SISTEMA DE ENTRETENIMIENTO EN LINEA BASADO EN EL JUEGO DE ESTRATEGIA BATTLETECH

MIGUEL ANGEL CAÑAS CELY

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FISICOMECANICAS
ESCUELA DE INGENIERIA DE SISTEMAS E INFORMATICA
BUCARAMANGA
2004

SISTEMA DE ENTRETENIMIENTO EN LINEA BASADO EN EL JUEGO DE ESTRATEGIA BATTLETECH

MIGUEL ANGEL CAÑAS CELY

Proyecto de grado para optar al titulo de
Ingeniero de Sistemas

Director
ENRIQUE SARMIENTO MORENO
Ingeniero Electricista

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FISICOMECANICAS
ESCUELA DE INGENIERIA DE SISTEMAS E INFORMATICA
BUCARAMANGA
2004

A mi Madre, Quien ha hecho posible este logro y tantos otros.

A Mónica por su Amor y su incondicional compañía.

AGRADECIMIENTOS

El autor expresa sus agradecimientos a:

Profesor Enrique Sarmiento Moreno, director del presente proyecto.

Mónica Andrea Blanco, por su incondicional colaboración, su cariño y amor que ayudaron a sacar adelante este proyecto.

Mauricio Ortega, "Mao X" por su amistad, por su valiosa colaboración en la elaboración de varios de los algoritmos presentes en el proyecto, por sus consejos y por el tiempo que dedico a discutir aspectos del desarrollo del presente proyecto.

Camilo Andrés Vargas, por su colaboración en la elaboración de los modelos en 3D presentes en el sistema y por la elaboración de la interfaz grafica de este.

José Luís Sandoval, por su amistad, por sus consejos y por las incontables ocasiones en que escucho mis problemas y los problemas que tenían relación con el presente proyecto.

Javier Vivas, por su amistad, por su trabajo para mejorar las partidas de battletech y por compartir incontables horas dedicadas al juego de Battletech.

A los jugadores de Battletech:
Mauricio Ortega "Mao X", José Luís Sandoval, Javier Vivas, Mónica Andrea Blanco, Nicolás Cadavid, Alex Cruz, Jhon Jairo Lozada y Efraín Marino "El flaco".

CONTENIDO

INTRODUCCION 13
1. GENERALIDADES 16
1.1 OBJETIVO DEL PROYECTO. 16
1.2 PROBLEMA. 17
1.3 ESTRUCUTRA BASICA DE BATTLETECH. 17
1.4 JUSTIFICACIÓN 18
1.5 ANTECEDENTES. 19
1.6 METODOLOGÍA Y DESARROLLO DEL SISTEMA. 20
2. MARCO TEÓRICO Y METODOLOGICO 23
2.1 PROGRAMACIÓN ORIENTADA A OBJETOS 23
2.1.1 Historia. 23
2.1.2 El Modelo de Objetos. 25
2.1.3 Fundamentos del Modelo de Objetos. 26
2.1.4 Elementos del modelo de Objetos.. 27
2.1.5 Programación Orientada a Objetos (POO) 34
2.1.6 Diseño orientado a Objetos (DOO).. 35
2.1.7 Análisis orientado a objetos (AOO). 36
2.1.8 ¿Por que usar Programación Orientada a Objetos (POO)? 36
2.1.9 Clasificación de los lenguajes Orientados a Objetos. 38
2.1.10 Conceptos fundamentales de la POO 39
2.2 UML 48
2.2.1 Unified Modeling Language (UML) 48
2.2.2 El UML como lenguaje. 48
2.2.3 El UML es un lenguaje para Visualizar.. 49
2.2.4 El UML es un lenguaje para especificar.. 50
2.2.5 El UML es un lenguaje para construir. 50
2.2.6 El UML es un lenguaje para documentar. 51
2.2.7 Modelo conceptual del UML. 51
2.3 DELPHI. 65
2.4 OpenGL 65
2.4.1 Evolución del estándar.. 66
2.4.2 El ARB (Architecture Review Board) de OpenGL. 66
2.4.3 Cómo trabaja OpenGL.. 67
2.4.4 Implementaciones genéricas.. 67
2.4.5 Implementaciones hardware.. 68
2.5 METODOLOGÍA 69
3. DISEÑO 72
3.1 CASOS DE USO DE DISEÑO. 73

3.2 DIAGRAMA DE CLASES DISEÑO. 82
3.3 DISEÑO CLIENTE / SERVIDOR 86
3.4 DISEÑO AMBIENTE TRIDIMENCIONAL. 90
3.5 HERRAMIENTAS UTILIZADAS 93
4. DESARROLLO 94
4.1 METODOLOGIA 94
4.2 DESARROLLO DE LOS TRES PROTOTIPOS 98
4.2.1 Primer prototipo. 98
4.2.1.1 Fase de elaboración Primer Prototipo. 98
4.2.1.2 Fase De Construcción Primer Prototipo. 112
4.2.1.2.1 Análisis. 113
4.2.3.1 Clases persistentes que guardar el estado de la partida. 171
5. PRUEBAS FINALES 173
5.1 PRUEBAS PRIMER PROTOTIPO. 174
5.3 PRUEBAS TERCER PROTOTIPO. 179
6. CONCLUSIONES 181
7. RECOMENDACIONES 184
BIBLIOGRAFÍA 186

LISTA DE TABLAS

Tabla 1. Descripción documentación producida por EA 75
Tabla 2. Fragmento de archivos SMD de referencia y animación 166

LISTA DE FIGURAS

Figura 1. Clasificación de los Lenguajes Orientados a Objetos 39
Figura 2. Clases. 53
Figura 3. Interfaz. 54
Figura 4. Colaboraciones. 54
Figura 5. Casos de uso. 54
Figura 6. Clases activas. 55
Figura 7. Componentes. 55
Figura 8. Nodos. 56
Figura 9. Mensajes 56
Figura 10. Maquina de estado 57
Figura 11. Paquetes 57
Figura 12. Notas. 58
Figura 13. Dependencia. 59
Figura 14. Asociación. 59
Figura 15. Generalización 59
Figura 16. Realización. 60
Figura 20. Capas lógicas del sistema. 72
Figura 21. Casos de uso Diseño 73
Figura 22. Diagrama de Clases. 83
Figura 23. Capas lógicas del sistema. 84
Figura 24. Estados de partida. 85
Figura 25. Diagrama de estados del servidor 87
Figura 26. Estructura de un mensaje 88
Figura 27. Diagrama de despliegue* de la aplicación. 89
Figura 28. Mech con esqueleto a la vista. 91
Figura 29. Diagrama animación de movimientos 92
Figura 30. Metodologías. 95
Figura 31. Diagrama de casos de uso. 101
Figura 32. Casos de uso de análisis. 102
Figura 33. Modelo conceptual. 109
Figura 34. LOS. 111
Figura 35. Prototipo LOS, Distancia entre. 112
Figura 37. Diagrama de Casos de Uso 115
Figura 38. Transición de Estados para el ciclo de un juego. 120
Figura 39. Diagrama de Clases primer prototipo 122
Figura 40. Capas lógicas del sistema. 123
Figura 41. Estados de partida. 124
Figura 42. Hoja de Mech. 125
Figura 43. Armadura en hoja de mech 126
Figura 44. Componente Grafico de armadura. 127
Figura 45. Barras de calor 128
Figura 46. Tabla estructura crítica. 129
Figura 47. Componente estructura crítica 130
Figura 48. Inventario armas de la hoja de mech. 131

Figura 49. Componente para inventario armas, munición y ayudas. 131
Figura 50. Creador de mechs. 132
Figura 51. Empalme de mapas. 133
Figura 52. Guardar mapa. 134
Figura 53. Método son adyacentes. 136
Figura 54. Clase simple 137
Figura 55. Primer Prototipo 1. 140
Figura 56. Primer Prototipo 2. 140
Figura 57. Primer Prototipo 3. 141
Figura 58. Cliente de prueba. 147
Figura 59. Servidor de prueba. 147
Figura 60. Configuración posible de los mapas. 149
Figura 61. Diagrama de estados del servidor. 151
Figura 62. Estructura de un mensaje. 152
Figura 63. Diagrama Actividades para moverse. Fase Preliminar. 154
Figura 64. Diagrama de despliegue de la aplicación. 157
Figura 65. Keyrframe animation. 160
Figura 66. Keyframe interpolation. 161
Figura 67. Sharky. Creado por Camilo Vargas. 162
Figura 68. Mech con esqueleto a la vista. 163
Figura 69. Cilindro con esqueleto. 164
Figura 70. Brazo con esqueleto. 164
Figura 71. Visualizador de modelos SMD. 169
Figura 72. Diagrama animación de movimientos 170
Figura 73. Programa de prueba LOS y distancia entre. 175
Figura 74. Herramienta para visualizar modelos SMD. 178

TITULO: SISTEMA DE ENTRETENIMIENTO EN LINEA BASADO EN EL JUEGO DE ESTRATEGIA BATTLETECH[13].

AUTOR: MIGUEL ANGEL CAÑAS[**].

PALABRAS CLAVES

UML
Delphi
OpenGL.

CONTENIDO:

El sistema desarrollado es una herramienta capaz de recrear una partida del juego de estrategia BattleTech (marca registrada de FANPRO) de tal forma que parezca que los jugadores se encuentran frente al tablero de juego cuando en realidad están conectados desde lugares remotos (Internet o redes locales). El sistema esta en capacidad de aplicar todas las reglas necesarias para desarrollar la partida, de tal forma que puedan ser controlados los movimientos de los jugadores para que éstos se ajusten a las reglas del juego. Sumado a esto, el sistema permite guardar el estado de las partidas en caso de querer terminar una sesión de juego en otra oportunidad.

El sistema se desarrollo usando prototipado evolutivo como metodología, UML *(Unified Modeling Language)*, Delphi profesional versión cinco como lenguaje de desarrollo y OpenGL. Además se contó con la colaboración de un diseñador industrial para la elaboración de los modelos tridimensionales usados dentro de la aplicación.

Como herramientas adicionales se uso Enterprise Architect version 3.51 herramienta CASE, que permitió crear diagramas en UML, generar código desde estos diagramas en delphi y también permitió la actualización de algunos diagramas desde el código. También se uso 3ds max 5 como herramienta para la creación de los modelos tridimensionales usados dentro del sistema.

La elaboración de este sistema permitió demostrar que la filosofía de orientación a objetos usada con una metodología iterativa e incremental y apoyada en un lenguaje como UML permite dar solución a problemas complejos que poseen una gran cantidad de reglas.

[13] Trabajo de Grado
[**] Facultad de Ingenierías Físico mecánicas. Escuela de Ingeniería de Sistemas e Informática. Ingeniero Enrique Sarmiento Moreno.

TITLE: ENTERTAIMENT SYSTEM BASED IN THE STRATEGY GAME BATTLETECH[14].

AUTHOR: MIGUEL ANGEL CAÑAS CELY[**].

KEY WORDS:

UML
Delphi
OpenGL.

CONTENT:

The system developed is a tool able to recreate a complete match of the strategy game BattleTech (FANPRO's trademark) in a way that players think that they are in front of the game board while they are really using a remote connection (Internet or local network). The system is in capacity to apply the needed rules to develop a game so that all the movements that players will do fit the game rules. Besides these characteristics, the system permits to save the current state of the game to finish it in other session.

The system was developed using prototyping methodology, UML *(Unified Modeling Language)*, Delphi professional version 5 as a developing language and OpenGL. Besides this an Industrial Designer collaborates in the creation of the three-dimensional models used in the system.

As an additional tools were used Enterprise Architect version 3.51 a CASE tool that permitted the creation of UML diagrams and the generation of Delphi code from this diagrams. Also was used 3ds max 5 as a tool to create the models used into the system.

The development of this system showed that object oriented philosophy used with an iterative and incremental methodology and supported in UML permits give a solution to complicated problems that have a lot of rules.

---

[14] Degree Project.

[**] School of Physique Mechanic Engineering, Department of System Engineering. Engineer Enrique Sarmiento Moreno.

INTRODUCCION

El sistema de entretenimiento en línea basado en el juego de estrategia battletech, se desarrolló con el objetivo de aplicar ingeniería del software a un sistema complejo, mas exactamente a un sistema que debe aplicar una gran variedad de reglas.

Battletech[15] es un juego de estrategia inicialmente desarrollado por FASA Corporation, quien luego vendió los derechos de éste a FANPRO, una filial de la conocida compañía Norteamericana de juegos WIZKIDS[16]. Este juego es un juego de estrategia que se basa en un mundo de ciencia ficción en el cual los campos de batalla están dominados por robots gigantescos fuertemente armados. El objetivo de este juego es recrear estas batallas colosales en las que participan poderosos robots utilizando un sistema reglas complejo contenido en un libro de 47 páginas.

El sistema desarrollado esta en capacidad de permitir a dos o más jugadores, conectados a través de una red de computadores (local o Internet), participar en un juego de battletech. En caso de que la partida no sea terminada, ya que estas suelen extenderse por horas, el sistema esta en capacidad de guardar el estado de la partida que se esta jugando para que sea posible terminarla en varias sesiones de juego.

Para recrear las partidas el sistema utiliza un motor de reglas y una interfaz grafica en 2D y 3D. Para la interfaz gráfica en 2D se utilizaron componentes especialmente diseñados para la aplicación. Para la interfaz en 3D, que es la que representa el tablero, los robots y sus animaciones se utilizó OpenGL. La elaboración de estos modelos tridimensionales y sus animaciones se hizo en 3DS Max versión cinco y como formato para guardar cada uno de estos modelos se utilizo el formato de archivo .SMD utilizado por el juego Half-Life, de la compañía ValveSfotware[17], para guardar sus modelos dentro de este juego.

---

[15] www.classicbattletech.com
[16] www.wizkids.com
[17] www.valvesoftware.com

Las animaciones y los diseños tridimensionales se realizaron con la ayuda de un diseñador industrial, quien también colaboró en el diseño de la interfaz en 2D final de la aplicación.

