTIFICADOR: RSF003		
Nombre:	**Movimiento de una figura.**	
Descripción:	Los usuarios pueden invocar el movimiento de una figura y observar los cambios que se producen.	
Prioridad: X Alta □ Media □ Baja	**Necesidad: X** Esencial □ Opcional □ Conveniente	
Estabilidad: X Si □ No		
Prerrequisito:	**La figura de ajedrez debe existir.**	
Fuente:	**Reuniones con el cliente.**	

5.1.2 Requisitos no funcionales

Se definen aquí, las restricciones propias del sistema.

IDENTIFICADOR: RSNF005		
Nombre:	**Instalación del JDK.**	
Descripción:	Para poder utilizar la herramienta Greenfoot, previamente se tiene que tener instalado el JDK. Mínima versión que se pide, JDK 5.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**Ninguno.**	
Fuente:	**Requisitos de la herramienta Greenfoot.[19]**	

IDENTIFICADOR: RSNF006		
Nombre:	**Instalación de Greenfoot.**	
Descripción:	El escenario está optimizado para funcionar con versión mínima de la herramienta: Greenfoot Versión 1.5.0.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**Ninguno.**	
Fuente:	**Desarrollador del sistema.**	

IDENTIFICADOR: RSNF007		
Nombre:	**Portabilidad de la aplicación.**	
Descripción:	Esta aplicación se puede implantar en los sistemas operativos Windows, MacOS y Linux • Greenfoot 1.5.3. • Al menos JDK 5.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**Ninguno.**	
Fuente:	**Documento de propuesta y requisitos la herramienta Greenfoot.**	

IDENTIFICADOR: RSD008		
Nombre:	**Utilización del API de la herramienta Greenfoot.**	
Descripción:	Todo desarrollador que desee implementar un escenario Greenfoot debe sujetarse a las restricciones impuestas por la herramienta de utilización de su API.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**Ninguno.**	
Fuente:	**Requisitos de la herramienta Greenfoot.**	

5.1.3 Requisitos del dominio

Se definen aquí los requisitos que reflejan las características del dominio del sistema solución, en este caso características de la herramienta Greenfoot y características de representación visual de la solución.

IDENTIFICADOR: RSD009		
Nombre:	**Eliminación del tablero de una figura de ajedrez.**	
Descripción:	Los usuarios pueden eliminar del tablero la figura de ajedrez que deseen.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**La figura a eliminar debe existir.**	
Fuente:	**Tutorial de la herramienta Greenfoot.**	

IDENTIFICADOR: RSD010		
Nombre:	**Consulta de las clases de las que consta la aplicación.**	
Descripción:	Los usuarios pueden observar de manera visual que clases componen el sistema nada más acceder a él y tener acceso a su código.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**La clase a consultar debe existir.**	
Fuente:	**Tutorial de la herramienta Greenfoot.**	

IDENTIFICADOR: RSD011		
Nombre:	**Consulta del estado de la figura de manera visual.**	
Descripción:	Los usuarios pueden consultar de manera visual los atributos que posee un objeto situado en el tablero de ajedrez y los valores de esos atributos.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**La figura que se consulta debe existir.**	
Fuente:	**Tutorial de la herramienta Greenfoot.**	

DENTIFICADOR: RSD012		
Nombre:	**Consulta del estado del tablero de juego de manera visual.**	
Descripción:	Los usuarios pueden consultar de manera visual los atributos que posee un objeto tablero y los valores de esos atributos.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**El tablero de juego debe haber sido creado.**	
Fuente:	**Tutorial de la herramienta Greenfoot.**	

IDENTIFICADOR: RSD018	
Nombre:	**La aplicación muestra ayuda para los mensajes de error de compilación que se producen.**
Descripción:	Los usuarios cuando realizan operaciones y no son permitidas, deben estar informados mediante mensajes de error, que la herramienta no sólo muestra, también proporciona una interpretación para ellos.
Prioridad: □Alta **X** Media □ Baja	**Necesidad:** □ Esencial **X** Opcional □ Conveniente
Estabilidad: X Si □ No	
Prerrequisito:	**Ninguno.**
Fuente:	**Tutorial de la herramienta Greenfoot.**

IDENTIFICADOR: RSD019	
Nombre:	**La invocación de los métodos correspondientes a los objetos del escenario, provocan cambios reales.**
Descripción:	Los usuarios pueden invocar cualquier método dentro del escenario actual y observar visualmente que esa invocación produce un cambio real y observable.
Prioridad: □Alta **X** Media □ Baja	**Necesidad:** □ Esencial **X** Opcional □ Conveniente
Estabilidad: X Si □ No	
Prerrequisito:	**Ninguno.**
Fuente:	**Tutorial de la herramienta Greenfoot.**

IDENTIFICADOR: RSD020	
Nombre:	**Interfaz gráfica de la aplicación.**
Descripción:	La aplicación debe presentar la apariencia de un tablero de ajedrez con casillas blancas y negras. Las figuras deben poseer una imagen que represente la imagen de las piezas del ajedrez dependiendo de la pieza de la que se trate.
Prioridad: X Alta □ Media □ Baja	**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No	
Prerrequisito:	**Ninguno.**
Fuente:	**Reuniones con el cliente.**

IDENTIFICADOR: RSD021	
Nombre:	**Información al usuario acerca del modo de juego en el que está.**
Descripción:	Los usuarios pueden saber en qué modo de juego se encuentran, a través de un icono en el escenario de juego que informa de ello, ofreciendo dibujada la figura de ajedrez a la que corresponde el tipo de juego, en ese momento.
Prioridad: X Alta □ Media □ Baja	**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No	
Prerrequisito:	**Ninguno.**
Fuente:	**Reuniones con el cliente.**

5.1.4 Requisitos de usabilidad

Se definen aquí los requisitos de usabilidad de la solución, que se corresponden con unas especificaciones para que los usuarios del sistema puedan conseguir una experiencia satisfactoria en el uso del escenario.

IDENTIFICADOR: RSU022		
Nombre:	**El usuario debe poseer un conocimiento básico de inglés.**	
Descripción:	Los usuarios deben tener algunas nociones básicas de inglés para saber interpretar las posibilidades que Greenfoot ofrece en su uso.	
Prioridad: □Alta **X** Media □ Baja		**Necesidad:** □ Esencial **X** Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**Ninguno.**	
Fuente:	**Herramienta Greenfoot.**	

IDENTIFICADOR: RSU023		
Nombre:	**El usuario debe poseer conocimientos básicos sobre la sintáxis del lenguaje Java.**	
Descripción:	Los usuarios deben tener conocimientos básicos sobre la sintáxis del lenguaje Java que es el lenguaje sobre el que se basa la herramienta.	
Prioridad: X Alta □ Media □ Baja		**Necesidad: X** Esencial □ Opcional □ Conveniente
Estabilidad: X Si □ No		
Prerrequisito:	**Ninguno.**	
Fuente:	**Herramienta Greenfoot.**	

5.2 Diagramas de casos de uso.

Del análisis de requisitos, obtenemos los siguientes casos de uso con los que mostramos las acciones que pueden realizar los usuarios**. Sólo existe un tipo de usuarios**, que son los estudiantes que interactúan con la aplicación y la completan para ver el resultado de su trabajo de manera gráfica.

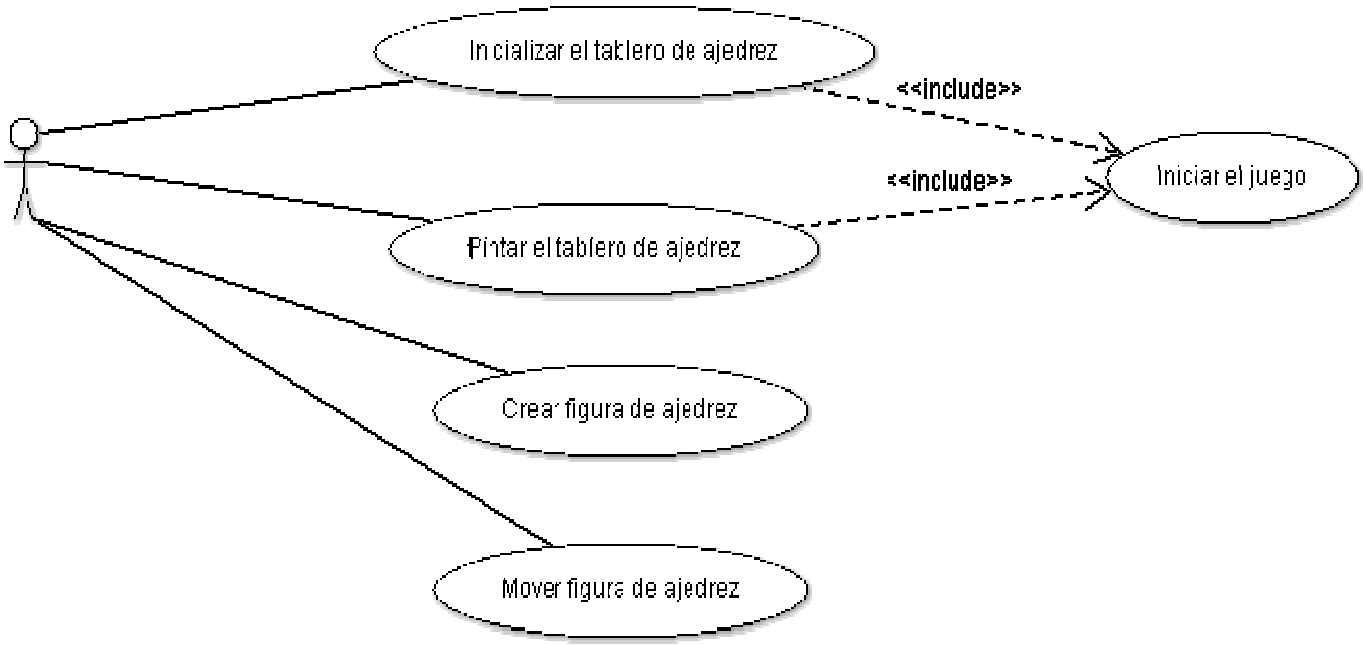


Figura 5.2: Diagrama de casos de uso del sistema.

5.2.1 Detalle de los casos de uso

En este apartado se presenta una explicación detallada de cada uno de los casos de uso mostrados en la figura anterior. Los casos de uso: *Inicializar el tablero de ajedrez* y *Pintar el tablero de ajedrez* están incluidos dentro del caso de uso *Iniciar el juego*, por lo tanto este no se detallará, al detallarse los dos anteriores.

Escenario " Inicializar el tablero de ajedrez"
Precondiciones:-
Postcondiciones: Pintar el tablero de ajedrez
Quién lo comienza: Usuario
Excepciones: - Si se introduce una dimensión de tablero mayor de 20x20 casillas, el sistema lanza una excepción. -El modo de juego al que se inicializa el juego es uno de los seis existentes.
Descripción: El usuario del escenario intenta inicializar el tablero de ajedrez con una dimensión de tablero de 6x6 casillas: El sistema comprueba que: • La dimensión introducida no excede de 20. • Modo de juego correcto dentro de los seis existentes. Tras estas comprobaciones, si resultan validas el sistema inicializa los atributos de esta clase y por lo tanto el tablero de juego queda inicializado. Se muestra por pantalla el mensaje que informa de este hecho. Si las comprobaciones no son válidas, el sistema no es inicializado, poseerá los valores por defecto.

Figura 5.2.1.1: Tabla descriptiva del caso de uso "Inicializar el tablero de ajedrez".

Escenario " Pintar el tablero de ajedrez"
Precondiciones: Inicializar el tablero de ajedrez
Postcondiciones: Crear figura de ajedrez
Quién lo comienza: Usuario
Excepciones: -
Descripción: El usuario del escenario intenta pintar el tablero de ajedrez tras ser éste inicializado. El sistema comprueba que: • El tablero se haya inicializado. Tras esta comprobación el sistema añade las casillas de manera alternada entre los colores, blanco y negro, como ya se había inicializado. La pantalla se refresca y ofrece el tablero de juego pintado adecuadamente.

Figura 5.2.1.2: Tabla descriptiva del caso de uso "Pintar el tablero de ajedrez".