La metodología utilizada para el desarrollo del presente proyecto, fue prototipado evolutivo. Esta metodología sufrió cambios durante el desarrollo del proyecto y se vio influenciada por el proceso unificado. Estas modificaciones a la metodología se realizaron para facilitar el desarrollo de éste y para lograr, de manera sencilla, la integración de UML con la metodología planteada, lenguaje escogido como herramienta adicional para la elaboración del presente proyecto.

La filosofía de programación utilizada fue la de la orientación a objetos durante el desarrollo de todo el proyecto, usando como herramienta de desarrollo el programa Delphi profesional en su versión 5.

Además de esta herramienta para implementación, se utilizo la herramienta CASE Enterprise Architect en su versión 3.51. Esta herramienta sirvió para realizar diagramas en UML durante las etapas de análisis y diseño, permitió la documentación y generación de código a partir de estos diagramas y la actualización de algunos diagramas UML desde el código realizado en Delphi.

Durante la elaboración del proyecto, se contó con la colaboración del Diseñador Industrial Camilo A. Vargas y la colaboración de varios jugadores de Battletech pertenecientes al grupo de juego del autor.

El presente documento esta distribuido de la siguiente manera:

El capitulo 1 contiene las generalidades del proyecto.

El capitulo 2, marco teórico, habla sobre la programación orientada a objetos, UML, Delphi OpenGL y sobre el prototipado evolutivo, metodología utilizada para el desarrollo del presente proyecto. En este capitulo se hace especial énfasis en la programación orientada a objetos, ya que esta es parte fundamental de este.

El capitulo 3 muestra el diseño completo del sistema, los casos de uso, los diagramas de clases del sistema, el diseño del sistema cliente/servidor, el diseño de los modelos tridimensionales y habla sobre las herramientas utilizadas durante el desarrollo del presente proyecto.

El capitulo 4 muestra el desarrollo del sistema de manera detallada. En la parte inicial de este capitulo se habla sobre los cambios que se introdujeron al prototipado evolutivo durante el desarrollo del sistema, para luego mostrar con detalle cómo se pasó por cada una de las fases de la metodología durante el desarrollo del primer prototipo. En las secciones que hablan del segundo y tercer prototipo se hace énfasis en el desarrollo del sistema de comunicación para el segundo prototipo, y para el desarrollo del tercero se muestra el trabajo realizado para lograr la interfaz tridimensional del sistema.

El capitulo 5 trata sobre las pruebas que se realizaron luego de tener listos cada uno de los tres prototipos que se plantearon como metas durante el desarrollo del proyecto.

El capitulo 6 muestra las conclusiones obtenidas debido al desarrollo del proyecto.

El capitulo 7 contiene las recomendaciones para darle continuidad al presente trabajo.

1. GENERALIDADES

1.1 OBJETIVO DEL PROYECTO.

El Objetivo de este proyecto es desarrollar un sistema capaz de recrear una partida del juego de estrategia BattleTech de tal forma que parezca que los jugadores se encuentran frente al tablero de juego cuando en realidad están conectados desde lugares remotos, bien sea mediante una red local o Internet. El sistema está en capacidad de aplicar las reglas necesarias para desarrollar una partida, de tal forma que puedan ser controlados los movimientos y las acciones de los jugadores para que se ajusten a éstas. El sistema declarará ganador al bando que logre la destrucción de su enemigo.

El sistema está en capacidad guardar el estado de las partidas, para permitir su culminación en varias sesiones. Debido a que las partidas se pueden tornar bastante largas, dependiendo del número de BattleMechs y del campo de batalla, el sistema permite guardar el estado de estas en caso de querer terminar una sesión de juego en otra oportunidad.

El sistema recrea una partida de BattleTech dentro de un mundo tridimensional, ciñéndose a las reglas estipuladas para este juego. Además de permitir controlar los BattleMechs y llevar a cabo una partida, se ha hecho un gran esfuerzo en condensar toda la información necesaria en una interfaz grafica amigable y de fácil manejo para aquellos jugadores que ya conocen las reglas básicas, y así permitir no solo estar dentro de una partida, sino conseguir que los jugadores se centren en la estrategia y en el mejoramiento de su estilo de juego. Gran parte del esfuerzo para permitir una interfaz amigable está en el cambio del tablero de juego plano del juego original por una imagen tridimensional del campo, al igual que de los robots que intervienen en la partida.

1.2 PROBLEMA.

El desarrollo de este sistema comienza luego de identificar la necesidad existente entre la comunidad de jugadores de Battletech de poder ampliar la cantidad de jugadores con los que se podía participar en una partida. Se conocía la existencia de clubes de jugadores en Bogotá y Medellín y de un club mundial de jugadores que realizaban torneos en Estados Unidos, algunas zonas de Europa principalmente y un torneo mundial anual. Lamentablemente no todos los adeptos a este juego están en capacidad de viajar a los lugares donde se realizan este tipo encuentros, así que como respuesta a este problema nace la idea de desarrollar este sistema que brinda la posibilidad de jugar una partida de este famoso juego en línea con jugadores conectados desde lugares remotos a través de Internet.

1.3 ESTRUCUTRA BASICA DE BATTLETECH.

BattleTech[18] es un juego que está clasificado como un juego de estrategia. Estos juegos se diferencian de los clásicos juegos de mesa porque en este caso no hay que llevar una pequeña pieza por un camino determinado y esperar que el azar determine al ganador. BattleTech implica un movimiento concienzudo de las piezas sobre un campo de batalla, una retícula hexagonal, para así lograr alcanzar un objetivo, que puede variar entre destruir al enemigo (el más común) hasta el rescate o la destrucción de una persona considerada objetivo militar. Además de esto, el sistema de reglas es mucho mas complejo y extenso, tanto como para llenar varios libros.

BattleTech es un juego que se basa en un mundo de ciencia-ficción en el cual para el siglo XXXI el hombre habrá conquistado el espacio y habrá creado unas máquinas increíbles llamadas BattleMechs, que son unos robots de unos 12 metros de altura que lideran las guerras de ese tiempo. Cada jugador está en capacidad de controlar una o varias de estas maquinas dentro de cada partida del juego. Para controlar cada una de estas máquinas, el jugador usa una hoja de papel tamaño carta que contiene la información para hacer que estas máquinas existan dentro del juego y se diferencien unas de otras. Dentro de esta información se encuentra el peso, el movimiento, la armadura que posee, las armas que tiene e información sobre piezas que se consideran importantes porque permiten el funcionamiento del robot, como por ejemplo el motor, entre otras.

[18] http://www.BattletechClassic.com

El juego se desarrolla entre dos bandos con un número de robots limitado solo por las características del escenario que se quiera jugar o por el espacio de juego. Además de la hoja que representa la máquina de cada jugador, también se utiliza una ficha de cartón o plástico, que representa la máquina de guerra para moverla sobre la retícula hexagonal que representa el campo de batalla. El movimiento de las piezas sobre el campo de batalla está regido por las reglas del movimiento; dependiendo de la máquina escogida se realizará el movimiento, parecido al ajedrez, solo que las posibilidades de éste son las mismas para todos, lo que es limitado es la cantidad de espacio recorrido y la forma como lo recorre (se puede caminar, correr o saltar), dependiendo de la máquina.

Otro elemento importante del juego es el azar; para resolver las situaciones del combate se utiliza un par de dados de 6 caras. Dependiendo de la situación en el campo del jugador atacante y de los jugadores a los que se atacará, se calcula un número que está en el rango entre 2 y 12, que es la base; entonces el jugador atacante lanza los dados (dos dados de seis caras) y si obtiene un numero mayor o igual al calculado, su acción es exitosa (esta acción puede ser un ataque con armas, un chequeo de habilidad, etc). Como estos números se calculan dependiendo de las acciones realizadas durante el movimiento, entra en juego la estrategia que se escogió para moverse y obtener una posición ventajosa.

Otras acciones dentro del juego se realizan de la misma forma, combinando la estrategia con el azar y el BattleMech (robot) escogido, que puede generar ventajas o desventajas dependiendo de la situación en el juego.

Para ampliación de las reglas del BattleTech ver anexo A.

1.4 JUSTIFICACIÓN.

Después de reconocer la necesidad existente entre la comunidad local de jugadores, se identifica el presente proyecto como una aplicación que une varias tecnologías informáticas que no se encuentran en otras aplicaciones, que permite revisar las teorías de desarrollo de software y aplicar los conocimientos aprendidos durante el curso de la carrera y también se identifica la posibilidad de que la escuela de Ingeniería de Sistemas e Informática de la Universidad Industrial de Santander

empiece a desarrollar proyectos enfocados al entretenimiento, que es una de las áreas más productivas y con más desarrollo en la época actual en el mundo.

Además de permitir aplicar varias tecnologías informáticas, el desarrollo del presente proyecto también representa una buena oportunidad para aplicar ingeniería del software a la solución de problemas complejos, como los que tienen muchas reglas. Para este caso el juego escogido es un juego de estrategia que tiene un sistema de reglas extenso y complejo ya definido.

1.5 ANTECEDENTES.

Antes de empezar el desarrollo de este sistema se buscó una solución similar a la que se planteaba dentro de este proyecto en la comunidad de jugadores, dentro de las industrias de video juegos independientes o reconocidas y se encontraron dos enfoques diferentes que pretendían atacar este problema.

El primer enfoque tenía que ver con dar solución solamente al problema de realizar partidas con jugadores ubicados en lugares distantes y unirlos en partidas en red. La solución que se presenta en varias comunidades de Internet es la de la realización de partidas enviando información vía correo electrónico, acogiéndose a unos estándares previamente definidos para el lanzamiento de dados y el desarrollo de la partida. Otra solución muy parecida a esta y de mayor aceptación, tenía que ver con la realización de partidas en salones de Chat, principalmente en salones de Chat IRC[19] utilizando bots (palabra utilizada para designar programas que realizan cierta funcionalidad automáticamente, proviene de robot) como controladores de algunas reglas y con la ayuda de alguno de los participantes como arbitro de las partidas.

El otro enfoque no tiene que ver con la creación de un sistema que recree las reglas de juego, sino se centra en recrear la experiencia de lo que podría ser pilotar una de estas máquinas si de verdad existieran. En esta área se ha creado una saga de juegos llamados Mechwarrior, actualmente en la versión 4, producidos por Microsoft.

[19] Internet Relay Chat. Protocolo de Chat.

Teniendo en cuenta lo anterior, no se había atacado el problema de la forma como se está planteando en este proyecto, ya que la experiencia de jugar a través de correo o en un salón de Chat no se puede comparar con lo planteado con el desarrollo de este sistema.

1.6 METODOLOGÍA Y DESARROLLO DEL SISTEMA.

Para el desarrollo de este sistema se utilizó como metodología el prototipado incremental, planteando tres prototipos con características muy especificas que permitieron llegar al producto final. Durante el desarrollo del proyecto se mejoró la metodología originalmente escogida con aspectos del Proceso Unificado que enriquecieron el análisis y el diseño, y permitieron realizar algunos prototipos intermedios para poder perfeccionar el sistema. Además del enriquecimiento de los procesos, el proceso unificado también suministró algunos lineamientos para la elaboración de los diagramas necesarios para el desarrollo del proyecto y para la documentación del sistema en UML, lenguaje que se estaba utilizando desde que se planteó el proyecto.

Como complemento para el uso de UML se utilizó la herramienta CASE Enterprise Architect, que permitió no solo realizar los diagramas, sino que también permitió la documentación y la generación de código a partir de diagramas y la actualización de estos desde el código realizado en Delphi.

Todo el sistema se desarrolló con la filosofía de la orientación a objetos, sin la cual hubiera sido bastante difícil encarar el problema de recrear la partida, ya que las reglas del juego ocupan un libro de 47 paginas. Estos requerimientos son muy específicos y contemplan varias posibilidades no muy fáciles de predecir, así que la mejor forma para atacar este problema fue el análisis y el diseño orientado a objetos, ya que permitió que todo fuera más intuitivo y parecido a la realidad. Por ejemplo, una de las clases fundamentales sobre las que gira el sistema es la clase TMech que como es fácil de imaginar contiene todos los métodos y atributos que permiten que un robot ataque, sea atacado y se mueva sobre el mapa, cuya clase es TMapa, que también es de gran importancia.

Además de tener estos objetos en tiempo de ejecución de la partida, algunas clases son persistentes, como la clase Tmech o la clase Tmapa, lo que permite guardar directamente los objetos creados ya sea para tener estos objetos disponibles para guardar partidas anteriores o para que estén listos al

momento de iniciar una nueva partida. La creación de estos objetos y su consiguiente diagrama en UML, permitió hacer un seguimiento al sistema durante todo el desarrollo de una manera fácil, sin tener que estar revisando de manera periódica el código ya realizado; además la encapsulación de la información permitió "olvidar" casi por completo partes de código que se habían realizado con anterioridad y que fueron ampliamente probadas. Uno de estos ejemplos es la clase Tmech que contiene unos 57 métodos y que de no ser por esta abstracción y la ayuda de UML hubiera sido bastante difícil seguir el comportamiento de sus objetos y permitir la escalabilidad necesaria para aumentar el grado de complejidad a medida que se pasaba de un prototipo a otro. Otro de los buenos ejemplos de la exitosa combinación entre el prototipado y los objetos es la clase Tmapa, que tuvo métodos y atributos implementados y funcionando al final de la etapa de análisis, debido a que se identificaron problemas que parecían difíciles de solucionar en ese momento, y gracias a estos desarrollos se evitaron posibles cuellos de botella que retrasarían el trabajo siguiente.

Al implementar esta clase de forma temprana se pudieron realizar pruebas exhaustivas de tal forma que después de esto lo único importante era conocer cómo crear este objeto y cómo utilizar sus métodos y se "olvidó" la manera como estaban implementados (abstracción y encapsulación de la información). Otro ejemplo importante de las ventajas de la programación orientada a objetos de resaltar en el desarrollo del sistema fue la creación de un método de la clase Tmapa que se probó por completo por fuera del sistema después de haber detectado sus existencia y luego de realizado el algoritmo se introdujo dentro de la clase sin crear ningún traumatismo en las demás clases que interactuaban con esta y sin necesidad de cambiar código en diferentes partes del sistema, ya que de antemano se había hecho un análisis de los parámetros que debía recibir el método y de los que debía retornar (interfaz del método). Todo lo anterior no se habría podido realizar de esta forma si no se hubiera escogido un lenguaje de desarrollo que permitiera la programación orientada a objetos como Delphi.

El desarrollo de este sistema permitió mostrar a la comunidad que hacer un sistema de entretenimiento no es una tarea fácil y que es un muy buen espacio para aplicar lo aprendido durante el pregrado. El desarrollo de este tipo de aplicaciones permite que se pongan en práctica conocimientos de cálculo, álgebra, ingeniería del software, programación, bases de datos, entre otros y obliga al realizador a escoger muy bien sus herramientas de desarrollo, así como a investigar sobre tecnologías nuevas, ya que el entretenimiento es una de las áreas que avanza con mas rapidez dentro de la industria del software. Sumado a lo anterior, también permitió explorar otras áreas que

están un poco olvidadas dentro de la ingeniería del software, como es el caso del desarrollo de interfaces gráficas que para este caso se apoyó en los consejos y la asesoría de un diseñador industrial.

El desarrollo de este sistema permitió comprobar la utilidad y el poder de la programación orientada a objetos en el momento de resolver problemas de gran complejidad de una forma satisfactoria y permitiendo que sean escalables y fáciles de entender, no solo para los desarrolladores del sistema sino también para las personas que entren en contacto con el sistema y para los usuarios finales. También se confirmó que tener un lenguaje único (UML) utilizado para el análisis, diseño y desarrollo de la herramienta es de gran utilidad no solo para documentar lo que se quiere hacer y lo que se ha hecho, sino también para poder encontrar posibles fallas y para permitir que otros profesionales brinden asesoría sin necesidad de conocer por completo la herramienta que se está desarrollando; no obstante, un aspecto negativo que puede tener UML, es que en algunos momentos el desarrollador debe detenerse y dejar de hacer diagramas de todo lo que se está haciendo, porque se puede caer en el exceso y terminar utilizando mucho tiempo desarrollando diagramas que pueden llegar a ser más detallados de lo necesario; es importante encontrar un equilibrio entre detalle y utilidad para no quedar estancado en el diseño, análisis o documentación solamente.

Además de lo anterior se logró demostrar que el prototipado evolutivo como metodología de desarrollo de software aplicada de manera disciplinada y apoyada en herramientas de desarrollo de software (UML, Delphi), es capaz de dar solución de manera satisfactoria a problemas complejos con una gran cantidad de reglas. También se demostró que en el desarrollo actual de software es muy importante lograr la interacción de varios profesionales para lograr resultados provechosos; para este caso se logró la interacción entre los conocedores del juego (jugadores experimentados), un diseñador industrial y un ingeniero de sistemas.

2. MARCO TEÓRICO Y METODOLOGICO

2.1 PROGRAMACIÓN ORIENTADA A OBJETOS

2.1.1 Historia. Al principio de la década de los sesenta, Kristen Nygaard y Ole-Johan Dhal desarrollaron SIMULA en el Norwegian Computer Center (NCC), un lenguaje que soportaba modelado para simulación de procesos industriales y científicos. Fue un lenguaje que ofrecía la capacidad de simulación de sistemas. Sin embargo, sus usuarios descubrieron pronto que SIMULA proporcionaba nuevas y poderosas posibilidades para fines distintos de la simulación, como la elaboración de prototipos y el diseño de aplicaciones. Las sucesivas versiones de Simula fueron mejorándose hasta llegar a Simula-67 (apareció en diciembre de 1966). Simula-67 se deriva de Algol-60, donde tiene sus raíces, y se diseñó fundamentalmente como un lenguaje de programación de diseño; tomó de Algol el concepto de bloque e introdujo el concepto de Objeto. Los objetos de Simula tenían su propia existencia y podían comunicarse entre si durante un proceso de simulación.

Conceptualmente, un objeto tenía tanto datos como las operaciones que manipulaban esos datos. Las operaciones se llamaban métodos. Simula incorporaba también la noción de clases, que se utilizaron para describir la estructura y el comportamiento de un conjunto de objetos. La herencia de clases también fue soportada por Simula. La herencia organiza las clases en jerarquías, permitiendo compartir implementación y estructura. En esencia Simula sentó la base de los lenguajes orientados a objetos y definió algunos de los conceptos clave de la orientación a objetos. Además, Simula era un lenguaje 'fuertemente tipificado'. Esto significa que el tipo de cada variable se conoce a tiempo de compilación, de modo que los errores que implican tipos se encuentran en esta etapa y no cuando el programa se ejecuta.

Simula-67 fue el primer lenguaje de programación que incorporó mecanismos para soportar los conceptos orientados a objetos más sobresalientes:

- *Encapsulamiento y objetos*, que agrupan juntos atributos de datos y acciones (métodos) que procesan esos datos.
- *Verificación estática de tipos*, que se realiza durante el proceso de compilación para proporcionar seguridad en tiempo de ejecución para la manipulación externa de los atributos de los objetos.
- *Clases*, como plantillas o patrones de los objetos
- *Herencia*, como medio de agrupar clases con propiedades comunes.
- *Ligadura dinámica (Polimorfismo),* para permitir a las clases de objetos que tengan interfaces idénticas y propiedades que se puedan intercambiar.

Alan Kay creó Smalltalk-80 en Xerox PARC (Xerox Palo Alto Research Park) con Adele Goldberg, que previamente había trabajado con una implementación de Simula. Smalltalk es un lenguaje orientado a objetos *Puro*. '*Todo es un objeto de una clase*' y todas las clases heredan de una única clase *Object*. Smalltalk afirmó el término *método*, para describir las acciones realizadas por un objeto y el concepto de *paso de mensajes* como el medio para activar *métodos*. Es también un lenguaje tipificado dinámicamente, que *liga* (*enlaza*) un método a un mensaje en tiempo de ejecución.

A partir de este momento Smalltalk fue el onspirador de un gran numero de lenguajes orientados a objetos, entre ellos Eiffel, Smalltalk-80, Smalltalk/V, C++, Actor, Objective-C y CLOS, así como extensiones OO de lenguajes tradicionales, tales como Object-Pascal, Object-Cobol entre otros.

Durante la década de los ochenta, los conceptos orientados a objetos (tipos abstractos de datos, herencia, identidad de objetos y concurrencia), Smalltalk, Simula y otros lenguajes comenzaron a mezclarse y producir nuevos lenguajes orientados a objetos, así como extensiones y dialectos. El desarrollo de lenguajes orientados a objetos se muestra en la siguiente clasificación:

- *Extensiones y dialectos de Smalltalk.*
- *Extensiones orientadas a objetos de lenguajes tradicionales*. C++ (Stroustrup AT&T, 80's), Objective-C, Object-Pascal (Niklaus Wirt y su grupo, Apple Computer) y los nuevos lenguajes ADA-95 y Java (totalmente orientados a objetos).
- *Lenguajes Orientados a objetos fuertemente tipificados*. Simula (80's). Eiffel (Interactive Software Inc), ADA.
- *Extensiones orientadas a objetos de LISP*.

La década de los ochenta lanzo la orientación a objetos como la base de la futura ingeniería de software orientada a objetos de la década de los noventa. En 1982 se predijo que la programación orientada a objetos seria en la década de los ochenta lo que la programación estructurada fue en los setenta (Rentsch, 1982). La profecía se cumplió y la década de los ochenta se consagró como el origen de la explosión de la orientación a objetos que se produciría en los noventa.

Sin duda, el desarrollo de conferencias internacionales sobre orientación a objetos (como OOPSLA, Object-Oriented Programing Systems and LAnguajes) al igual que la creación de publicaciones periódicas (como The journal of Object-Oriented Programing) sobre el tema han influido en el enorme desarrollo de los Lenguajes de Programación Orientados a Objetos (LPOO).

En la década de los noventa los lenguajes, técnicas, interfaces gráficas y bases de datos se hicieron muy populares. Los lenguajes más implantados en la actualidad son Smalltalk y Eiffel, junto con C++, Object-Pascal, Visual BASIC y Object Visual como los lenguajes híbridos y JAVA como un híbrido propio de Sun que reúne las propiedades de C++, ADA-95, Object-Pascal, Smalltalk, etc., diseñado para Internet con propiedades de objetos.

2.1.2 El Modelo de Objetos. El diseño orientado a objetos se constituye bajo un fundamento de ingeniería, cuyos elementos son llamados colectivamente el modelo de objetos. El modelo de objetos abarca los principios de la abstracción, de la encapsulación, de la modularidad, de la

jerarquía, de la concurrencia y de la persistencia por si mismos, ninguno de estos principios es nuevo. Lo que es importante acerca del modelo de objetos es que estos elementos son unidos de una manera sinergética.

No hay duda de que el diseño orientado a objetos es fundamentalmente diferente de los otros enfoques de diseño tradicionales: requiere una manera de pensamiento acerca de la descomoposición y está ampliamente por fuera del dominio de otros métodos de diseño. Estos hechos aparecen debido a que el diseño orientado a objetos se construye bajo la programación orientada a objetos.

2.1.3 Fundamentos del Modelo de Objetos. Los métodos de diseño estructurado evolucionaron para guiar a los desarrolladores quienes estuvieron intentando construir sistemas complejos usando algoritmos como sus bloques de construcción fundamentales. Similarmente, los métodos de diseño orientado a objetos han evolucionado para ayudar a los desarrolladores a explotar el poder expresivo de los lenguajes de programación basados en objetos y orientados a objetos, usando la clase y el objeto como los bloques de construcción básicos.

Debido a que el modelo de objetos ha sido influenciado por un gran número de factores y a que ha demostrado ser un concepto unificador en el mundo de la ciencia de la computación, aplicable no solo a los lenguajes de programación sino al diseño de interfaces, a bases de datos, a bases de conocimiento, y aun a arquitecturas de computador, se ha extendido ampliamente como uno de los modelos capaces de enfrentar la complejidad inherente de los sistemas.

Como dice Booch "el diseño orientado a objetos representa un desarrollo evolucionista, no uno revolucionista; no rompe con los avances del pasado, sino que construye sobre los probados"[20]

[20] BOOCH, Grady. Análisis y Diseño Orientado a Objetos con aplicaciones. Segunda Edición. Addison-Wessley / Diaz de Santos. E.U.A. 1996.

2.1.4 Elementos del modelo de Objetos. La programación orientada a objetos (POO) se suele conocer como un nuevo paradigma de programación. Otros paradigmas de programación conocidos son: el paradigma de la programación imperativa (Pascal, C), el paradigma de la programación lógica (PROLOG), el paradigma de la programación funcional (LISP). Un paradigma se describe como un conjunto de teorías, estándar y métodos que juntos representan un medio de organización del conocimiento, es decir un medio de visualizar el mundo. En este sentido la programación orientada a objetos es un nuevo paradigma. La orientación a objetos fuerza a reconsiderar nuestro pensamiento sobre la computación, sobre lo que significa realizar computación y sobre como se estructura la información dentro de la computadora.

La mayoría de los programadores utilizan un estilo de programación forzado por el lenguaje que están utilizando y la mayoría de las veces son incapaces de ver las ventajas y desventajas de utilizar este estilo para resolver sus problemas específicos debido a que no se enfrentan con nuevos o diferentes estilos de programación. Bobrow y Setif definen un estilo de programación como "un medio de organización de programas sobre la base de algún modelo conceptual de programación y un lenguaje apropiado para hacer programas en un estilo claro". Sugieren que existen cinco estilos de programación:

1. Orientados a procedimientos *Algoritmos*
2. Orientados a Objetos *Clases Objetos*
3. Orientación a lógica *Expresado en cálculo de predicados*
4. Orientación a reglas *Reglas* if-then *(si-entonces)*
5. Orientados a Restricciones *Relaciones invariables*

No existe un lenguaje de programación idóneo para todas las clases de programación. La orientación a objetos se acopla a la simulación de situaciones del mundo real.

En POO (Programación Orientada a Objetos), las entidades centrales son Objetos, que son tipos de datos que encapsulan con el mismo nombre estructuras de datos y las operaciones o algoritmos que manipulan esos datos.

La orientación a objetos puede describirse según Joyanes como "el conjunto de disciplinas (ingeniería) que desarrollan y modelizan software que facilitan la construcción de sistemas complejos a partir de componentes"[21]. El principal atractivo de la POO es que se logra una representación más fiel del mundo real. Las ventajas de la POO son muchas en programación y modelado de datos. Como apuntan Ledbetter y Cox (1985): "La programación Orientada a Objetos permite una representación más directa del modelo del mundo real en el código. El resultado es que la radical transformación normalmente llevada a cabo de los requisitos del sistema (definido en términos de usuario) a la especificación del sistema (definido en términos de computadora) se reduce considerablemente."

Los conceptos y herramientas orientadas a objetos permiten que los problemas del mundo real sean expresados de un modo más fácil y de manera más natural. Las técnicas orientadas a objetos proporcionan mejoras y metodologías para construir sistemas de software complejos a partir de unidades de software modularizado y reutilizable.

El soporte fundamental de la POO es el *Modelo Objeto*, según Grady Booch los cuatro elementos (propiedades) principales de este modelo son:

1. Abstracción
2. Encapsulación
3. Modularidad
4. Jerarquía [22]

Principales quiere decir que un modelo sin cualquiera de estos elementos no es un modelo Orientado a Objetos. Además de estos cuatro elementos fundamentales, otros autores (Joyanes,

[21] JOYANES AGUILAR, Luis. Programación Orientada a Objetos. Segunda Edición. McGraw Hill. España 1998. pag 18.
[22] BOOCH, Grady. Análisis y Diseño Orientado a Objetos con aplicaciones. Segunda Edición. Addison-Wessley / Diaz de Santos. E.U.A. 1996.