Escenario " Crear figura de ajedrez"
Precondiciones: Pintar el tablero de ajedrez
Postcondiciones:
Quién lo comienza: Usuario
Excepciones: - Si se introducen coordenadas que exceden los límites del tablero, se lanza una excepción.
Descripción: El usuario del escenario intenta crear una figura de ajedrez para insertarla en el tablero de juego. El sistema comprueba que: • Si se introducen coordenadas que exceden los límites del tablero de ajedrez. Tras esta comprobación, si es válida el sistema añade la figura en la posición deseada por el usuario. Se refresca la pantalla y se nos ofrece la figura de ajedrez posicionada en el tablero. Si no es válida, la figura no es creada y se informa de este hecho mediante una excepción.

Figura 5.2.1.3: Tabla descriptiva del caso de uso "Crear figura de ajedrez".

Escenario "Mover figura de ajedrez"
Precondiciones: Crear figura de ajedrez
Postcondiciones:
Quién lo comienza: Usuario
Excepciones: - Si se introducen coordenadas de destino que exceden los límites del tablero, se lanza una excepción. El sistema lanza una excepción cuando se introduce el número de una figura, que no existe. El sistema también comprueba que la posición de destino se encuentre entre los posibles movimientos de la figura debido al tipo de figura de ajedrez.
Descripción: El usuario del escenario intenta mover una figura de ajedrez para comprobar la validez del movimiento. El sistema comprueba que: • La figura existe en el tablero de juego. • La posición destino no excede los límites del tablero. • La posición destino se encuentra entre los movimientos posibles a realizar por la figura. Tras estas comprobaciones si el movimiento es válido la pantalla se refresca y se refleja el movimiento de la pieza, se ofrece por pantalla la información del movimiento realizado. Si no es válido, se informa por pantalla del hecho y no se realiza la acción.

Figura 5.2.1.4: Tabla descriptiva del caso de uso "Mover figura de ajedrez".

6 Diseño

Una vez comentados los requisitos y construidos los casos de uso en base a las especificaciones de requisitos, se presenta a continuación, el diseño del sistema.

6.1 Arquitectura

La arquitectura del escenario desarrollado se basa en la arquitectura provista por la herramienta Greenfoot para el desarrollo de un escenario en esta herramienta. Se va a proponer a continuación la arquitectura del sistema como una colaboración entre los paquetes de clases desarrollados para la solución, en la interfaz de usuario provista por Greenfoot.

Como introducción básica para conocer como está distribuida la interfaz de usuario de la herramienta, se describen sus aspectos fundamentales:

- **Escenario:** en este espacio se desplegaría en nuestro caso, el tablero de ajedrez en dos dimensiones y se permite la visualización de las figuras.

- **Vista de las clases:** en la parte derecha de la interfaz, se despliega una vista de todas las clases que componen el proyecto.

- **Panel de control:** donde se encuentran los botones que controlan la ejecución del programa.

En la vista de clases se observan tres tipos de clases:

- **World Classes:** en Greenfoot todas las clases que se crean en este paquete, descienden de la clase abstracta "World" provista por la herramienta. Esta clase define el mundo en el que los objetos "viven".

- **Actor Classes:** en este paquete de clases, todas las clases creadas, descenderán de la clase abstracta "Actor" provista por la herramienta. Un actor es un objeto que "vive" en el "mundo", cada actor tiene una localización en el mundo y una apariencia.

- **Other Classes:** en este paquete de clases se definen las clases que no descienden de ninguna de las clases predefinidas de Greenfoot y que son totalmente independientes de su API.

A continuación se ofrece el diagrama de paquetes que muestra la arquitectura del sistema:

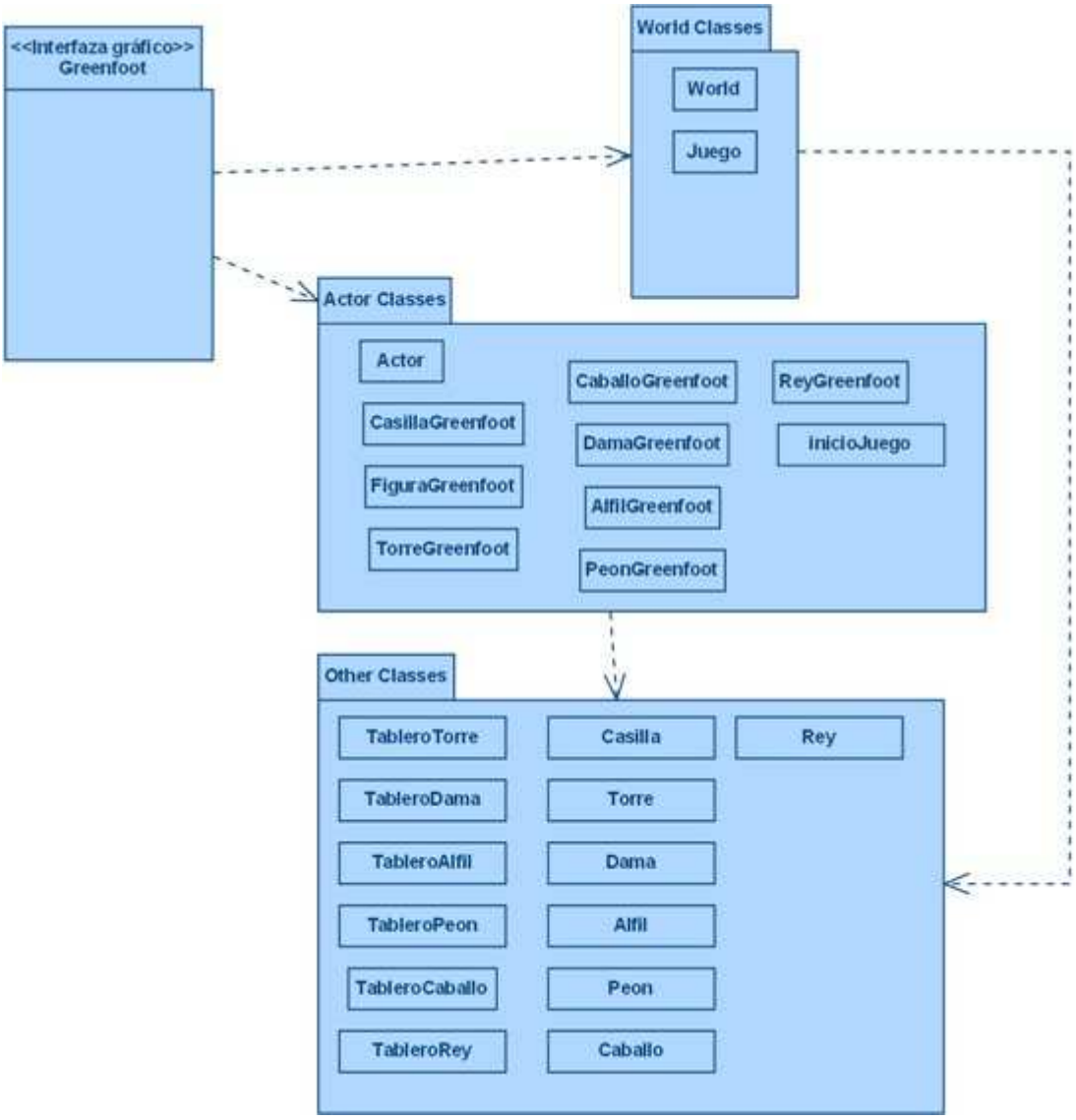


Figura 6.1: Diagrama de paquetes del sistema.

En la figura se puede observar la colaboración entre los paquetes de clases que componen la solución del sistema. Las partes que integran la arquitectura son:

- **El interfaz gráfico de Greenfoot:** paquete ofrecido por Greenfoot en cuya implementación no nos vamos a introducir, se compone de las clases que hacen posible la visualización de los objetos creados por el usuario y su comportamiento.
- **World Classes:** es el paquete en el cual se encuentran las clases necesarias para la representación visual del mundo en el que se despliegan los objetos creados.

- **Actor Classes:** es el paquete en el cual se encuentran las clases necesarias para la creación de los objetos y su representación gráfica en el mundo en el cual son desplegados.

- **Other Classes:** es el paquete de clases en el cual se encuentran las clases de apoyo para los paquetes World y Actor Classes.

6.2 Diagrama de clases de la solución

A continuación se ofrece el diagrama de clases de cada paquete por el que está compuesto el sistema.

6.2.1 Diagrama de clases del paquete World Classes

Este diagrama se corresponde con el paquete World Classes, se presentan los métodos y atributos más importantes de cada clase. En este paquete se observa que la clase creada Juego es subclase de la clase abstracta World, provista por la herramienta. La clase Juego hereda todos los atributos y métodos de la clase World e implementa los suyos propios.

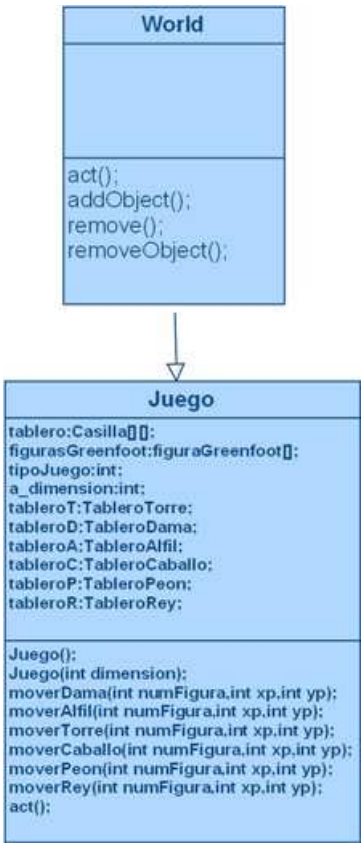


Figura 6.2.1: Diagrama de clases del paquete "World Classes".

6.2.2 Diagrama de clases del paquete Actor Classes

Este diagrama se corresponde con el paquete Actor Classes. Se observa una herencia de todas las clases que componen el paquete, de la clase abstracta provista por Greenfoot, Actor. La clase FiguraGreenfoot se trata de la clase padre para las clases de las figuras de ajedrez. Se crea esta clase debido a que aunque cada figura de ajedrez posee su propio comportamiento, poseen atributos comunes como los de localización y los necesarios para la realización de la simulación de sus movimientos. Cada una de éstas hereda los atributos definidos en FiguraGreenfoot, así como sus métodos, aunque en cada una de éstas se sobrescriben los métodos necesarios.