Ege) agregan una quinta propiedad bastante significativa del *modelo objeto*, esta es:

5. Polimorfismo

Hay tres elementos menores de este modelo de objetos:

1. Tipaje
2. Concurrencia
3. Persistencia

Sin embargo, algunos autores (como Joyanes) hablan también de otros elementos menores del modelo de objetos, estos son:

4. Genericidad
5. Manejo de excepciones

Menores quiere decir que cada uno de estos elementos es una parte útil pero no esencial del modelo de objetos. Es importante destacar que algunos autores (como Raimund Ege) no presentan de una manera formal dentro del modelo de objetos estos elementos menores. Para ellos solo los elementos principales son importantes.

Sin esta estructura se podría estar programando en un lenguaje Orientado a Objetos (Object-Pascal, C++, SmallTalk) sin estar utilizando todas las capacidades del modelo de Objetos. En este caso solo se estaría haciendo uso de la potencia expresiva del lenguaje OO o basado en objetos que se está usando para la implementación.

2.1.4.1 Abstracción. Según Booch "una Abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos y por lo tanto proporciona fronteras conceptuales frágilmente definidas, relativas a la perspectiva del visualizador"[23]

[23] BOOCH, Grady. Op cit.

Lo que podría definirse de una manera más sencilla como la propiedad que permite representar las características esenciales de un objeto sin preocuparse de las restantes características no esenciales. Una abstracción se centra en la vista externa del objeto, de tal forma que sea posible separar el comportamiento esencial de un objeto de su implementación.

El concepto de abstracción en los lenguajes de programación no es nuevo, lo que sí es innovador es el uso de clases para gestionar estas abstracciones dentro de los lenguajes de programación.

2.1.4.2 Encapsulación. El Encapsulamiento o la encapsulación es la propiedad que permite asegurar que el contenido de la información de un objeto está oculto de un mundo exterior. Esto quiere decir que el objeto A no conoce lo que hace el Objeto B y viceversa. La encapsulación (también conocida como ocultamiento de la información) es en esencia el proceso de ocultar todos los secretos que no contribuyen a sus características esenciales.

La encapsulación permite la implementación de los módulos. Estos módulos se implementan mediante clases, de forma que una clase represente la encapsulación de una abstracción. En la práctica esto significa que una clase debe tener dos partes, una interfaz, la que captura su vista externa, y la implementación que contiene la representación de la abstracción, así como los mecanismos que realizan el comportamiento deseado.

2.1.4.3 Modularidad. Según Liskov "la Modularidad consiste en dividir un programa en módulos que pueden ser compilados separadamente, pero que tengan conexiones con otros módulos". O de una manera más sencilla, la Modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas llamadas módulos, cada una de las cuales debe ser tan independiente de las otras como sea posible. La Modularidad está ampliamente relacionada con la abstracción y la encapsulación y cada uno de los lenguajes orientados a objetos la implementan de manera diferente.

En la etapa de diseño la escogencia de los módulos en algunas ocasiones está relacionada con los equipos de trabajo o en otras se crean los módulos con la idea de reutilizarlos en otras aplicaciones. Para finalizar, Booch[24] agrega que esta descomposición debe hacerse en un conjunto de módulos cohesivos y ligeramente acoplados.

La importante aquí es no perder el punto de vista. Encontrar las clases y los objetos correctos y luego separarlos en módulos, son decisiones de diseño enteramente independientes. La identificación de clases y objetos hace parte del diseño lógico mientras que el diseño de los módulos hace parte del diseño físico. Sin embargo no es posible tomar primero todas las decisiones de diseño lógico antes de las físicas o viceversa, este es más un proceso iterativo.

2.1.4.4 Jerarquía. Booch define la jerarquía así: "La jerarquía es el ranking u ordenamiento de las abstracciones"[25].

Las dos jerarquías más importantes de un sistema son: Estructura de clases (jerarquía es-un (is-a): Generalización/Especialización) y estructura de objetos (jerarquía es parte de (part-of): Agregación).

La jerarquía de Generalización/Especialización se conoce como herencia. Básicamente la herencia se define como una relación entre clases, en donde una clase comparte la estructura y el comportamiento definido en una o más clases (herencia simple y herencia múltiple).

La agregación es el concepto que permite el agrupamiento físico de estructuras relacionadas lógicamente.

2.1.4.5 Polimorfismo. Aunque algunos autores no consideran esta propiedad como fundamental, es bastante importante y por eso es considerada en este caso. En otras ocasiones se considera el polimorfismo como un resultado de la ligadura tardía o postergada (como lo hace

[24] BOOCH, Grady. Análisis y Diseño Orientado a Objetos con aplicaciones. Segunda Edición. Addison-Wessley / Diaz de Santos. E.U.A. 1996.
[25] Ibid.

Ege[26]), mientras que Booch considera el polimorfismo un resultado del Tipaje dinámico que para nuestro caso es lo mismo que ligadura tardía o dinámica.

El polimorfismo es la propiedad que indica la posibilidad de que una entidad tome muchas formas. En términos prácticos, el polimorfismo permite referirse a objetos de clases diferentes mediante el mismo elemento del programa y realizar la misma operación de diferentes formas, según sea el objeto que se refiere en ese momento.

Clases, herencia y polimorfismo son aspectos claves de la POO, y se reconoce a estos elementos como esenciales en la misma. El polimorfismo adquiere su máxima expresión mediante la propiedad de derivación de clases o herencia. Así por ejemplo si se dispone de una figura que represente figuras geométricas genéricas, se puede enviar cualquier mensaje, tanto a un tipo derivado (elipse, circulo, cuadrado, etc.) como al tipo base. Por ejemplo una figura puede aceptar los mensajes dibujar, borrar y mover. Cualquier tipo derivado de una figura es un tipo de figura y puede recibir el mismo mensaje. Cuando se envía un mensaje, por ejemplo dibujar, esta tarea será distinta según que la clase sea un triangulo, un cuadrado, un circulo, etc. El polimorfismo es la propiedad que permite que una función se comporte de manera diferente dependiendo de la clase sobre la que se aplica. La función dibujar se aplicará igualmente en un círculo, un cuadrado o un triangulo y el objeto ejecutará el código apropiado dependiendo del tipo especifico.

El polimorfismo requiere ligadura tardía o postergada (también llamada dinámica), y esto significa que el programa no puede determinar la dirección de código a ejecutar hasta el tiempo de ejecución. Cuando se envía un mensaje a un objeto, el código que se llama no se determina hasta el momento de ejecución. El compilador se asegura que exista la función y hace la comprobación de tipos de los argumentos y del valor de retorno, pero no conoce el código exacto a ejecutar.

Para realizar la ligadura tardía, el compilador inserta código especial en lugar de la llamada absoluta (así lo haría un lenguaje que no soporte ligadura tardía). Este código calcula la dirección en tiempo

[26]EGE, Raimund K. Programing in an object-oriented enviroment. Academic Press Inc. United Kingdom. 1992. 300p

de ejecución utilizando información almacenada en el propio objeto. Por consiguiente cada objeto se puede comportar de manera diferente de acuerdo al contenido de ese puntero.

2.1.4.6 Tipaje. Tipificación es el proceso de declarar el tipo de información que puede contener una variable. Se dice que un lenguaje es fuertemente tipificado si los errores de programación relacionados con el número de parámetros, tipos de parámetros e interfaces de módulos se detectan durante las fases de diseño e implementación, en lugar de en tiempos de ejecución.

2.1.4.7 Concurrencia. Es conveniente que el lenguaje soporte la creación de procesos paralelos independientes del sistema operativo. Cada programa tiene por lo menos un hilo de control, pero un sistema que involucra concurrencia puede tener muchos hilos: Algunos que son transitorios y otros que duran toda la ejecución del sistema. Los sistemas ejecutándose en múltiples CPUs tienen en cuenta los hilos de control verdaderamente concurrentes, mientras los sistemas ejecutándose en una sola CPU pueden solamente alcanzar la ilusión de hilos de control concurrentes, generalmente por medio de algún algoritmo que divide el tiempo.

2.1.4.8 Persistencia. Un objeto en el software toma una cantidad de espacio y existe durante una cantidad particular de tiempo. Atkinson sugiere que hay un continuo de la existencia de objetos que esta entre objetos transitorios que aparecen dentro de la evaluación de una expresión, hasta objetos en una base de datos que sobreviven la ejecución de un programa único. El espectro de persistencia de objetos abarca lo siguiente:

- Resultados transitorios en la evaluación de expresiones
- Variables locales en las activaciones de procedimiento
- Variables propias, variables globales, ítems apilados cuya extensión es diferente desde su alcance
- Datos que existen dentro ejecuciones de programa
- Datos que existen entre varias versiones de programa
- Datos que sobreviven al programa

Los lenguajes de programación tradicionales generalmente tienen en cuenta los tres primeros tipos de persistencia de objetos, mientras los últimos tres están típicamente en el dominio de las bases de datos.

Una definición formal de persistencia dada por Booch es:

"La persistencia es la propiedad de un objeto a través de la cual su existencia trasciende en el tiempo (es decir el objeto continua existiendo después que su creador deja de existir) y/o en el espacio (es decir, la localización del objeto se mueve desde el espacio de direcciones en el cual fue creado)"27.

2.1.4.9 Genericidad. Las clases parametrizadas (mediante plantillas –templates- o unidades genéricas) sirven para soportar un alto grado de reutilización. Estos elementos genéricos se diseñan con parámetros formales que se instanciarán con parámetros reales, para crear instancias de módulos que se compilan y enlazan y ejecutan posteriormente.

2.1.4.10 Manejo de excepciones. Se deben poder detectar, informar y manejar condiciones excepcionales utilizando construcciones del lenguaje. Esta propiedad añadida al soporte de tolerancia a fallos del software permitirá una estrategia de diseño eficiente.

2.1.5 Programación Orientada a Objetos (POO)

Grady Booch, define la Programación Orientada a Objetos (POO) como: "Un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representan una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas mediante relaciones de herencia"[28].

Es importante destacar tres aspectos fundamentales de esta definición:

[27] BOOCH, Grady. Op. Cit.
[28] BOOCH, Grady. Op. Cit.

1. Utiliza *Objetos*, no algoritmos, como bloques fundamentales de construcción lógicos (jerarquía de objetos).
2. Cada objeto es una instancia de una *clase*.
3. Las *clases* se relacionan unas con otras por medio de la *herencia*.

El concepto de objeto, junto con los tipos abstractos de datos o tipos definidos por el usuario, son una colección de elementos datos, junto con las funciones asociadas utilizadas para operar sobre esos datos.

Un programa puede parecer orientado a objetos, pero si cualquiera de estos elementos falta, no es un programa orientado a objetos. Específicamente la programación sin herencia es distinta de la Programación Orientada a Objetos; se denomina *Programación con tipos abstractos de datos* o *programación basada en objetos*.

Dada la definición anterior, un lenguaje esta orientado a objetos si y solo si satisface los siguientes requisitos:

1. Soporta objetos que son abstracciones de datos con una interfaz de operaciones con nombre y un estado local oculto.
2. Los objetos tienen un tipo asociado (clase).
3. Los Tipos (Clases) pueden heredar atributos de los supertipos (superclases).

A estos requisitos algunos autores agregan un cuarto:

4. Polimorfismo

Así podemos decir que los conceptos fundamentales de la POO son: *Objetos, Clases, herencia, mensajes y polimorfismo*.

2.1.6 Diseño orientado a Objetos (DOO). Grady Booch define el diseño orientado a objetos

como: "El diseño orientado a objetos es un método que abarca el proceso de la descomposición orientada a objetos y una notación para representar, tanto modelos lógicos como físicos, así como también modelos dinámicos del sistema bajo el diseño"[29].

Hay dos partes importantes en esta definición: el diseño orientado a objetos, primero lleva a una descomposición orientada a objetos y segundo que el diseño orientado a objetos utiliza notaciones diferentes para expresar modelos diferentes de la lógica (clases y estructuras de objetos) y el diseño físico (módulo y arquitectura de procesos) de un sistema.

El soporte de la descomposición orientada a objetos es lo que hace el diseño orientado a objetos bastante diferente del diseño estructurado. El DOO utiliza abstracciones de clase y objetos para los sistemas lógicamente estructurados, mientras que el diseño estructurado utiliza abstracciones algorítmicas.

2.1.7 Análisis orientado a objetos (AOO). Grady Booch define el Análisis Orientado a Objetos como: "El análisis orientado a objetos en un método de análisis que examina los requisitos de las perspectivas de las clases y objetos encontradas en el vocabulario del dominio de problemas"[30]

El análisis orientado a objetos enfatiza la construcción de modelos del mundo real, utilizando una visión orientada a objetos del mundo.

La relación entre AOO, DOO y POO es la siguiente: Los productos del AOO pueden servir como los modelos desde los cuales podemos comenzar un DOO. Los productos del DOO pueden entonces ser utilizados como dibujos para implementar completamente un sistema que use los métodos de la POO.

2.1.8 ¿Por que usar Programación Orientada a Objetos (POO)? Algunas de las causas

[29] BOOCH, Grady. Op. Cit.
[30] BOOCH, Grady. Op. Cit.

que están influyendo en el notable desarrollo de las técnicas de la programación orientada a objetos son:

- La Orientación a Objetos (OO) es especialmente adecuada para realizar determinadas aplicaciones, sobre todo realización de prototipos y simulación de programas.
- Los mecanismos de encapsulación de POO soportan un alto grado de reutilización de código, que se incrementa por sus mecanismos de herencia.
- En entorno de las bases de datos, la OO se adjunta bien a los modelos semánticos de datos para solucionar las limitaciones de los modelos tradicionales, incluido el modelo relacional.
- Aumento espectacular de lenguajes de Programación Orientados a Objetos (LPOO).
- Interfaces de usuario gráficas (con iconos) y visuales. Las interfaces de usuario de una aplicación manipulan la entrada y salida del usuario.