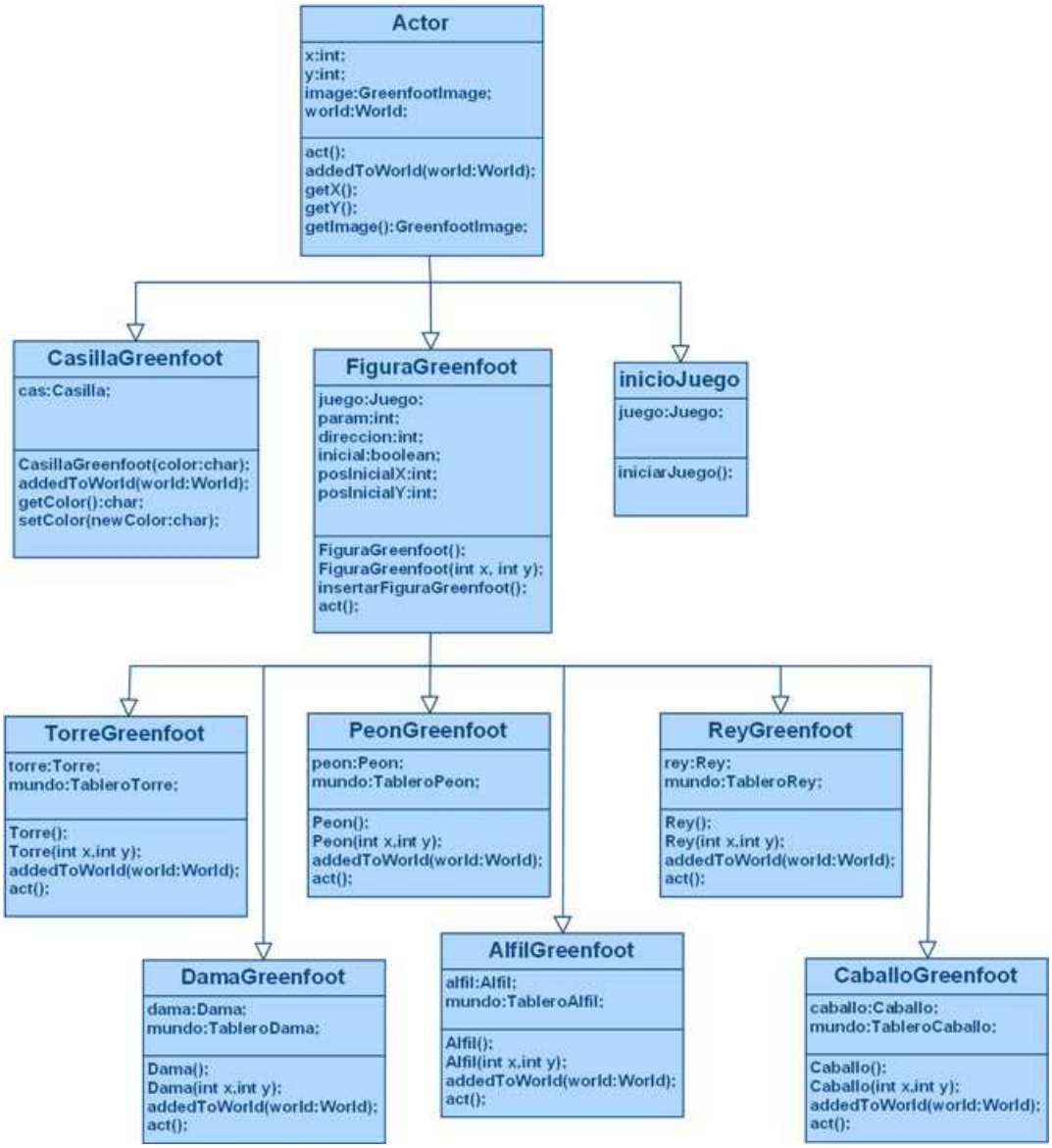


Figura 6.2.2: Diagrama de clases del paquete "Actor Classes".

6.2.3 Diagrama de clases del paquete Other Classes

Este diagrama se corresponde con el paquete Other Classes. En este paquete se encuentran las clases que los estudiantes desarrollan en el entorno jGrasp y portan a la herramienta Greenfoot.

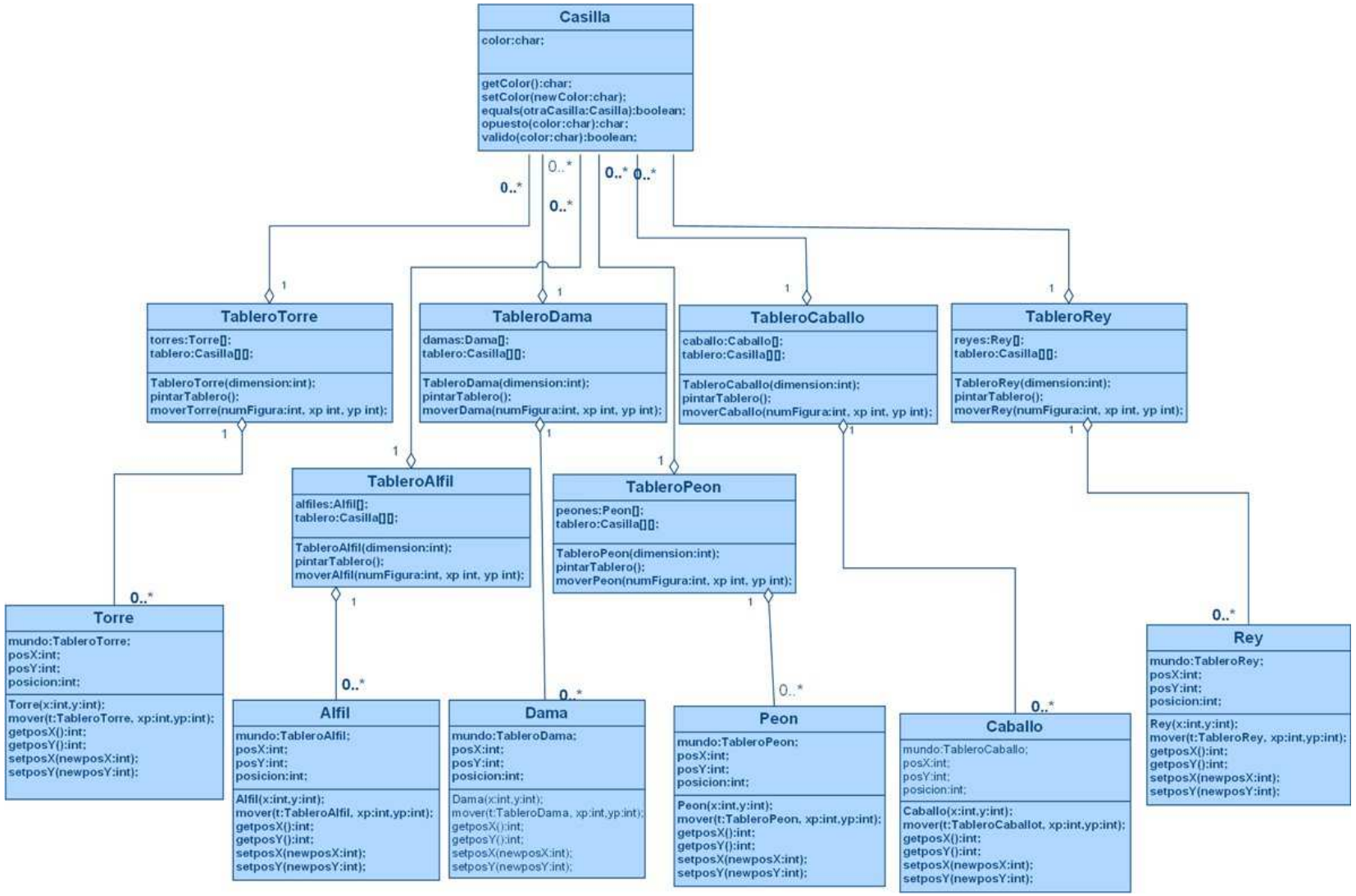


Figura 6.2.3: Diagrama de clases del paquete "Other Classes".

6.3 Diagramas de secuencia del sistema.

A continuación se presenta el diagrama de secuencia que representa la interacción entre los objetos que componen el sistema.

El diagrama de secuencia propuesto es el correspondiente a las acciones del usuario dentro del escenario: *creación del tablero de ajedrez, creación de una figura de ajedrez y posicionamiento en el tablero y por último, petición de movimiento de la figura.* No se especifica una clase de figura concreta, puesto que para todos los tipos de figuras existentes, el diagrama sería el mismo.

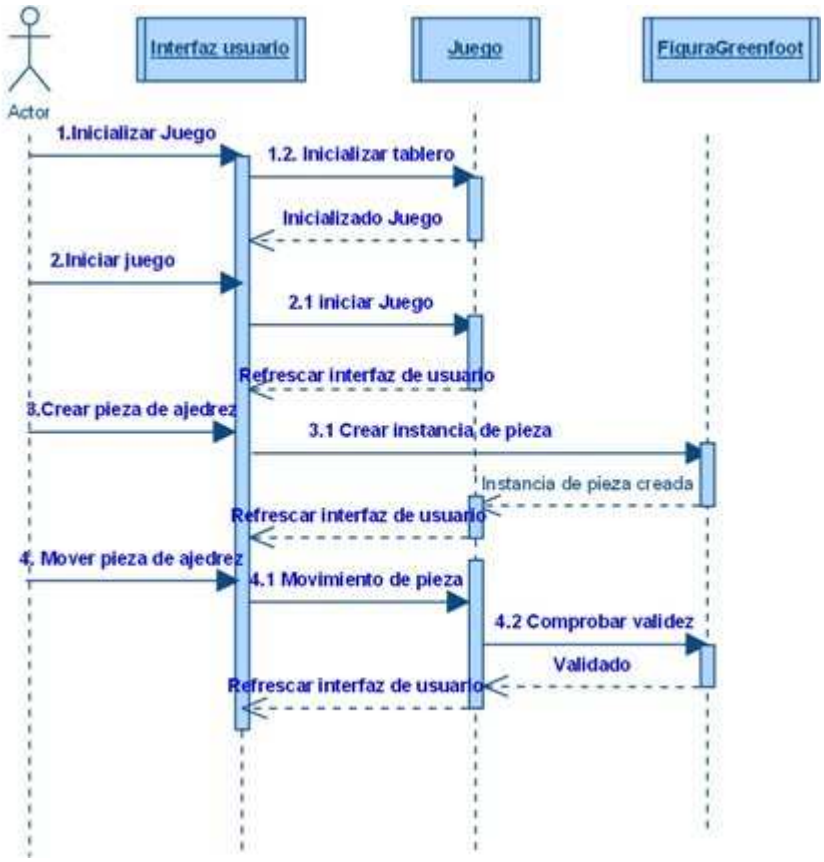


Figura 6.3: Diagrama de secuencia del sistema.

7 Implementación

Una vez comentado el diseño de la solución, pasamos a la fase de implementación. El desarrollo se realiza en la tecnología Java, que es la tecnología en la que está basada la herramienta Greenfoot.

Tomando lo anteriormente citado en el apartado de diseño, la interfaz gráfica es ofrecida por Greenfoot, por lo tanto ya está implementada y no nos introduciremos en ese aspecto. A continuación vamos a pasar a la descripción a alto nivel de cada módulo que compone el sistema.

7.1 Clases que componen el paquete World Classes

En este paquete es donde se define la clase que permite la inicialización del juego y sus atributos.

Esta clase se denomina **Juego** y se trata de una subclase de la clase abstracta **World** que provee la herramienta Greenfoot. En esta clase, se define un método que permite, al abrir el escenario, ofrecer al usuario las instrucciones de juego y los pasos a seguir en el lienzo destinado para la instanciación de objetos. Con esto se pretende facilitar la experiencia del usuario en el juego.

En esta clase se encuentran los constructores del tablero de ajedrez en el que se situarán las piezas. Despliega dos constructores para instanciar la clase: uno por defecto, necesario para poder abrir el proyecto en Greenfoot y el segundo constructor, en el que se elige la dimensión del tablero deseada.

En esta clase también se inicializan los seis modos de juego existentes, tantos como tipos de piezas de ajedrez, esto es así para que cuando el usuario interactúe con un tipo de piezas en el tablero, sólo pueda observar piezas de ese tipo y no se permita la adicción de figuras de distinto tipo.

Continuando con la descripción de la clase Juego, en ésta se declaran los métodos correspondientes al movimiento de una figura de ajedrez, se despliegan seis métodos de movimiento, uno por cada figura de ajedrez. Por lo tanto, para ejecutar el movimiento de una pieza, el método debe ser invocado en la instancia creada de la clase Juego.

Greenfoot permite al usuario ejecutar una simulación de los objetos que crea y observar su comportamiento, en esta clase se definen una serie de métodos que permiten al usuario observar una simulación de la finalidad del juego, con todos los pasos que se deben realizar en el sistema. Esto se observará si el usuario acciona el botón de *Run* que ofrece la herramienta.

7.2 Clases que componen el paquete Actor Classes

En este paquete es donde se definen las clases que permiten la creación de las piezas y casillas que se situarán en el tablero de juego.

7.2.1 FiguraGreenfoot

Esta clase hereda directamente de la clase abstracta Actor provista por Greenfoot. Es necesario remarcar principalmente la manera en que Greenfoot tiene de crear objetos:

1. **Se instancia el objeto.**
2. **Se añade al mundo.**

Esta clase posee un método de adicción del objeto al lienzo de visualización. Puedes haber creado un objeto pero no se podrá interaccionar con él, hasta que no se haya añadido al mundo.

FiguraGreenfoot se trata de la clase padre de la que heredan todas las clases correspondientes a las seis figuras de ajedrez desarrolladas. En esta clase se definen los atributos de posición y del tablero al que pertenece una pieza de ajedrez. Se definen dos constructores para esta clase. Como ya se citaba antes, Greenfoot necesita un constructor por defecto, éste será aquel que no contenga parámetros para que las piezas puedan ser añadidas mediante el arrastre de la instancia al tablero de juego. El segundo constructor posee dos parámetros de entrada que se corresponden a la posición en la que se pretende insertar la pieza por el usuario.

Por último, remarcar que en esta figura se declarará un método, el cual heredarán el resto de subclases, de inserción de la figura creada en un array de figuras que se encuentran en el tablero de juego.

7.2.2 Subclases de la clase FiguraGreenfoot

Las subclases de esta clase, heredan sus atributos y aparte, añaden algunos métodos necesarios para cada tipo de pieza de ajedrez. Estas subclases son las siguientes: *DamaGreenfoot, TorreGreenfoot, PeonGreenfoot, ReyGreenfoot, AlfilGreenfoot y CaballoGreenfoot.*

Estas clases se definen para la creación de piezas de ajedrez con el comportamiento definido para estas piezas. Se apoyan en los métodos definidos en las clases correspondientes al paquete “Other Classes”. Se definen dos constructores de creación, uno que permita adicionar la pieza mediante el arrastre de la misma al tablero y otra con parámetros de entrada, correspondientes a la posición deseada. Estas clases también heredan métodos y atributos de la clase Actor. Se sobrescribe el método de adicción al mundo de esta clase, para añadir las piezas según conveniencia. La imagen de las piezas al ser añadidas al tablero de juego, ofrece un número que las identifica para poder invocar el movimiento de la pieza.

Como restricciones en la adicción de las instancias de estas clases se remarca, que si el juego no está inicializado o el tablero construido, no se permite la creación de estas piezas. Así como instanciar objetos de esta clase con coordenadas incorrectas o que sobrepasen los límites del tablero.

7.2.3 CasillaGreenfoot

También se trata de una subclase de Actor, esta clase se define para la instanciación de objetos casilla, que conforman el tablero del juego. Posee un atributo con el color de la casilla, que puede ser blanco o negro y una serie de métodos que se invocarán desde la clase Juego para una correcta construcción del tablero de ajedrez, con las casillas de colores alternados. Esta clase se apoya en los métodos de la clase Casilla que se encuentra en el paquete "Other Classes" para definir sus propios métodos y realizar las comprobaciones necesarias.

7.2.4 inicioJuego

Subclase de la clase Actor, es creada para poder añadir al escenario, un objeto a través del cual se puede inicializar el juego. Esta clase posee el método de inicio de juego en el cuál se añaden los objetos casillas que conforman el tablero. El algoritmo que se sigue para el relleno de este tablero es el siguiente:

Principio del procedimiento
Color=Blanco
For i=0 hasta tablero.longitud
For j=0 hasta tablero[i].longitud
Crear objeto Casilla (Color)
Color opuesto
Añadir el objeto Casilla
Fin For
Fin For
Fin del procedimiento

7.3 Clases que componen el paquete Other Classes

En este paquete se agrupan las clases que deben desarrollar los alumnos para sus prácticas. Estas clases son desarrolladas en la herramienta jGrasp, por lo tanto no utilizan la librería de Greenfoot. Se puede describir este paquete como el paquete de clases en el que se definen las reglas del juego, como se debe rellenar el tablero de juego correctamente, como se añaden las piezas y como se mueven por el tablero.

Como existen seis modos de juego, se crea una clase diferente de tablero para cada modo de juego. Estas clases sirven de apoyo a los otros dos paquetes del sistema, puesto que definen los métodos principales para poder realizar el movimiento de una figura en un tablero de ajedrez, además de la creación de un tablero de juego. Pero no pueden construir una instancia visual de un objeto.

Por cada modo de juego se declaran una serie de clases:

- **Modo de juego Torre:** se declara la clase TableroTorre y la clase Torre.
- **Modo de juego Caballo:** se declara la clase TableroCaballo y la clase Caballo.
- **Modo de juego Alfil:** se declara la clase TableroAlfil y la clase Alfil.
- **Modo de juego Peón:** se declara la clase TableroPeón y la clase Peón.
- **Modo de juego Dama:** se declara la clase TableroDama y la clase Dama.
- **Modo de juego Rey:** se declara la clase TableroRey y la clase Rey.

Las clases correspondientes a los tableros de juego se definen como las clases que almacenan el vector de casillas para conformar el tablero y el vector de figuras que se añaden al tablero. Contienen métodos utilizados por la clase Juego, para rellenar correctamente el tablero de juego y para realizar las comprobaciones necesarias a la hora de mover una figura de ajedrez.

La clase Casilla que se desarrolla en este paquete es la clase que contiene los métodos que la clase CasillaGreenfoot utiliza para añadir una casilla al tablero correctamente.

A continuación se presentan las clases correspondientes a la implementación del comportamiento de las figuras de ajedrez. Estas clases son las siguientes: **Alfil, Torre, Dama, Caballo, Peón y Rey.** Poseen atributos de posición necesarios para situarse en el tablero y métodos necesarios para su movimiento correcto por el tablero de juego.

Para cada pieza, se va a describir en pseudocódigo, el algoritmo de movimiento de la pieza.

7.3.1 Clase Alfil

Se presenta a continuación una imagen que ilustra el comportamiento de este tipo de figuras, en el tablero de ajedrez.

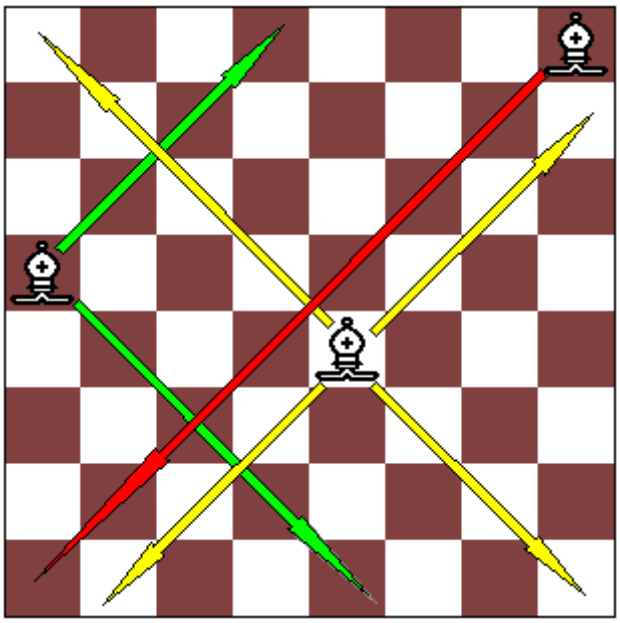


Figura 7.3.1: Imagen de los movimientos permitidos para el alfil.

Inicio del procedimiento

```
X=posicionX; //Se guarda la posición inicial de la pieza.
Y=posicionY;

        //Calcular el incremento que se debe dar para cada coordenada.
        IncrementoX=calcularIncremento(X, posicionDestinoX);
        incrementoY=calcularIncremento(Y, posicionDestinoY);

        Repetir
                posicionX =posicionX+ incrementoX;
                posicionY= posicionY+ incrementoY;

        Mientras
                (NohayFigura(posicionDestinoX, posicionDestinoY) y
                 posicionX ¡= posicionDestinoX y posicionY ¡= posicionDestinoY)

        Si      (hayFigura(posicionDestinoX, posicionDestinoY) y
                 posicionX ¡= posicionDestinoX y posicionY ¡= posicionDestinoY)

                posicionX=X;
                posicionY=Y;

                devolver false;
        Si no   devolver true;
```

Fin del procedimiento

7.3.2 Clase Torre

Se presenta a continuación una imagen que ilustra el comportamiento de este tipo de figuras, en el tablero de ajedrez.

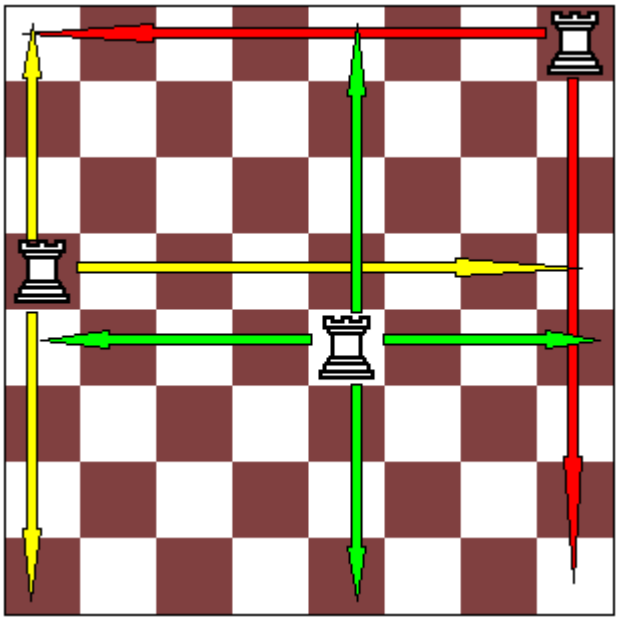


Figura 7.3.2: Imagen de los movimientos permitidos para la torre.

Inicio del procedimiento

X=posicionX; //Se guarda la posición inicial de la pieza.
Y=posicionY;

```
//Calcular el incremento que se debe dar para cada coordenada.
IncrementoX=calcularIncremento(X,posicionDestinoX);
incrementoY=calcularIncremento(Y,posicionDestinoY);

Repetir

        posicionX=posicionX+ incrementoX;
        posicionY= posicionY+ incrementoY;

Mientras
        (NohayFigura(posicionDestinoX, posicionDestinoY) y
         posicionX ¡= posicionDestinoX y posicionY ¡= posicionDestinoY)

Si      (hayFigura(posicionDestinoX, posicionDestinoY) y
         posicionX ¡= posicionDestinoX y posicionY ¡= posicionDestinoY)

        posicionX=X;
        posicionY=Y;

        devolver false;
Si no   devolver true;
```

Fin del procedimiento

7.3.3 Clase Dama

Se presenta a continuación una imagen que ilustra el comportamiento de este tipo de figuras, en el tablero de ajedrez.

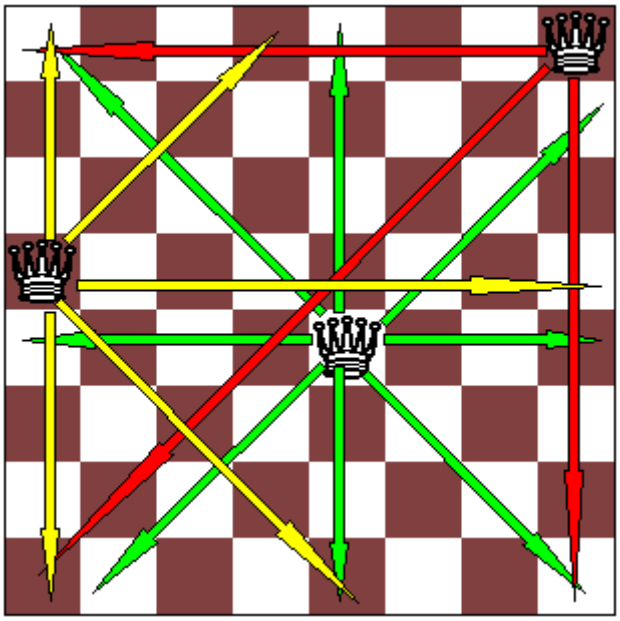


Figura 7.3.3: Imagen de los movimientos permitidos para la dama.

```
Inicio del procedimiento

X=posicionX; //Se guarda la posición inicial de la pieza.
Y=posicionY;

        //Calcular el incremento que se debe dar para cada coordenada.
        IncrementoX =calcularIncremento(X,posicionDestinoX);
        incrementoY =calcularIncremento(Y,posicionDestinoY);

        Repetir

                posicionX =posicionX+ incrementoX;
                posicionY = posicionY+ incrementoY;

        Mientras
                (NohayFigura (posicionDestinoX, posicionDestinoY) y
                 posicionX ¡= posicionDestinoX y posicionY ¡= posicionDestinoY)

        Si      (hayFigura(posicionDestinoX, posicionDestinoY) y
                 posicionX ¡= posicionDestinoX y posicionY ¡= posicionDestinoY)

                posicionX=X;
                posicionY=Y;

                devolver false;
        Si no   devolver true;
Fin del procedimiento
```

7.3.4 Clase Peón

Se presenta a continuación una imagen que ilustra el comportamiento de este tipo de figuras, en el tablero de ajedrez.