Estas razones hacen fundamentalmente que dentro de las tendencias actuales de la ingeniería de software, el marco de la POO se revele como el más adecuado para la elaboración del diseño y desarrollo de aplicaciones. Este marco se caracteriza por la utilización del diseño modular OO y la reutilización del software. Los Tipos abstractos de datos (TAD) han aumentado la capacidad de definir nuevos tipos (clases) de objetos, cuyo significado se definirá abstractamente, sin necesidad de especificar los detalles de implementación, tales como la estructura de datos a utilizar para la representación de los objetos definidos.

Los objetos pasan a ser los elementos fundamentales en este nuevo marco, en detrimento de los subprogramas que lo han sido en los marcos tradicionales.

2.1.9 Clasificación de los lenguajes Orientados a Objetos. Los lenguajes de programación orientados a objetos más conocidos son: Eiffel, Lisp, Prolog, Simula, Smalltalk, C++, Object Pascal, Java entre otros. Se entiende por lenguajes orientados a objetos aquellos que soportan las características de programación orientada a objetos. Otros lenguajes de programación tales como ADA, C, Cliper, pueden implementar algunas características orientadas a objetos, utilizando ciertas técnicas de programación, pero no se consideran orientados a objetos.

Algunos lenguajes como Ada (Ada-82) que no cumple las propiedades importantes de los Lenguajes de Programación Orientados a Objetos (LPOO), por ejemplo herencia y ligadura dinámica, soporta un enfoque orientado a objetos y se le conoce como basado en objetos. Ada-95 ya estandarizado por la ISO y la ANSI, soporta herencia y ligadura dinámica, en consecuencia, aunque todavía con restricciones se le considera orientado a objetos.

Una taxonomía de los lenguajes orientados a objetos fue hecha por Wegner. La clasificación incluye los siguientes grupos:

1. *Basado en objetos*: Un lenguaje de programación es basado en objetos si su sintaxis y semántica soportan la creación de objetos que tienen las propiedades del modelo de objetos.
2. *Basado en clases*: Si un lenguaje de programación es basado en objetos y soporta además la creación de clases, se considera basado en clases.
3. *Orientación a Objetos*: Un lenguaje de programación orientado a objetos es un lenguaje basado en clases que soporta también herencia.

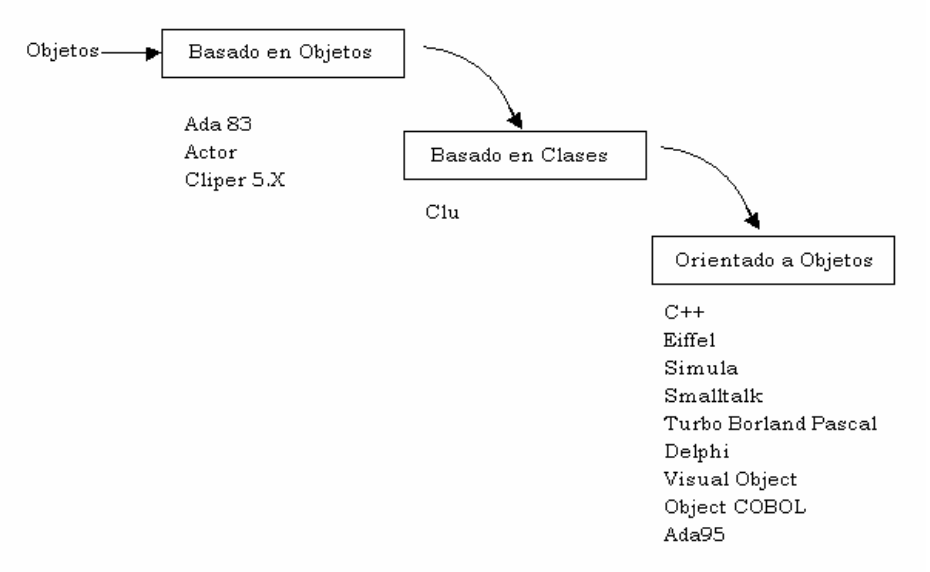


Figura 1. Clasificación de los Lenguajes Orientados a Objetos

2.1.10 Conceptos fundamentales de la POO

2.1.10.1 Objeto. La idea fundamental en los lenguajes orientados a objetos es combinar en una sola unidad datos y funciones que operan sobre esos datos. Tal unidad se denomina Objeto. Así que podemos definir un objeto como una entidad tangible que exhibe un comportamiento bien definido. Desde la perspectiva del conocimiento humano un objeto es cualquiera de lo siguiente:

- Una cosa tangible y/o visible.
- Algo que puede ser intelectualmente aprendido.
- Algo hacia lo cual se dirige la acción.

Los objetos del mundo real no son el único tipo de objetos que nos interesan en el momento de diseñar software. Otra clase de objetos que es de importancia son los inventos del proceso de diseño cuyas colaboraciones con otros objetos sirven como los mecanismos que proporcionan algún tipo de comportamiento de nivel superior.

Algunos objetos pueden ser tangibles y además tener fronteras físicas ocultas. Los objetos tales

como ríos, niebla, y agrupamiento de personas encajan perfectamente en esta definición, pero el desarrollador orientado a objetos debe tener cuidado porque no todo en el mundo es un objeto, existen algunas cosas que ciertamente no son objetos. Por ejemplo, los atributos tales como el tiempo, la belleza, el color no son objetos. Estas cosas son potencialmente propiedades de otros objetos. Por ejemplo se podría decir que un hombre (objeto) ama a su esposa (otro objeto) o que un gato (objeto) tiene el pelo gris. Por lo tanto necesitamos una definición un poco más formal de objeto. Grady Booch define objeto como:

> "Un objeto tiene estado, tiene comportamiento e identidad; la estructura del comportamiento de objetos similares se define en su clase común; los términos instancia y objeto son intercambiables"[31].

La estructura interna de un objeto consta de dos componentes básicos:

- *Atributos*: Los atributos describen el comportamiento de un objeto. Un atributo consta de dos partes: un nombre de atributo y un valor de atributo.

  Los objetos simples pueden constar de tipos primitivos, tales como entero, carácter, boolean, reales, o tipos simples definidos por el usuario. Los objetos complejos pueden constar de pilas, conjuntos, listas, array, etc. O incluso estructuras recursivas de alguno o todos los elementos.

- *Métodos (Operaciones o Servicios):* Los métodos describen el comportamiento asociado a un objeto. Representan las acciones que pueden realizarse por un objeto o sobre un objeto. La ejecución de un método puede conducir a cambiar el estado del objeto o dato local del objeto. Cada método tiene un nombre y un cuerpo que realiza la acción o comportamiento asociado con el nombre del método. En LPOO, el cuerpo de un método consta de un bloque de código procedimental que ejecuta la acción requerida. Todos los métodos que alteran o acceden a los datos de un objeto, se definen dentro del objeto. Un método puede modificar directamente o

[31] BOOCH, Grady. Op. Cit.

acceder a los datos de otros objetos.

Un método dentro de un objeto se activa por un mensaje que se envía por otro objeto al objeto que contiene el método. De modo alternativo se puede llamar por otro método en el mismo objeto por un mensaje local enviado de un método a otro dentro del objeto.

2.1.10.2 Clases. La clase es la construcción del lenguaje utilizada frecuentemente para definir los tipos abstractos de datos en lenguajes de programación orientados a objetos.

Generalmente una clase se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por un *estado* especifico y es capaz de realizar una serie de *operaciones* o también se podría definir como la representación de un conjunto de *objetos* que comparten una *estructura* y un *comportamiento* común.

En programación una clase es una estructura que contiene datos y procedimientos (o funciones) que son capaces de operar sobre esos datos. Dentro de un programa las clases tienen dos propósitos principales: Definir abstracciones y favorecer la Modularidad.

¿Cuál es la diferencia entre un objeto y una clase? Una clase verdaderamente describe una familia de elementos similares. En realidad una clase es una plantilla para un tipo particular de objetos. Si se tienen muchos objetos del mismo tipo, solo se tienen que definir una sola vez, en lugar de en cada objeto.

En resumen un *objeto* es una instancia de una *clase*.

2.1.10.3 Relaciones

- **Relaciones entre objetos.** Un objeto por si mismo no es muy interesante. Los objetos contribuyen al comportamiento del sistema en la medida en que colaboren unos con otros. La relación entre objetos cualesquiera abarca las suposiciones que cada objeto hace acerca del otro incluyendo qué operaciones pueden ser desempeñadas y qué comportamiento resulta. Se han encontrado dos tipos de relaciones de importancia en el diseño orientado a objetos:

  1. El uso de relaciones
  2. El hecho de tener relaciones (Relaciones de contención)

1. El uso de relaciones. En el caso de las relaciones de uso, un objeto esta en capacidad de pasar mensajes a otro. El pasar mensajes entre objetos generalmente es unidireccional, sin embargo también es posible la comunicación bidireccional. Dada una colección de objetos involucrada en relaciones de uso, cada objeto puede desempeñar uno de tres roles:

- Actor: Un objeto puede operar sobre otros objetos pero nunca otros objetos actúan sobre él; en algunos contextos el termino objeto activo y actor son sinónimos.
- Servidor: Un objeto que nunca opera sobre otros objetos; solamente otros objetos operan sobre él.
- Agente: Un objeto puede tanto operar sobre otros objetos y ser operado por otros objetos; un agente generalmente es creado para hacer algún trabajo en nombre de un actor o de otro agente.

2. Relaciones de contención. Podemos decir que una escalera eléctrica particular se hace de otros objetos tales como el motor y un censor de movimiento. En otras palabras el motor, el censor de movimiento son encapsulados como parte del estado de la escalera eléctrica.

Hay tres intercambios entre las relaciones de contenencia y de uso. El contener a diferencia del uso de un objeto es mejor porque el contener reduce el número de objetos que tienen

que ser visibles a nivel del objeto encerrador. Por otro lado, el usar es a veces mejor que el contener porque el contener lleva a una conexión más apretada y no deseable entre objetos. En ingeniería las decisiones inteligentes requieren de la ponderación cuidadosa de estos dos factores.

- **Relaciones entre Clases.** Podemos encontrar varios tipos de relaciones entre clases. Dentro de los lenguajes orientados a objetos, podemos encontrar que soportan alguna combinación de las siguientes relaciones entre clases:

  1. Relaciones de herencia.
  2. Relaciones de uso
  3. Relaciones de instancia.
  4. Relaciones de metaclase.

  1. Herencia. La herencia es la propiedad que permite a los objetos ser construidos a partir de otros objetos. Dicho de otro modo, la capacidad de un objeto para utilizar las estructuras de datos y los métodos previstos en antepasados o ascendientes. El objetivo final es la *reutilización* de código anteriormente desarrollado.

  La herencia se apoya en el significado de ese concepto en la vida diaria. Así las clases básicas o fundamentales se dividen en subclases. Los animales se dividen en mamíferos, anfibios, insectos, pájaros, peces, etc. La clase vehículos se divide en subclase automóvil, motocicleta, camión, autobús, etc. El principio en el que se basa la división de clases es la jerarquía compartiendo características comunes. Así todos los vehículos citados tienen motor y ruedas, que son características comunes; si bien los camiones tienen una caja para transportar mercancías mientras que las motocicletas tienen un manillar en lugar de un volante.

  La herencia supone una clase base y una jerarquía de clases que contienen las clases derivadas de la clase base. Las clases derivadas pueden heredar el código y los datos de su

clase base, añadiendo su propio código especial y datos a ellas, incluso cambiar aquellos elementos de la clase base que necesita sean diferentes.

La herencia es un mecanismo potente para tratar con la evolución natural de un sistema y con modificación incremental. Existen dos tipos diferentes de herencia: *Simple y Múltiple.*

En la *herencia simple*, un objeto (clase) solo puede tener un ascendiente, o dicho de otro modo solo puede heredar datos de una única clase, así como añadir o quitar comportamientos de la clase base.

La *herencia múltiple* es la propiedad de una clase de poder tener más de un ascendiente inmediato, o lo que es igual, adquirir datos y métodos de más de una clase.

2. **Relaciones de uso entre clases.** Hay dos tipos de relaciones de uso entre clases. La interfaz de una clase puede usar otra clase o la implementación de una clase puede usar otra clase. En el primer caso la clase usada tiene que ser visible a cualquier cliente, mientras que en el segundo caso la clase utilizada se oculta como parte del secreto de la clase de uso.

3. **Relaciones de instancia.** En algunas ocasiones podría ser de utilidad definir una clase que este en capacidad de contener objetos, una clase contenedora o lo que podríamos llamar un conjunto.

Idealmente nos gustaría definir una clase de conjuntos para que podamos enunciar con precisión la clase de objeto permitido en el conjunto y luego permitir que las reglas de tipado de nuestro lenguaje nos eviten hacerlo de otra manera.

Un conjunto es una clase de contenedor, el cual es una clase cuyas instancias son colecciones de otros objetos. Las clases contenedor pueden denotar colecciones

homogéneas, significando que todos los objetos de la colección son de la misma clase, o ellos pueden representar colecciones heterogéneas, en las cuales los objetos pueden ser de clases diferentes aunque tengan todos que compartir una superclase común. Las clases contenedoras más comunes incluyen pilas, listas, hileras de colas, mapas, anillos, conjuntos, bolsas, árboles y graficas.

Meyer ha indicado que la herencia es un mecanismo aun más poderoso que la Genericidad y que mucho del beneficio de la Genericidad puede ser logrado mediante la herencia, pero no viceversa. En la práctica encontramos útil utilizar un lenguaje que utilice tanto la herencia como la Genericidad.