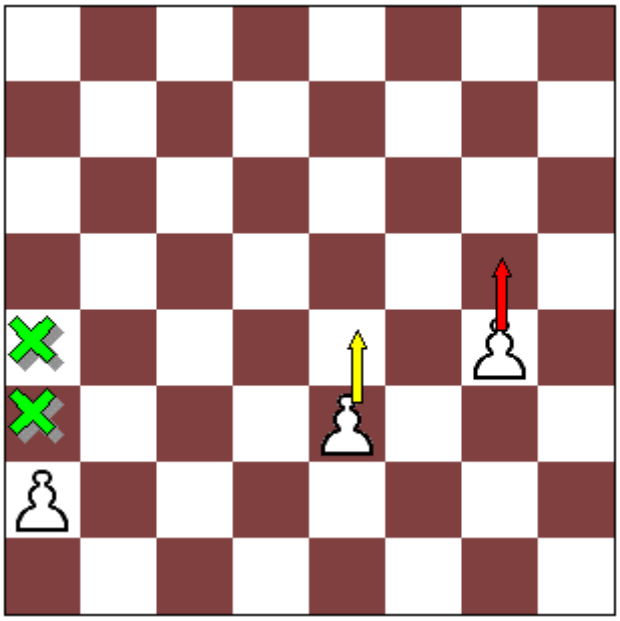


Figura 7.3.4: Imagen de los movimientos permitidos para el peón.

Inicio del procedimiento

X=posicionX; //Se guarda la posición inicial de la pieza.
Y=posicionY;

```
//Calcular el incremento que se debe dar  para la coordenada Y
        incrementoY =calcularIncremento (Y,posicionDestinoY);

Repetir

        posicionY = posicionY+ incrementoY;

Mientras
        (NohayFigura(posicionDestinoX, posicionDestinoY) y
         posicionY ¡= posicionDestinoY)

Si      (hayFigura (posicionDestinoX, posicionDestinoY) y
         posicionY = posicionDestinoY)

        posicionY=Y;

        devolver false;
Si no   devolver true;
```

Fin del procedimiento

7.3.5 Clase Caballo

Se presenta a continuación una imagen que ilustra el comportamiento de este tipo de figuras, en el tablero de ajedrez.

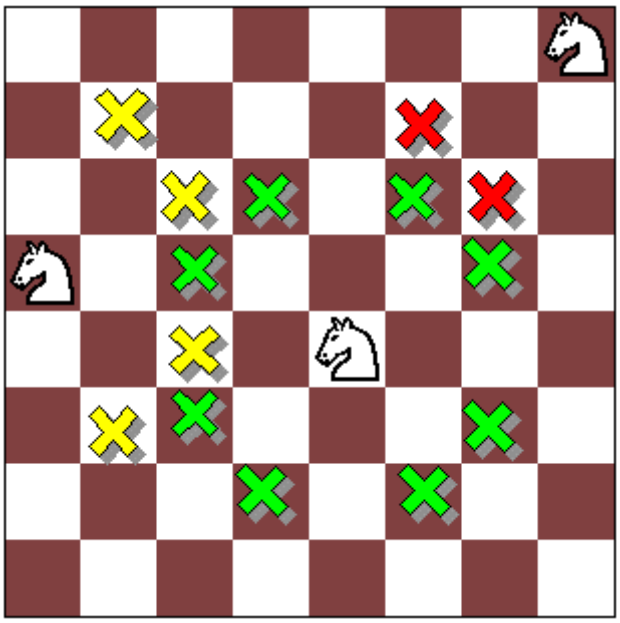


Figura 7.3.5: Imagen de los movimientos permitidos para el caballo.

Inicio del procedimiento

X=posicionX; //Se guarda la posición inicial de la pieza.
Y=posicionY;

Si (hayFigura (posicionDestinoX, posicionDestinoY) y
posicionY ¡= posicionDestinoY)

avanzarModoCaballo (posicionDestinoX, posicionDestinoY)

devolver true;

Si no devolver false;

Fin del procedimiento

7.3.6 Clase Rey

Se presenta a continuación una imagen que ilustra el comportamiento de este tipo de figuras, en el tablero de ajedrez.

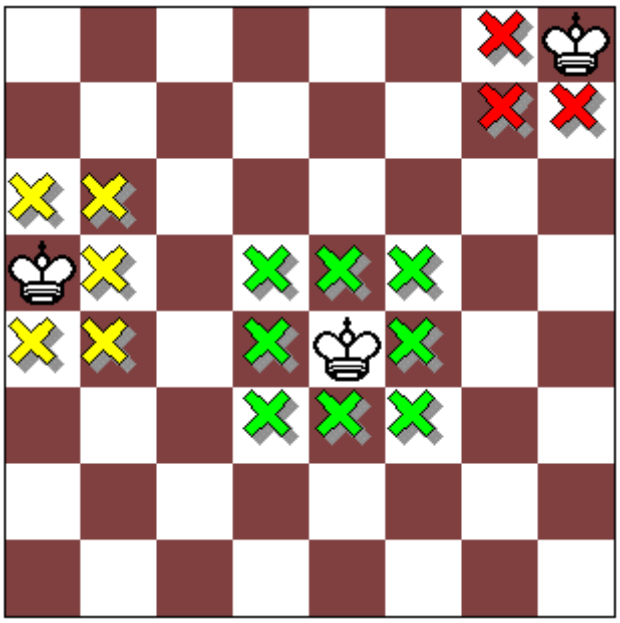


Figura 7.3.6: Imagen de los movimientos permitidos para el rey.

Inicio del procedimiento

X=posicionX; //Se guarda la posición inicial de la pieza.
Y=posicionY;

```
//Calcular el incremento que se debe dar  para las coordenadas
        incrementoX =calcularIncremento(X,posicionDestinoX);
        incrementoY =calcularIncremento(Y,posicionDestinoY);

        posicionX= posicionX+ incrementoX;
        posicionY= posicionY+ incrementoY;

Si      (hayFigura(posicionDestinoX, posicionDestinoY)

        posicionX=X;
        posicionY=Y;
        devolver false;
Si no   devolver true;
```

Fin del procedimiento

8. Pruebas

8.1 Introducción

La necesidad de comprobar el correcto funcionamiento del producto hace que sea imprescindible un plan de pruebas, con el cual se procederá a realizar una serie de ensayos que permitan obtener resultados correctos y erróneos con el fin de analizar el proceso de ejecución.

El aspecto más importante para realizar la planificación de este conjunto de pruebas, es abarcar con ellas todos los requisitos que debe cumplir el programa y que por tanto responda correctamente a las funcionalidades que se le solicitan inicialmente.

Puesto que en el apartado de especificación de requisitos software ya se ha realizado una evaluación de las funcionalidades que debe incluir el programa, tomaremos estos requisitos de referencia para desarrollar el plan de pruebas de sistema.

8.2 Definición de las pruebas del sistema

La propuesta que se plantea relacionada con el plan de pruebas es que se realizan un conjunto de pruebas basadas directamente en la especificación de requisitos. Con estas pruebas se intenta verificar que el sistema final cumple con las especificaciones funcionales descritas por los casos de uso originales.

Como estrategia de pruebas se van a seguir una serie de principios propuestos para un plan de pruebas razonable. [19]

Casos correspondientes al curso normal del caso.

1. Casos correspondientes al curso normal del caso.
2. Casos correspondientes a cursos excepcionales

8.3 Casos de prueba

A continuación, revisaremos los casos de uso principales, los cuáles fueron descritos en el análisis de requisitos.

8.3.1 CASO DE PRUEBA: Iniciar el juego

- Nótese que los casos de uso *Inicializar el tablero de ajedrez* y *Pintar el tablero de ajedrez* no se prueban independientemente ya que son inclusiones del caso de uso *Iniciar el juego.* La secuencia de este caso de uso es la siguiente, primero se invoca al constructor de Juego y se inicializan los atributos y luego se invoca al método *iniciarJuego()* de la clase Juego, que pinta el tablero de ajedrez.

1. **Curso normal del caso de uso.**

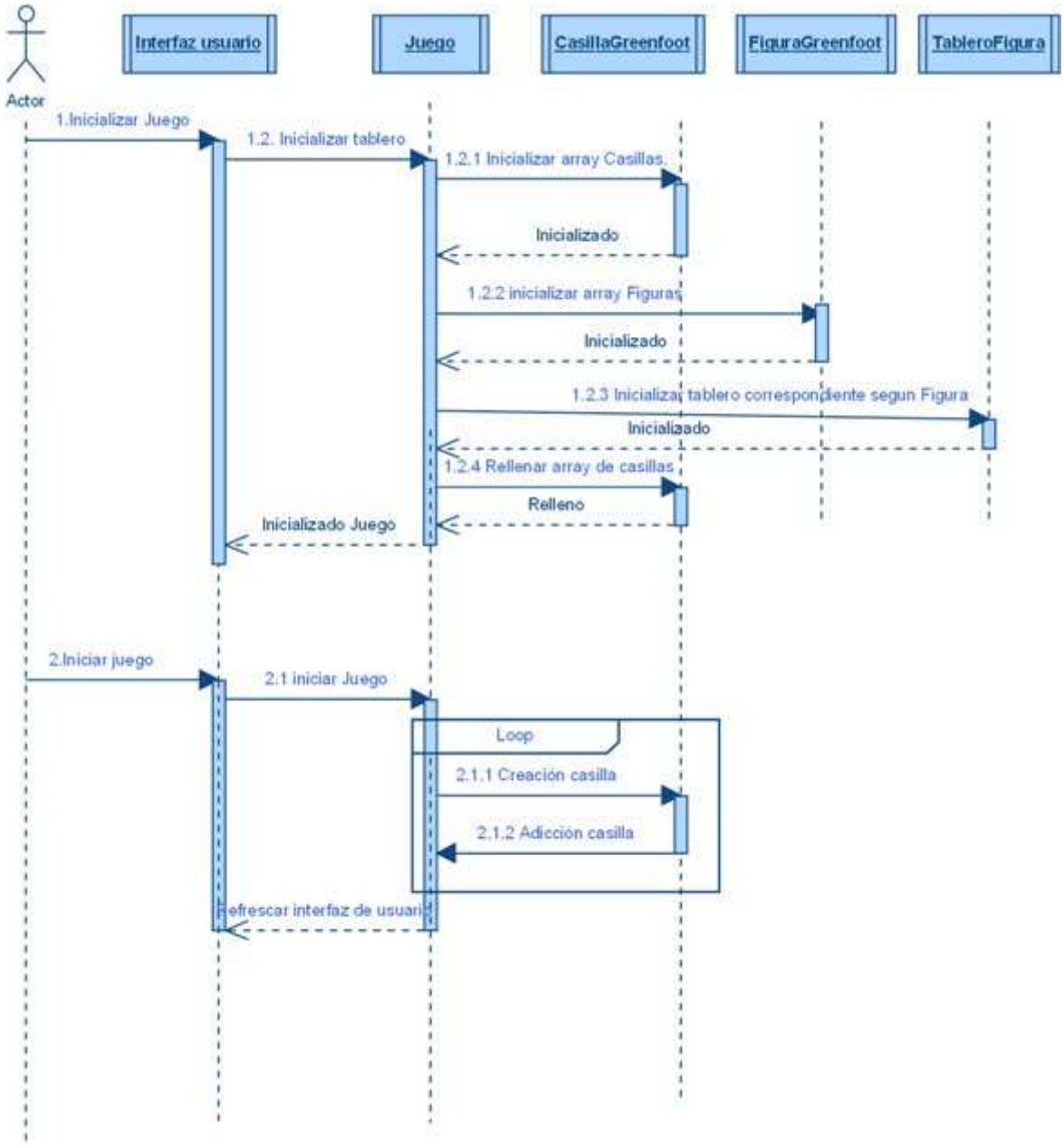


Figura 8.3.1.1: Diagrama de secuencia par el caso de prueba "Iniciar Juego".

A continuación se muestra la inicialización del tablero de juego: se introduce como dimensión del tablero el valor 5 y tipo de juego el valor 1.

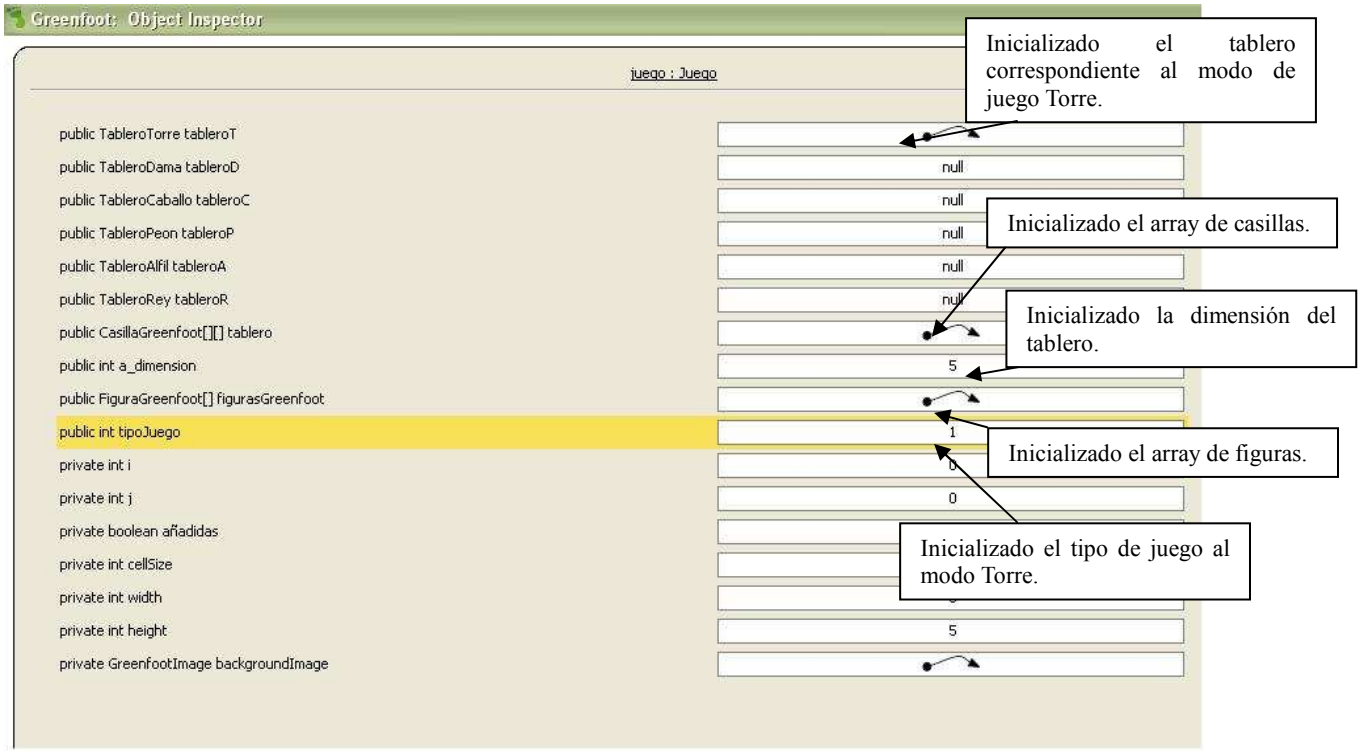


Figura 8.3.1.2: Imagen acerca del estado del tablero de juego en el caso de prueba "Iniciar Juego".

Una vez mostrada la inicialización del tablero de juego, se muestra el inicio del juego que se corresponde con la apariencia del tablero, que debe ser la propia de un tablero de ajedrez.

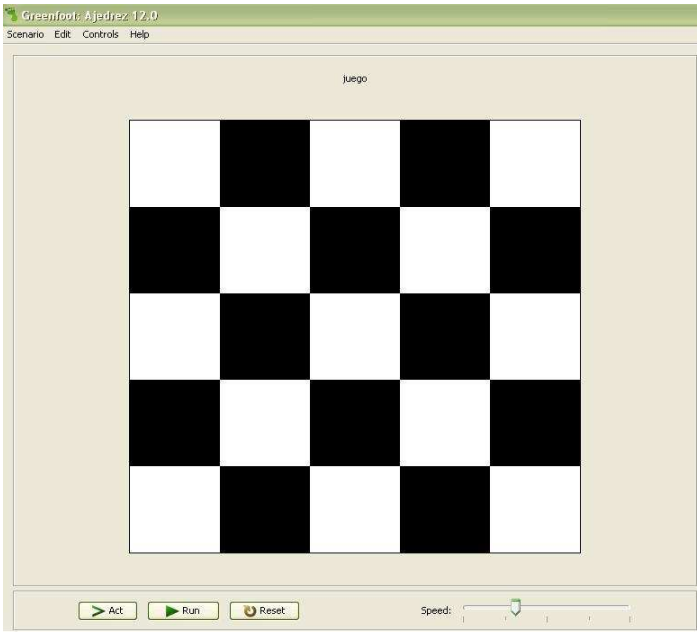


Figura 8.3.1.3: Imagen del interfaz una vez "iniciado el juego".

2. Curso excepcional del caso de uso.

A continuación se muestra la respuesta del sistema cuando se introduce una dimensión para el tablero de ajedrez mayor de lo permitido, que es dimensión 20.

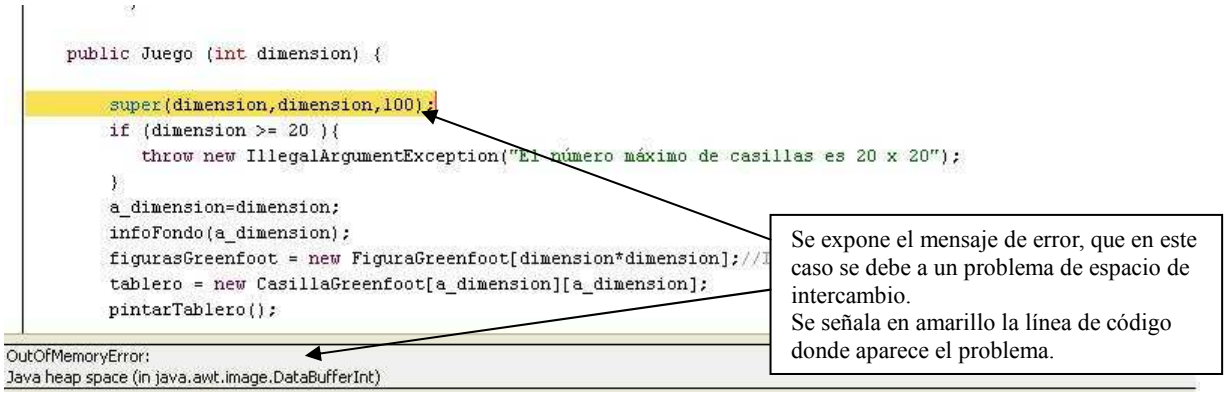


Figura 8.3.1.3: Imagen de la excepción lanzada en pantalla, por dimensión de tablero mayor de 20.

El método *iniciarJuego ()* podrá ser invocado aunque no se haya introducido una dimensión correcta, ya que por defecto, el tablero está inicializado a dimensión 5 y tipo de juego Torre.

8.3.2 CASO DE PRUEBA: Crear figura de ajedrez

1. Curso normal del caso de uso.

La secuencia de este caso de uso se corresponde con una llamada al constructor de la figura que se desea añadir al tablero, previamente tiene que estar iniciado el juego, es decir, construido el tablero de juego.

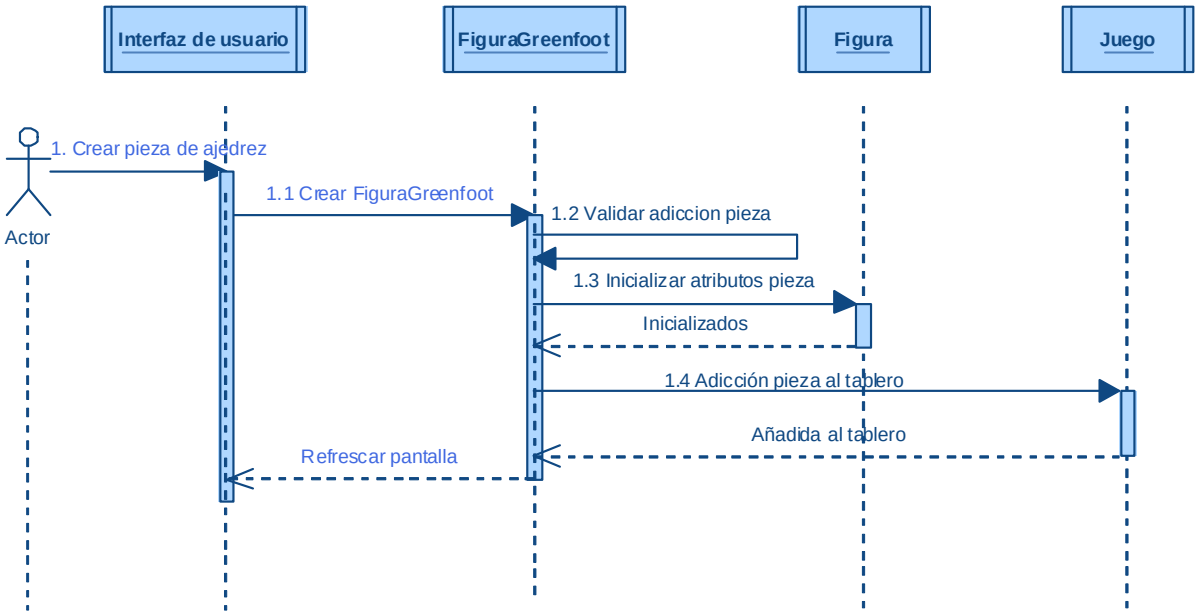


Figura 8.3.2.1: Diagrama de secuencia del caso de prueba para "Crear Figura de Ajedrez".

En la siguiente figura se muestra la inicialización de los atributos de la pieza de ajedrez.

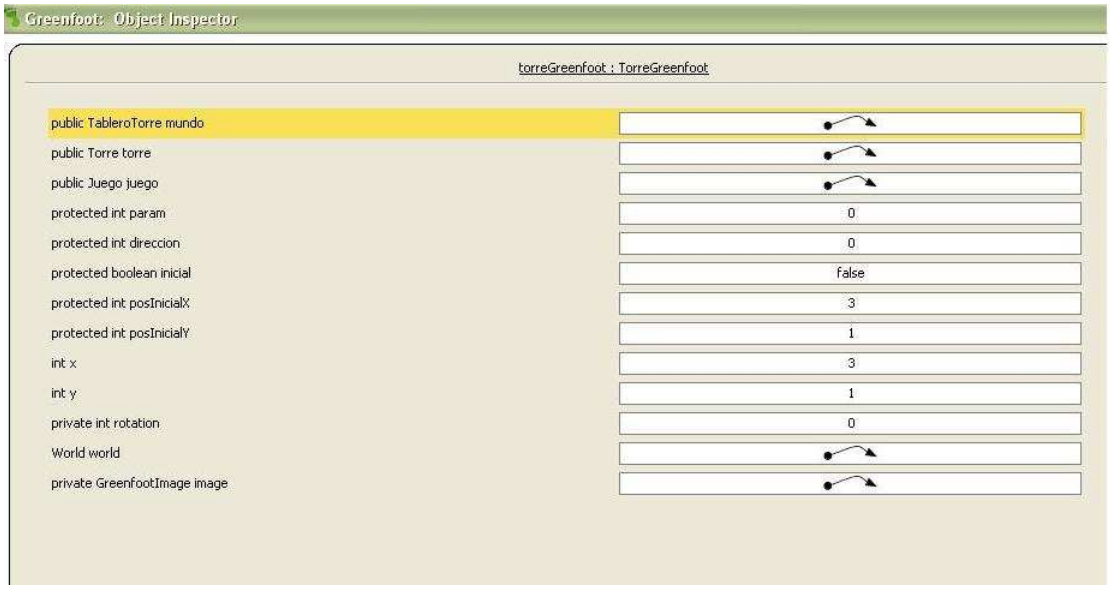


Figura 8.3.2.2: Imagen del estado de la figura creada.

A continuación, se observa la pieza añadida al tablero de juego.