4. **Relaciones de Metaclase.** Las relaciones de herencia, de uso y de instancia, juntas cubren todos los tipos importantes de relaciones de clase que un desarrollador podría necesitar. Sin embargo, aun es posible otro tipo de relación entre clases. Si partimos del hecho de que cada objeto es una instancia de una clase, ¿qué sucede si tomamos una clase como un objeto que puede ser manipulado? Para hacerlo así tenemos que preguntarnos cuál es la clase de una clase y la respuesta es simplemente una metaclase. Para decirlo de otra manera una metaclase es una clase cuyas instancias son clases por si mismas.

2.1.10.3.2 Anulación/Sustitución. La herencia permite que los atributos y los métodos definidos en una superclase sean heredados por las subclases. Sin embargo, si la propiedad se define nuevamente en la subclase, aunque se haya definido anteriormente a nivel de superclase, entonces la definición realizada en la subclase es la utilizada en esa subclase. Entonces se dice que anulan las correspondientes propiedades de la superclase. Esta propiedad se denomina *anulación* o *sustitución* (*overriding*).

2.1.10.4 Sobrecarga. La sobrecarga es una propiedad que describe una característica adecuada que utiliza el mismo nombre de operación para representar operaciones similares que se comportan de modo diferente cuando se aplican a tipos diferentes. Por consiguiente los nombres de las operaciones se pueden sobrecargar, esto es, las operaciones se definen con nombres idénticos,

aunque su código programado puede diferir.

Los lenguajes de programación tradicionales soportan la sobrecarga para algunas de las operaciones sobre algunos tipos de datos como enteros, reales y caracteres. Los sistemas orientados a objetos ahondan un poco más en la sobrecarga y la hacen disponible para operaciones sobre cualquier tipo de objeto.

2.1.10.5 Funciones o métodos virtuales. Las funciones virtuales permiten especificar un método como virtual en la definición de una clase particular. La implementación real del método se realiza en las subclases. En este caso, por consiguiente, la selección del método utilizado se determina utilizando ligadura dinámica o tardía en tiempo de compilación. Esto permite definir un método de un número de formas diferentes para cada una de las diferentes clases.

2.1.10.6 Polimorfismo. El polimorfismo en su expresión más simple, es el uso de un nombre o un símbolo para representar o significar más de una acción. La utilización de operadores o funciones de formas diversas, dependiendo de cómo se estén operando, se denomina polimorfismo (múltiples formas). Cuando un operador existente en el lenguaje tal como +, -, * o = se le asigna la posibilidad de operar sobre un nuevo tipo de dato, se dice que esta sobrecargado. La sobrecarga es una clase de polimorfismo, que también es una característica importante de la POO. Un uso típico de los operadores aritméticos es la sobrecarga de los mismos para actuar sobre tipos de datos definidos por el usuario (objetos), además de sobre los tipos de datos predefinidos.

La gran ventaja ofrecida por el polimorfismo y la sobrecarga, en particular es permitir que los nuevos tipos de datos sean manipulados de forma similar que los datos predefinidos, logrando así ampliar el lenguaje de programación.

En un sentido más general, el polimorfismo supone que un mismo mensaje puede producir acciones totalmente diferentes cuando se reciben por objetos diferentes. Con polimorfismo un usuario puede enviar un mensaje genérico y dejar los detalles de la implementación exacta para el objeto que

recibe el mensaje. El polimorfismo se fortalece con el mecanismo de herencia.

2.1.10.7 Reutilización. La reutilización es la propiedad que permite que el software desarrollado pueda ser utilizado cuantas veces sea necesario en más de un programa. La programación orientada a objetos facilita el manejo de sistemas complejos en software. El manejo y las facilidades de la representación del conocimiento de la programación orientada a objetos proveen un gran incremento en la productividad del programador.

El modo estándar de la programación orientada a objetos es buscar soluciones anteriores que estén disponibles para el problema de aplicación; en lugar de programar desde el inicio (o reinventar la rueda), esto es posible empezando desde un inicio predefinido. Esto requiere que se coloque más énfasis en el desarrollo de clases reutilizables que estén bien documentadas y probadas. Siempre que una nueva clase sea creada es importante que sea tan general como sea posible, esto es, que provea el más alto nivel de abstracción y el más alto grado de ocultamiento de la información. Desde luego es también importante que la clase misma este bien descrita y documentada, también como sus características publicas.

Sin embargo hay que tener cuidado porque en algunos casos la reutilización es vendida como la mayor ventaja de la programación orientada a objetos. Mientras que esto es cierto en algunos casos, generalmente en una aplicación de gran envergadura, esto no es cierto en general. La mayor reutilización ocurre dentro de una muy cerrada y entretejida jerarquía de clases. Subclases heredan de sus superclases y en una jerarquía de clases sustancial es evidente que la subclase más baja heredara una gran porción de código de las otras clases en la jerarquía.

Mientras que es posible y fácil reutilizar o compartir código dentro de una aplicación, hay algunas barreras para la reutilización entre aplicaciones o en futuras tareas de programación. Aparte de los recurrentes tipos abstractos de datos, tales como E/S, listas, conjuntos, colecciones, strings, etc., tipos abstractos de datos de aplicaciones específicas son difíciles de reutilizar.

Los lenguajes de programación que confían solamente en la ligadura tardía o dinámica, esto es, toda función es virtual permiten mayor flexibilidad a la hora de reutilizar clases en diferentes contextos. Sin embargo esto no garantiza que la reutilización se pueda hacer y si es así que se haga de una manera fácil.

2.2 UML

2.2.1 Unified Modeling Language (UML) . El UML es un lenguaje estándar para escribir planos de software (software blueprints). El UML puede ser usado para visualizar, especificar, construir y documentar los artefactos de un sistema de software.

El UML es un lenguaje muy expresivo, que provee todas las vistas necesarias para poder desarrollar e implantar un sistema. A pesar de ser un lenguaje muy expresivo, no es difícil de entender y utilizar. Para entender y poder aplicar el UML efectivamente es necesario comenzar por formarse un modelo conceptual del lenguaje, lo cual requiere entender tres grandes elementos:

1. Bloques básicos de construcción.
2. Reglas: Dictan como los bloques básicos de construcción deben ser puestos juntos.
3. Mecanismos.

El UML es un lenguaje y de esta forma es solo una parte del método de desarrollo de Software. El UML es independiente del proceso, sin embargo, óptimamente debe ser usado en un proceso que sea manejado por casos de uso, centrado en la arquitectura, iterativo e incremental.

2.2.2 El UML como lenguaje. Un lenguaje provee un vocabulario y las reglas para combinar palabras dentro de este lenguaje con el propósito de comunicar. Un lenguaje de modelamiento es un lenguaje cuyo vocabulario y reglas están enfocados en la representación física y conceptual de

un sistema.

Para modelar un sistema software no es suficiente un solo modelo, la mayoría de las veces serán necesarios múltiples modelos conectados entre si para lograr entender el sistema. El UML es un lenguaje cuyo vocabulario y reglas son capaces de decir como crear y leer modelos bien formados, sin embargo, el UML no es capas de determinar que modelos deben ser creados y cuando. Para poder determinar los modelos que deben ser creados y cuando se debe utilizar un proceso de desarrollo.

2.2.3 El UML es un lenguaje para Visualizar. Para muchos programadores el espacio entre pensar una implementación y codificarla es casi cero. El programador piensa la implementación, y la codifica. De hecho, algunas cosas son mejor expresadas en código. En estos casos el programador esta modelando, pero este es un proceso enteramente mental. Sin embargo, hay varios problemas con esto.

El primero de ellos es comunicar estos modelos a otros miembros de la organización. Esta comunicación esta sujeta a errores ya que no todos los integrantes de la organización manejan el mismo lenguaje. Típicamente las organizaciones desarrollan su propio lenguaje, pero esto no es suficiente si se desean comunicar con alguien que no pertenezca a la organización. En el caso de un nuevo miembro tomara algo de tiempo para que este aprenda el lenguaje que se esta utilizando.

El segundo problema tiene que ver con que hay algunos aspectos del sistema software que no son posibles de entender a menos que se construya un modelo que trascienda el lenguaje de programación textual. Un ejemplo de esto puede ser las jerarquías de clases.

El Tercer problema radica en que algunos desarrolladores nunca escriben el modelo que esta en su cabeza y debido a esto esta información se pierde para siempre o en el mejor de los casos solo se reconstruirá parcialmente de la implementación una vez el desarrollador se halla ido.

El UML presenta una solución para cada uno de estos problemas:

- Escribir modelos en UML soluciona el tercer problema: Un modelo explicito facilita la comunicación.
- Algunas cosas son modeladas mejor en texto; otras se modelan mejor gráficamente. De hecho en todos los sistemas interesantes hay estructuras que trascienden lo que puede ser representado en un lenguaje de programación. El UML es un lenguaje grafico, esto solucione el segundo problema.
- El UML es más que una gran cantidad de símbolos gráficos. Más que eso, detrás de cada símbolo en la notación de UML esta una semántica bien definida. De esta forma un desarrollador puede escribir un modelo en UML y otro desarrollador o incluso otra herramienta puede interpretar este modelo sin ambigüedades. Esto soluciona el primer problema.

2.2.4 El UML es un lenguaje para especificar. En este contexto, especificar significa construir modelos que son precisos, sin ambigüedades, y completos. En particular el UML trata todas las decisiones importantes en cuanto a análisis, diseño e implementación que puedan ser hechas en el desarrollo y la implantación de un sistema software.

2.2.5 El UML es un lenguaje para construir. UML nos es un lenguaje de programación visual, pero estos modelos pueden ser directamente conectados a una gran variedad de lenguajes de programación. Esto significa que es posible llevar un modelo en UML a un lenguaje de programación como Java, C++, Object-Pascal o incluso a una base de datos relacional u orientada a objetos.

UML permite ingeniería hacia delante: Generación de código a partir de un modelo dentro de un lenguaje de programación. Ingeniería inversa también es posible: generar un modelo en UML a partir de una implementación. Sin embargo la ingeriría inversa no es mágica, es decir puede que no sea posible reconstruir el modelo por completo. Uniendo la ingeniería hacia delante y la ingeniería inversa obtenemos lo que podríamos llamar el viaje completo de la ingeniería, la habilidad de trabajar ya sea en una vista textual o una grafica mientras se mantienen las dos vistas consistentes.

En adición a esto el UML es suficientemente expresivo y sin ambigüedades como para permitir la ejecución directa de modelos, la simulación de sistemas la instrumentación de sistemas en ejecución.

2.2.6 El UML es un lenguaje para documentar. Una organización produce todo tipo de artefactos para producir nuevo código ejecutable. Estos artefactos incluyen (pero no están limitados a):

- Requerimientos
- Arquitectura
- Diseño
- Código fuente
- Planes de proyecto
- Pruebas
- Prototipos
- Versiones

Dependiendo de la cultura de desarrollo algunos de estos artefactos son tratados con más o menos formalidad.

El UML reúne la documentación de la arquitectura de un sistema y todos sus detalles. El UML también provee un lenguaje para expresar requerimientos y para hacer pruebas. Finalmente, el UML provee un lenguaje para modelar las actividades del planeamiento de un proyecto.

2.2.7 Modelo conceptual del UML. Para entender el UML es necesario crear un modelo conceptual del lenguaje, y esto requiere aprender tres elementos principales: Los bloques básicos de construcción del UML, las reglas que dictan como esos bloques pueden colocarse juntos y algunos

mecanismos comunes que se aplican a través del UML. Una vez entendido esto será posible leer y crear algunos modelos en UML.

Bloques de construcción básicos del UML. El vocabulario del UML encierra tres clases de bloques de construcción:

- Entidades
- Relaciones
- Diagramas

Las entidades son abstracciones que son ciudadanas de primera clase en un modelo; las relaciones unen estas entidades; los diagramas agrupan interesantes colecciones de entidades.

- **Entidades en UML.** Hay cuatro clases de entidades dentro del UML:
  1. Entidades estructurales
  2. Entidades comportacionales
  3. Entidades grupales
  4. Entidades de anotación

  Estas entidades son los bloques de construcción básicos basados en objetos de UML. Estos bloques serán usados para escribir modelos bien formados.

  1. **Entidades Estructurales.** Las entidades estructurales son los sustantivos de los modelos en UML. Estos son los elementos más estáticos de UML, representado elementos que pueden ser conceptuales o físicos. Encontramos siete clases de elementos estructurales:
     a. Clases
     b. Interfaces

c. Colaboraciones

d. Casos de Uso

e. Clases Activas

f. Componentes

g. Nodos

a. Clases. Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Una clase puede implementar una o más interfaces. Gráficamente una clase es representada por un rectángulo, usualmente incluyendo su nombre, atributos, y operaciones como lo muestra la figura 2.

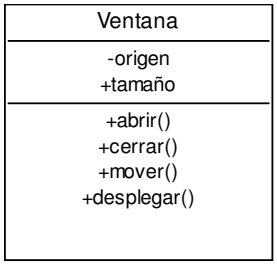


Figura 2. Clases.

b. Interfaces. Una interfaz es una colección de operaciones que especifican un servicio de una clase o un componente. Una interfaz describe el comportamiento externo de un elemento. Una interfaz puede representar el comportamiento completo de una clase o componente o solo puede mostrar una parte de este. Una interfaz define un grupo de especificación de operaciones pero nunca las implementaciones de estas. Una interfaz es representada gráficamente como un circulo con su nombre, esta raramente esta sola. Por lo general esta acompañada de una clase o componente que se encarga de realizar la interfaz.

Figura 3. Interfaz.

c. Colaboraciones. Una colaboración define una interacción de una sociedad de roles y otros elementos que trabajan juntos para proveer un tipo de comportamiento colectivo que es más grande que la suma de los partes. Gráficamente una colaboración es dibujada como una elipse con la línea punteada, usualmente incluyendo solo su nombre.