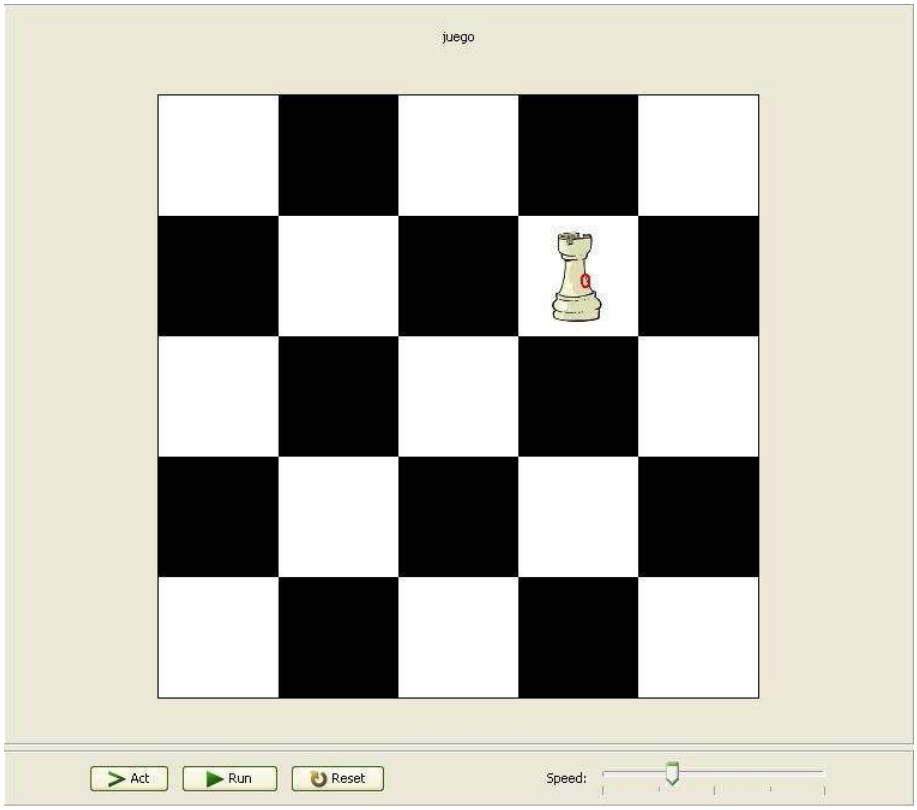


Figura 8.3.1.3: Imagen de la interfaz, con la figura añadida al tablero.

3. Curso excepcional del caso de uso.

A continuación se muestra la respuesta del sistema cuando se introducen unas coordenadas de posición para la pieza a crear erróneas.

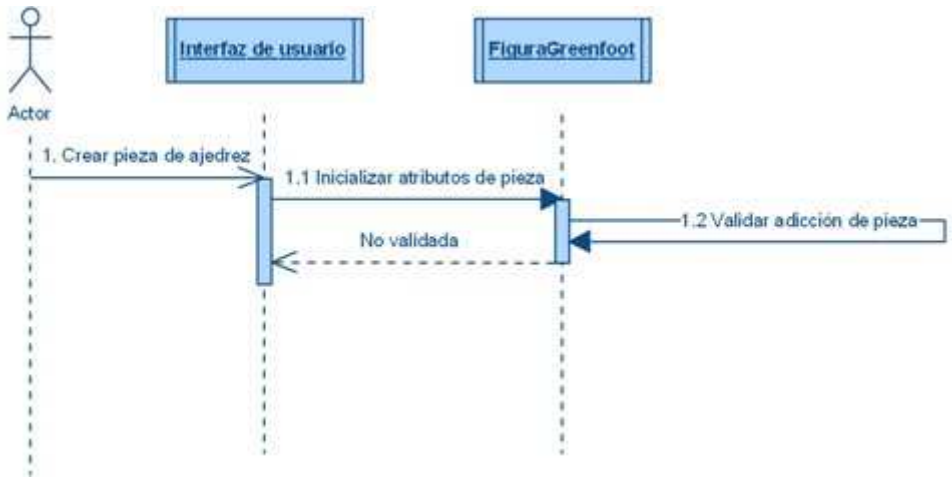


Figura 8.3.1.4: Diagrama de secuencia para caso de prueba excepcional de "Crear figura de ajedrez".

Se presenta la excepción que es lanzada por pantalla cuando las coordenadas no son correctas.

Figura 8.3.1.5: Excepción lanzada cuando las coordenadas para crear una figura no son correctas.

8.3.2 CASO DE PRUEBA: Mover figura de ajedrez

1. Curso normal del caso de uso.

La secuencia de este caso de uso se corresponde con una llamada al tablero de juego, que posee el método para el movimiento de las piezas, a continuación se procederá a la validación de ese movimiento y por último, se produce el movimiento de la figura.

La figura de ajedrez debe estar creada anteriormente.

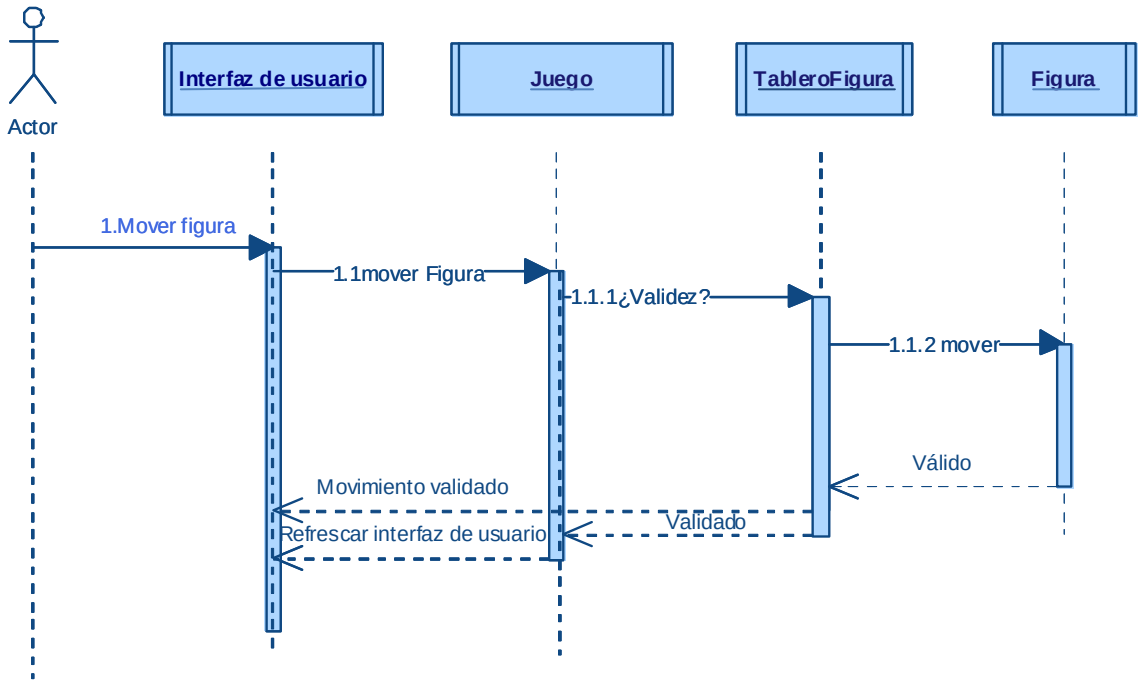


Figura 8.3.2.1: Diagrama de secuencia para caso de prueba "Mover figura de ajedrez"

A continuación se ofrece la imagen del cambio de localización de la figura, se informa por pantalla de que el movimiento es válido y ha sido realizado.

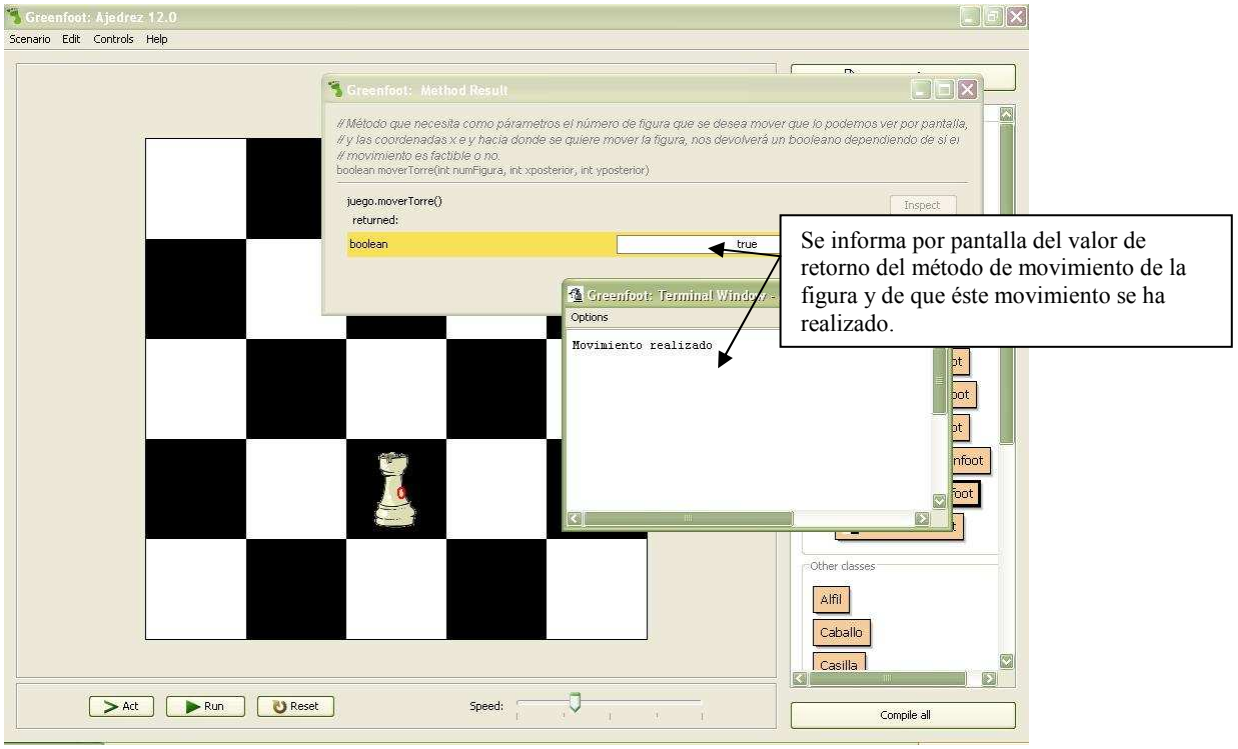


Figura 8.3.2.2: Imagen de salida por pantalla de información debido a que el movimiento es realizado.

2. Curso excepcional del caso de uso.

Se presenta un caso excepcional de este caso de uso. Presenta la respuesta del sistema cuando se introducen unas coordenadas para el movimiento de la figura no válidas.

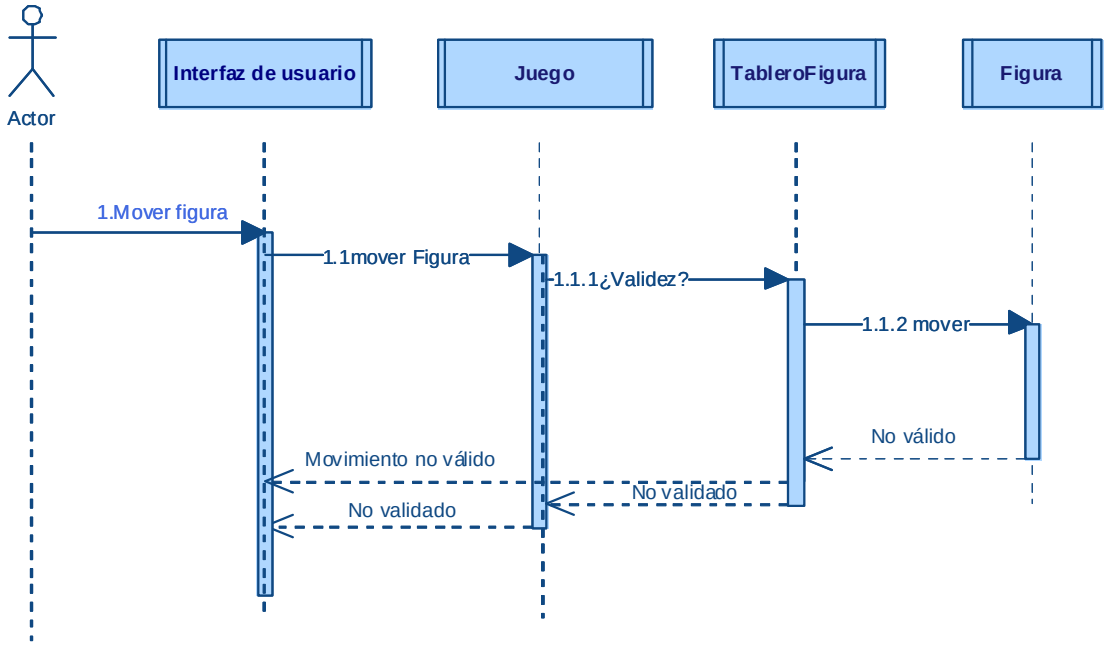


Figura 8.3.2.3: Diagrama de secuencia para caso de prueba excepcional de "Mover figura de ajedrez".