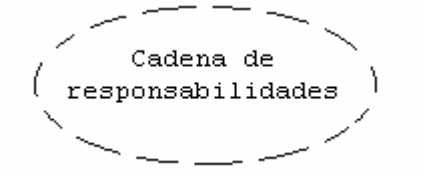


Figura 4. Colaboraciones.

d. Casos de Uso. Un caso de uso es una descripción de un grupo de secuencias de acciones que un sistema desarrolla y que produce un resultado observable de valor para un actor. Un caso de uso es utilizado para estructurar el comportamiento de las entidades en un modelo. Un caso de uso es realizado por una colaboración. Gráficamente un caso de uso es representado por una elipse con línea continua, usualmente incluyendo solamente su nombre, como lo muestra la figura 5.

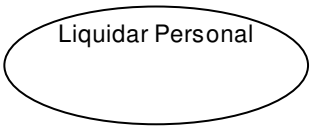


Figura 5. Casos de uso.

e. Clases Activas. Una clase activa en una clase cuyos objetos poseen uno o más procesos o hilos (threads) y de esta forma pueden iniciar un control de actividad. Una clase activa es como una clase excepto que sus objetos representan elementos cuyo

comportamiento es concurrente con otros elementos. Gráficamente una clase activa es representada como una clase, pero con líneas en negrita, usualmente incluyendo su nombre, atributos, y operaciones, como lo muestra la figura 6.

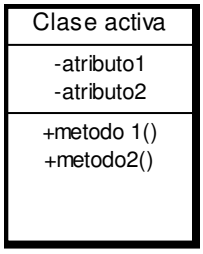


Figura 6. Clases activas.

f. Componentes. Un componente es una parte física y reemplazable de un sistema que conforman y proveen la realización de un grupo de interfaces. En un sistema se pueden encontrar diferentes clases de componentes desarrollados, tales como componentes COM o java beans, también como componentes que son artefactos del proceso de desarrollo, tales como código fuente. Un componente típicamente representa el empaquetamiento físico de elementos lógicos, tales como clases, interfaces y colaboraciones. Gráficamente un componente se representa como un rectángulo con marquillas, usualmente incluyendo solamente su nombre como lo muestra la figura 7.

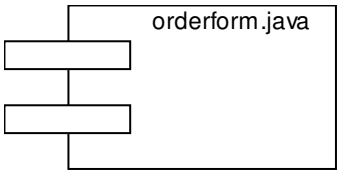


Figura 7. Componentes.

g. Nodos. Un nodo es un elemento físico que existe en tiempo de ejecución y que representa un recurso computacional, generalmente teniendo al menos algo de memoria y, a menudo, capacidad computacional. Gráficamente es representado por un cubo, usualmente incluyendo su nombre, como lo muestra la figura 8.

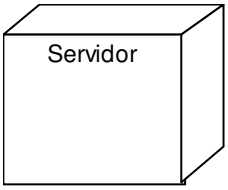


Figura 8. Nodos.

2. **Entidades comportacionales.** Las entidades comportacionales son las partes dinámicas de los modelos en UML. Estos son los verbos del modelo, representando comportamientos sobre tiempo y espacio. En UML encontramos dos clases de entidades comportacionales:

   a. Interacción

   b. Maquina de estado

   a. Interacciones. Una interacción es un comportamiento que comprende un grupo de mensajes intercambiados entre un grupo de objetos dentro de un contexto particular para llevar a cabo un propósito especifico. El comportamiento de una sociedad de objetos o de una operación individual puede ser especificado por una interacción. Gráficamente un mensaje es representado como una línea recta, casi siempre incluyendo el nombre de su operación, como lo muestra la figura 9.

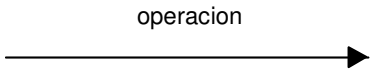


Figura 9. Mensajes.

   b. Maquinas de estado. Una maquina de estado es un comportamiento que especifica la secuencia de estados de un objeto o una interacción que se lleva a cabo durante su tiempo de vida en respuesta a eventos, junto con sus respuestas a esos eventos. El comportamiento de una clase individual o una colaboración de clases puede ser

especificado por una maquina de estado. Una maquina de estado envuelve un número de otros elementos incluyendo estados, transiciones y actividades. Gráficamente una maquina de estado es representado como un rectángulo con sus esquinas redondeadas, usualmente incluyendo su nombre y sus subestados, si hay, como lo muestra la figura 10.

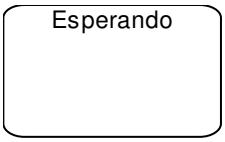


Figura 10. Maquina de estado.

3. **Entidades Grupales.** Las entidades grupales son la parte organizacional de los modelos en UML. Estas son las cajas dentro de las cuales se puede descomponer un modelo. Los *paquetes* son la parte primaria de las entidades grupales en UML.

   a. **Paquetes.** Un paquete es un mecanismo de propósito general para organizar elementos dentro de grupos. A diferencia de los componentes (los cuales existen en tiempo de ejecución), un paquete es puramente conceptual (significando que este solo existe en tiempo de desarrollo). Gráficamente un paquete es representado como un fólder con una marquilla, usualmente incluyendo solamente su nombre y algunas veces su contenido, como lo muestra la figura 11.

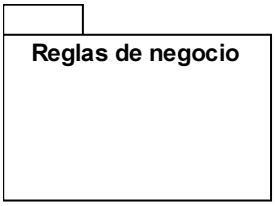


Figura 11. Paquetes.

4. **Entidades de anotación.** Estas entidades son la parte explicatoria de los modelos en UML. Estos son los comentarios que pueden ser colocados para describir, iluminar y remarcar cualquier elemento sobre el modelo. Hay una clase primaria de las entidades de anotación, llamada *nota*. Una nota es simplemente un elemento utilizado para colocar limites o comentarios atados a un elemento o un grupo de elementos. Gráficamente una nota es representada como un rectángulo con una esquina doblada hacia adentro, junto con un texto o un comentario grafico, como lo muestra la figura 12.

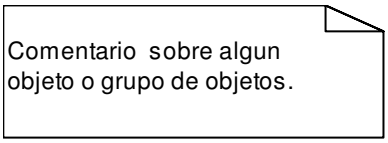


Figura 12. Notas.

- **Relaciones.** Hay cuatro clases de relaciones en UML:

1. Dependencia
2. Asociación
3. Generalización
4. Realización

Estas relaciones son los bloques de construcción básicos de relaciones en UML.

1. **Dependencia.** Una dependencia es una relación semántica entre dos entidades en la cual el cambio en una entidad (la entidad independiente) puede afectar la semántica de la otra entidad (la entidad dependiente). Gráficamente una dependencia es representada como una línea punteada, posiblemente dirigida y ocasionalmente con una etiqueta como muestra la figura 13.

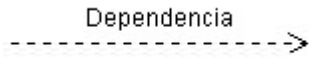


Figura 13. Dependencia.

2. **Asociación.** Una asociación es una relación que describe un grupo de uniones, entendiendo unión como una conexión entre objetos. La agregación es un caso especial de asociación, representando una relación estructural entre el todo y sus partes. Gráficamente una asociación es representada como una línea continua, posiblemente dirigida, ocasionalmente etiquetada, y a menudo incluyendo otros adornos, tales como multiplicidad y nombres de rol, como en la figura 14.

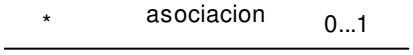


Figura 14. Asociación.

3. **Generalización.** Una generalización es una relación de generalización/especialización en la cual los objetos del elemento especializado (hijo) son sustituibles por objetos del elemento generalizado (padre). De esta forma el hijo comparte la estructura y el comportamiento del padre. Gráficamente una generalización es representada por una línea continua con una flecha hueca apuntando hacia el padre, como en la figura 15.

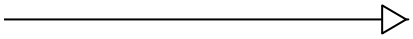


Figura 15. Generalización.

4. **Realización.** Una realización es una relación semántica entre clasificadores en la cual un clasificador especifica un contrato que otro garantiza llevar a cabo. Este tipo de relación se encontrara en dos lugares: Entre interfaces y las clases o componentes que las realizan y entre casos de uso y las colaboraciones que los realizan. Gráficamente una realización es representada por un cruce entre una relación de dependencia y una generalización como se muestra en la figura 16.

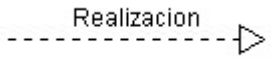


Figura 16. Realización.

- **Diagramas.** Un diagrama es la representación grafica de un grupo de elementos, la mayoría de las veces dibujado como un grafico de vértices (entidades) y arcos (relaciones). Los diagramas se realizan para visualizar un sistema desde diferentes perspectivas, así un diagrama es una proyección dentro de un sistema. En teoría un diagrama puede tener cualquier combinación de entidades y relaciones. En la práctica, sin embargo, un pequeño número de combinaciones comunes surgen, lo cual es consistente con las cinco visiones más útiles que hacen parte de la arquitectura de un sistema software. Por esta razón UML incluye nueve tipos de diagramas:

  1. Diagrama de clases
  2. Diagrama de objetos
  3. Diagrama de casos de uso
  4. Diagrama de secuencias
  5. Diagrama de colaboración
  6. Diagrama de estados
  7. Diagrama de actividades
  8. Diagrama de componentes
  9. Diagrama de despliegue

  1. **Diagrama de clases.** Un diagrama de clases muestra un grupo de clases, interfaces, y colaboraciones y sus relaciones. Estos diagramas son los más encontrados en el modelamiento de un sistema orientado a objetos. Los diagramas de clases reúnen el diseño estático de un sistema. Si estos diagramas incluyen clases activas, este diagrama reúne la vista estática de los procesos de un sistema.

  2. **Diagrama de objetos.** Un diagrama de objetos muestra un grupo de objetos y sus

relaciones. Un diagrama de objetos representa una fotografía de las instancias de las entidades encontradas en un diagrama de clases. Este diagrama trata con la vista de diseño estática o de proceso estática de un sistema como lo hace un diagrama de clases, pero desde la perspectiva de un caso real o un caso prototipo.

3. **Diagrama de casos de uso.** Un diagrama de casos de uso muestra un grupo de casos de uso y actores (una tipo especial de clase) y sus relaciones. Este diagrama muestra una vista estática de los casos de uso de un sistema. Estos diagramas son especialmente importantes en organizar y modelar el comportamiento de un sistema.

4. **Diagrama de secuencias.** Este diagrama es una clase de diagrama de interacción. Un diagrama de interacción muestra una interacción consistente en un grupo de objetos y sus relaciones incluyendo los mensajes que pueden ser enviados entre ellos. Los diagramas de interacción tratan con la parte dinámica del sistema. Un diagrama de secuencias es un diagrama de interacciones que hace énfasis en el tiempo de ordenación de los mensajes. Este diagrama es isomórfico con el diagrama de colaboración, lo que quiere decir que se puede tomar el diagrama de secuencias y convertirlo en un diagrama de colaboración.

5. **Diagrama de colaboración.** Este diagrama es un tipo de diagrama de interacción, haciendo énfasis en la organización estructural de los objetos que envían y reciben mensajes. Este diagrama es isomórfico con el diagrama de secuencias, lo que quiere decir que puede tomar el diagrama de colaboración y convertirlo en un diagrama de secuencias.

6. **Diagrama de estados.** Un diagrama de estados muestra una maquina de estados consistente en estados, transiciones, eventos, y actividades. Este diagrama trata con la vista dinámica de un sistema.

7. **Diagrama de actividades.** Un diagrama de actividades es un tipo especial de diagrama de estados, que muestra el flujo de actividad a actividad dentro de un sistema. Este tipo de

diagramas es importante modelando la función de un sistema y hace énfasis en el control de flujo entre los objetos.

8. **Diagrama de componentes.** Un diagrama de componentes muestra la organización y las dependencias dentro de un grupo de componentes. Los diagramas de componentes tratan con la vista de implementación estática de un sistema. Estos esta relacionados con los diagramas de clases porque generalmente un componente desarrolla a uno o más clases, interfaces, o colaboraciones.

9. **Diagrama de despliegue.** Un diagrama de despliegue muestra la configuración de los nodos procesando en tiempo de ejecución y los componentes que viven en ellos. Estos diagramas se relacionan con los diagramas de componentes porque por lo general un nodo encierra uno o más componentes.

Reglas del UML. Los bloques básicos de construcción del UML no pueden simplemente lanzarse juntos de una forma aleatoria. Como cualquier otro lenguaje el UML tiene un número de reglas que especifican como debe lucir un modelo bien formado en UML.

El UML tiene reglas semánticas para:

1. *Nombres:* se pueden nombrar entidades, relaciones y diagramas.
2. *Radio de acción:* El contexto que da un significado especial al nombre
3. *Visibilidad:* Como esos nombres pueden ser vistos y usados por otros
4. *Integridad:* Cómo las entidades propia y consistentemente se relacionan con otras
5. *Ejecución:* Lo que esto significa para correr o simular un modelo dinámico

Los modelos construidos durante el desarrollo de un sistema software tienden a complicarse y pueden ser vistos por varios observadores en diferentes caminos y a diferentes horas. Por esta razón es común para el equipo de desarrollo no solamente escribir modelos bien formados sino también

construir modelos que son:

1. *Elididos:* Ciertos elementos son escondidos para simplificar la vista
2. *Incompletos:* Ciertos elementos pueden perderse
3. *Inconsistentes:* La integridad del modelo no esta garantizada

Estos modelos menos que bien formados son inevitables mientras los detalles de un sistema se revelan y se mezclan durante el desarrollo del ciclo de vida del software.

Mecanismos comunes del UML. El UML esta conformado por cuatro modelos de características comunes que se aplican consistentemente a través del lenguaje. Estas características son de utilidad para construir modelos harmoniosos sencillamente. Los cuatro mecanismos comunes en UML son:

1. Especificaciones
2. Adornos
3. Divisiones comunes
4. Mecanismos de Extensibilidad

1. **Especificaciones.** Detrás de la notación grafica de UML hay una especificación que provee una declaración textual de la sintaxis y la semántica de sus bloques de construcción.