La salida por pantalla que observa el usuario se ofrece a continuación.

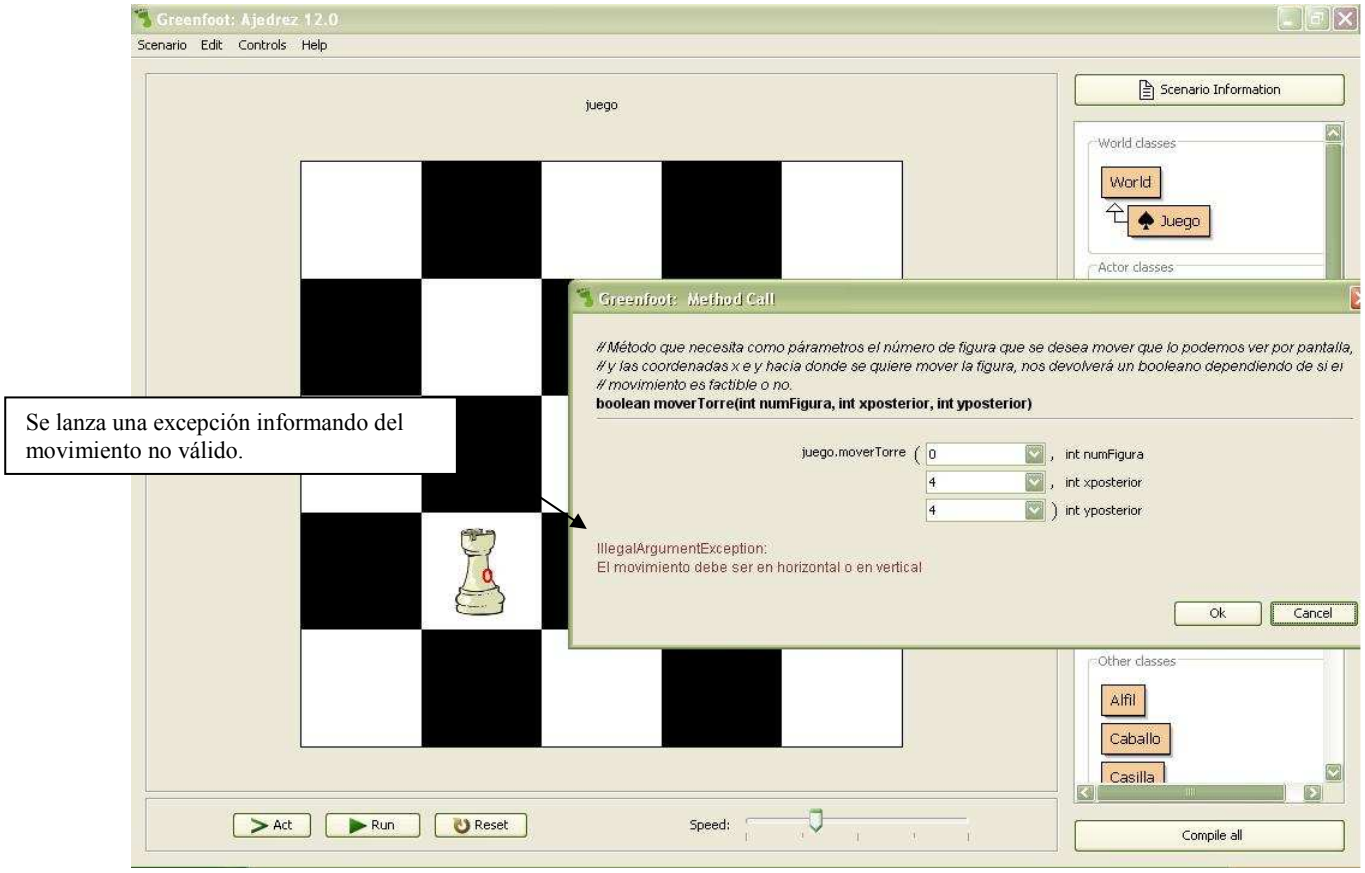


Figura 8.3.2.4: Excepción lanzada por argumentos inválidos al tratar de mover una figura.

9. Conclusiones

Se ha realizado un escenario basado en la herramienta Greenfoot que suministra una solución para los problemas encontrados en la enseñanza temprana de la POO. Con el mecanismo que se ofrece, el cuál posee una interfaz gráfica atractiva, también se defiende que la introducción en la programación puede ser atractiva.

La solución propuesta no pretende sustituir en las prácticas de asignaturas de introducción a la programación, el desarrollo de programas en entornos sin interfaz gráfica. Pretende ser un complemento a esta forma de desarrollo para asentamiento de los conceptos de la POO. Se intenta que el alumno no pierda el tiempo aprendiendo el funcionamiento de otra herramienta, por lo tanto la solución basada en Greenfoot permite que el usuario tenga un mínimo conocimiento de la herramienta, para no avasallar con conocimientos, a veces, innecesarios.

9.1 Trabajos futuros

El mecanismo propuesto se implantará en las prácticas de la asignatura de Programación de la titulación Ingeniería de Telecomunicación de la Universidad Carlos III de Madrid. Como ampliación de este proyecto se podría proponer un plan de evaluación del sistema en cuánto a usabilidad del mismo. Se podría evaluar la experiencia del usuario y el cumplimiento de objetivos de asentamiento de conceptos de la POO. Esto se podría realizar mediante cuestionarios para los estudiantes que experimentan con la solución propuesta y valoración de los resultados obtenidos en los exámenes teóricos y prácticos de la asignatura.

Tras este plan de evaluación, se podrían estudiar las mejoras a realizar y valorar si el sistema realmente cumple con el cometido de apoyo a los estudiantes en su introducción a al POO.

9.2 Conclusiones personales

La realización de este proyecto me ha aportado una introducción a la plataforma Java y a numerosas herramientas de aprendizaje que existen de la POO. La introducción a Java, es una de las causas principales de mi interés por este proyecto ya que en la titulación que he cursado no se me introduce a esta plataforma. Me ha sorprendido descubrir la cantidad de herramientas que existen para el apoyo al aprendizaje de la POO y lo intuitivas y fáciles en su uso que pueden llegar a ser. Con la realización de este proyecto, he afianzado conceptos de este tipo de programación, con lo cual yo valoro personalmente que si a mí me ha sido útil, puede ser útil a más estudiantes en el futuro.

La solución propuesta al comienzo era muy diferente y no permitía una portabilidad fácil de código a los estudiantes como la solución actual, lo cual chocaba con los requisitos del cliente. La mayor dificultad por tanto, ha estado en cumplir con los requisitos del cliente que se iban modificando a medida que avanzaba el proyecto, lo cual es algo que pasa todos los días en el mundo real.

La implementación del sistema no me ha supuesto mucha dificultad, puesto que he realizado escenarios previos a esta solución donde he cogido soltura y conocimiento en el manejo de la herramienta y he podido mejorar en la medida de lo posible la presentación de una solución adecuada.

Por último, considero que la realización de este proyecto me ha producido una mayor atracción hacia el descubrimiento de Java y sus posibilidades, que pueden ser infinitas y de las que solo he visto una mínima parte.

10. Bibliografía

[1][16][19] Greenfoot:
http://www.greenfoot.org/ (Última entrada Septiembre 2009)

[2] Sun Microsystems.
http://es.sun.com/ (Última entrada Septiembre 2009)

[3][4] Computer Science Education: Where Are the Software Engineers of Tomorrow? Robert B.K. Dewar, Edmond Schonberg CrossTalk, The Journal of Defense Software Engineering January 2008.
http://www.stsc.hill.af.mil/CrossTalk/2008/01/0801DewarSchonberg.html (Última entrada Septiembre 2009)

[5] Why BLUEJ? An introduction to the Problems BlueJ Addresses, Michael Kölling University of Kent.
http://www.bluej.org/about/why.html (Última entrada Septiembre 2009)

[6][13][14] Greenfoot: Combining object Visualization with interaction, Poul Henriksen, Michael Kölling.
http://www.greenfoot.org/papers/2004-10-OOPSLA-greenfoot.pdf (Última entrada Septiembre 2009)
Poul Henriksen and Michael Kölling, in Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA), pages 73-82, Vancouver, BC, CANADA, November 2004.

[7] BlueJ:
http://www.bluej.org/ (Última entrada Septiembre 2009)

[8] Clasificación de usuarios basada en la detección de errores usando técnicas de procesadores de lenguaje. Tesis doctoral presentada por Juan Ramón Pérez Pérez. Universidad de Oviedo.
http://www.di.uniovi.es/~cueva/investigacion/tesis/JuanRamon.pdf (Última entrada Septiembre 2009)

[9] The system BlueJ and its pedagogy, Michael Kölling, Bruce Quig, Andrew Patterson, John Rosenberg, Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol 13, No 4, Dec 2003
http://www.bluej.org/papers/2003-12-CSEd-bluej.pdf (Última entrada Septiembre 2009)

[10] Jeliot 3:
http://cs.joensuu.fi/~jeliot/(Última entrada Septiembre 2009)

[11] Alice:
http://www.alice.org/ (Última entrada Septiembre 2009)

[12] Teaching Objects-first in Introductory Computer Science:
http://www.alice.org/publications/TeachingObjectsfirstInIntroductoryComputerScience.pdf

[15] Modelo COCOMO II:
http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html#downloads (Última entrada Septiembre 2009)

[16] jGrasp:
http://www.jgrasp.org/ (Última entrada Septiembre 2009)

[17] Overview jGrasp and the Tutorials,
http://www.jgrasp.org/tutorials187/00_Overview.pdf (Última entrada Septiembre 2009)

[18] Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool, Kelsey Van Haaster and Dianne Hagan Monash University, Melbourne, Australia.
Information Science + Information Technology Education Joint Conference, Rockhampton, QLD, Australia, June 2004.
http://www.bluej.org/papers/2004-06-vanhaaster-hagan.pdf (Última entrada Septiembre 2009)

[19] Título: Principios para los casos de prueba basados en los casos de uso.
I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard: *Object-Oriented Software Engineering: A Use Case Driven Approach,* Addison-Wesley, 1992

[20] HTML:
http://www.lcc.uma.es/~eat/services/html-js/manual14.html (Última entrada Septiembre 2009)

11. Definiciones y acrónimos

A lo largo del documento se usan una serie de términos que pueden no contar con el conocimiento del cliente; de ahí la necesidad de aclararlos:

11.1 Acrónimos

Acrónimo	Descripción
POO	Programación Orientada a Objetos
GUI	Graphical User Interface
API	Application Program Interface
UML	Unified Modeling Language
JDK	Java Development Kit
COCOMO II	Constructive Cost Model II
KSLOC	Kilo Source Line of Code
IDE	Integrated Development Enviroment

11.2 Definiciones

- **Sun Microsystems:** empresa informática, situada en Silicon Valley, fabricante de semiconductores y software. Las siglas Sun se corresponden con Stanford University Network. Algunos productos populares de Sun son: Java, los sistemas operativos Sun OS y Solaris, servidores y estaciones de trabajo para procesadores SPARC, y el paquete StarOffice, que adquirió la compañía StarDivision y fue renombrado a OpenOffice.org.

- **Javadoc:** utilidad de Sun MicroSystems para la generación de documentación de APIs en formato HTML a partir de código fuente Java. Es el estándar de la industria para documentar clases de Java La mayoría de los entornos integrados los generan automáticamente.

- **HTML:** (Hyper Text Markup Language) lenguaje basado en marcas que indican las características del texto, utilizado para definir documentos de hipertexto en Web, es decir, documentos de texto presentados de forma estructurada, con enlaces (en inglés, *links*) que conducen a otros documentos o a otras fuentes de información (por ejemplo, bases de datos) que pueden estar en tu propia máquina o en máquinas remotas de la red. Todo ello se puede presentar acompañado de cuantos gráficos estáticos o animados y sonidos seamos capaces de imaginar.[20]

Anexos

A.A Diagrama de Gantt