2. **Adornos.** La mayoría de los elementos de UML tienen una única y directa notación que provee una representación visual de los aspectos más importantes del elemento, sin embargo, en algunas ocasiones se hace necesario complementar esta información básica. Para esto UML permite que a esta notación básica se adiciones una variedad de adornos que permitan representar esta información adicional.

3. **Divisiones comunes.** En el modelamiento de sistemas orientados a objetos, el mundo es

dividido en al menos un par de formas. Primero que toso podemos encontrar una división entre una clase y un objeto que es la manifestación concreta de tal abstracción. En UML se pueden modelar tanto las clases como los objetos.

Casi cada bloque de construcción de UML tiene esta misma clase de dicotomía. Por ejemplo en UML se pueden utilizar casos de uso e instancias de casos de uso, componentes e instancias de componentes, etc. Gráficamente UML distingue un objeto usando el mismo símbolo de su clase pero subrayando el nombre del objeto.

4. **Mecanismos de extensibilidad.** El UML provee un estándar para escribir planos de software, pero es imposible para un lenguaje cerrado representar todos los posibles matices de todos los modelos a través de todos los dominios, a través de todo el tiempo. Por esta razón el UML es abierto, haciendo posible extender el lenguaje de forma controlada. Los mecanismos de extensibilidad de UML incluyen:

   a. *Estereotipos:* un estereotipo extiende en lenguaje permitiendo crear nuevos tipos de bloques de construcción que son derivados de los ya existentes pero que son específicos para un problema determinado.

   b. *Valores marcados*: Estos valores extienden el uso de UML permitiendo crear nueva información en la especificación de un elemento.

   c. *Limites:* Los limites extienden la semántica de UML permitiendo adicionar nuevas reglas o modificar las existentes.

2.3 DELPHI

Delphi es un entorno de programación visual orientado a objetos para desarrollo rápido de aplicaciones (RAD, Rapid Application Development). Delphi provee todas las herramientas necesarias para desarrollar, probar, depurar aplicaciones, incluyendo una gran librería de componentes reusables, una colección de herramientas de diseño, aplicación y plantillas para formas. Estas herramientas simplifican el prototipado y reducen el tiempo de desarrollo.

El lenguaje utilizado por Delphi es Object Pascal. Object Pascal es un lenguaje orientado a objetos y por lo tanto soporta la gran mayoría de las características de los Lenguajes Orientados a Objetos (LOO) discutidas en la sección de 2.1.4 Elementos del Modelo de Objetos; las características no soportadas por Delphi son herencia múltiple y un concepto introducido por C que es la clase amiga. El hecho de que no soporte directamente estos dos conceptos no significa que no puedan ser utilizados; Delphi presenta soluciones para lograr el efecto de la herencia múltiple así como el de las clases amigas.

Además de las características presentadas Delphi es considerado como una de las herramientas más productivas gracias a la calidad del entorno visual, la velocidad de compilador frente a su complejidad, la potencia del lenguaje de programación frente a su complejidad, la flexibilidad y la escalabilidad de la arquitectura de la base de datos, los métodos de diseño y de utilización recomendados por el entorno.

2.4 OpenGL

OpenGL esta estrictamente definido como "una interfaz software para un hardware de gráficos". En esencia OpenGL es una librería de modelado y gráficos en 3D que es extremadamente portable y muy rápida. OpenGL esta en la capacidad de crear gráficos casi con la calidad de un *ray tracer*[*]. La gran ventaja de usar OpenGL es que este es más rápido en orden de magnitud que un *ray tracer*.

[*] Trazador de rayos.

OpenGL usa algoritmos cuidadosamente desarrollados y optimizados por Silicon Graphics Inc. (SGI), un reconocido líder mundial en gráficos por computadora y animación.

OpenGL no es un lenguaje de programación como C o C++. Este es como una librería de C, la cual provee alguna funcionalidad empaquetada. Realmente no exista algo como un "programa en OpenGL", pero más que esto un programa que un desarrollador escribe que utiliza OpenGL como uno de sus APIs[**].

OpenGL esta diseñado para usarse con computadoras con hardware que es diseñado y optimizado para mostrar y manipular gráficos en 3D. Las implementaciones de OpenGL que solo usan software también son posibles, una de estas implementaciones es la de Microsoft.

2.4.1 Evolución del estándar. OpenGL es un estándar relativamente nuevo en la industria que en pocos años ha ganado una gran cantidad de seguidores. El precursor de OpenGL fue IRIS GL de Silicon Graphics. Este fue el API de programación en 3D de las estaciones de trabajo IRIS. Estas computadoras fueron mucho más que computadoras de propósito general; estas estaciones tenían hardware especializado y optimizado para la visualización de graficas sofisticadas.

El hardware provee una matriz de transformación ultrarrápida (un prerrequisito para gráficos en 3D), soporte de hardware para la profundidad del buffer y otras características. Sin embargo, cuando SGI trato de portar IRIS GL a otras plataformas ocurrieron algunos problemas.

OpenGL es el resultado del esfuerzo de SGI para mejorar la portabilidad de IRIS GL. El nuevo API ofrece el poder de GL pero es "open" (abierto), permitiendo una adaptabilidad más fácil a otras plataformas de hardware y sistemas operativos.

2.4.2 El ARB (Architecture Review Board) de OpenGL. Un estándar no es realmente abierto

[**] Application Programming Interface.

si uno de los vendedores lo controla. Un estándar realmente abierto adoptado por un gran número de vendedores hace más fácil para los programadores la creación de aplicaciones y el contenido esta disponible para una gran variedad de plataformas. El software es lo que vende las computadoras, y SGI quiere vender más computadoras, necesitando por esto más software que corra en sus computadoras. Otros vendedores se dieron cuenta de esto también y por esta razón nace el OpenGL Architecture Review Board. Los fundadores del ARB fueron SGI, Digital Equipment Corporation, IBM, Intel y Microsoft. El ARB se reúne cuatro veces por año.

2.4.3 Cómo trabaja OpenGL. OpenGL es un API de gráficos procedural más que descriptivo. En lugar de describir la escena y como esta debería aparecer, el programador realmente prescribe los pasos necesarios para alcanzar una cierta apariencia o efecto. Estos pasos envuelven llamadas al API, el cual incluye más de 200 comandos y funciones. Estos comandos son utilizados para dibujar primitivas de gráficos tales como puntos, líneas y polígonos en tres dimensiones. En adición a esto OpenGL soporta iluminación y sombreado, desarrollo de texturas, matizado, transparencia, animación y otros efectos y capacidades especiales.

OpenGL no incluye ninguna función para manejo de ventanas, interacciones con el usuario, o archivos de E/S. Cada ambiente anfitrión (como Microsoft Windows), tiene sus propias funciones para este propósito y es responsable de implementar algunos medios para manejarlas.

No hay un formato de archivo "OpenGL" para los modelos o para un ambiente virtual. Estos ambientes son creados por el programador para ajustarse a sus necesidades de alto nivel y entonces cuidadosamente programadas usando los comandos de bajo nivel de OpenGL.

2.4.4 Implementaciones genéricas. Una implementación genérica es una implementación de software. Las implementaciones de hardware son creadas para un dispositivo especifico de hardware, tales como una tarjeta grafica o un generador de imágenes. Una implementación genérica puede correr sobre cualquier sistema siempre y cuando el sistema tenga la habilidad de mostrar la imagen generada.

La figura 17 muestra el lugar que OpenGL y una implementación genérica ocupan cuando una aplicación se esta ejecutando. El programa llama muchas funciones, algunas de las cuales el programador creó, otras son proporcionadas por el sistema operativo o las librerías de tiempo de ejecución del lenguaje de programación. Las aplicaciones bajo Windows que desean crear una imagen que sea mostrada en la pantalla usualmente llaman a un API llamado GDI (Graphic Devise Interface). El GDI contiene los métodos que permiten escribir texto, dibujar graficas simples, líneas en 2D, etc.

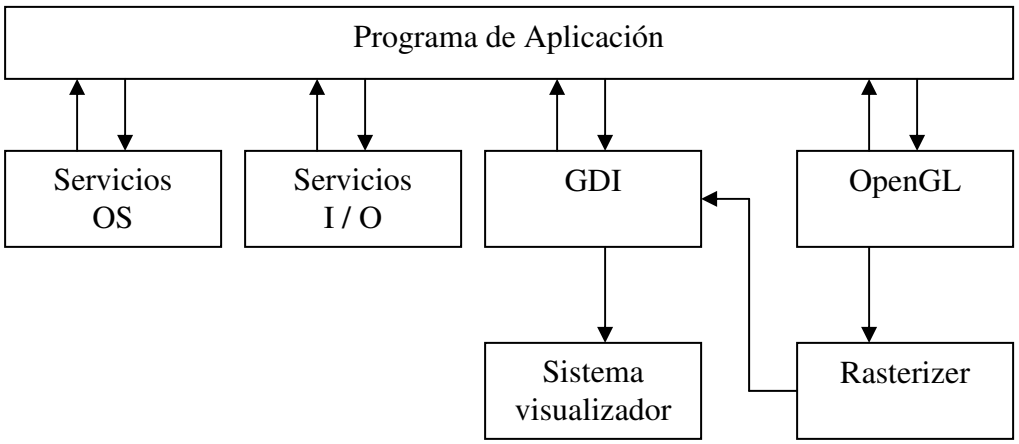


Figura 17. OpenGL dentro de una aplicación típica.

Usualmente los vendedores de tarjetas de video proveen un manejador que se conecta con el GDI para poder visualizar los gráficos en el monitor. Una implementación software de OpenGL toma la petición de gráficos de una aplicación y construye (Rasterizes) una imagen a color del grafico en 3D. Este es entonces enviado al GDI para que sea mostrado en el monitor. En otros sistemas operativos, el proceso es prácticamente el mismo con la diferencia que en lugar del GDI el sistema operativo utilizara su sistema de visualización nativo.

2.4.5 Implementaciones hardware. Una implementación hardware de OpenGL generalmente toma la forma de un controlador de una tarjeta de video. La figura 18 muestra la relación de OpenGL con la aplicación. Es importante notar como la llamada del API OpenGL pasa al controlador del hardware. Este controlador no pasa su salida al GDI de Windows para su visualización; el manejador se conecta directamente con la tarjeta de video.

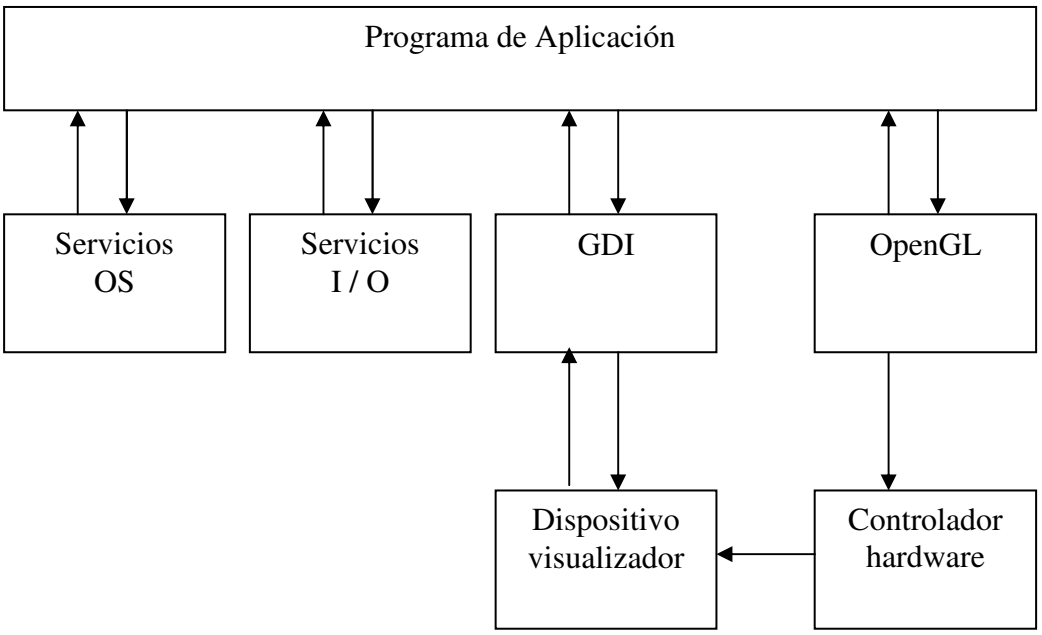


Figura 18. OpenGL acelerado por hardware dentro de una aplicación típica.

Una implementación hardware es a menudo referida como una implementación acelerada porque usualmente los gráficos asistidos por hardware son mucho más rápidos que las implementaciones que solo utilizan software. Lo que la figura no muestra es que algunas veces parte de la funcionalidad de OpenGL esta aun implementada en software como parte del controlador, y otras características y funcionalidad puede pasar directamente al hardware.

2.5 METODOLOGÍA

Existen diversos métodos que han planteado los profesionales del software para definir las actividades a llevar a cabo en un proyecto de desarrollo de sistemas. El presente proyecto utilizara la técnica de construcción por prototipos.

La construcción por prototipos se basa en la captura e implementación rápida de un conjunto inicial para posteriormente expandirlo y refinarlo.

Este método permite resolver acertadamente algunas situaciones muy comunes; generalmente el diseñador de una aplicación no sabe identificar realmente sus necesidades, lo cual hace que en la mayoría de casos los proyectos de ingeniería del software no tengan un desarrollo secuencial, sino iterativo que permite el ajuste de los programas a los nuevos requerimientos, sin embargo, el usuario solo puede ver una versión operativa del sistema al final del desarrollo del proyecto, lo cual en algunos casos genera situaciones de desesperación y problemas de autoestima.

La figura 19 muestra las fases representativas de esta metodología.

Como se puede apreciar la construcción de prototipos posee cinco etapas:

Recopilación de requerimientos: En esta etapa el usuario final y el desarrollador del sistema se reúnen y definen los objetivos globales del software.

Diseño rápido: Diseño de la presentación de las interfaces del sistema con el usuario final. Pantallas de captura y reportes de salida.

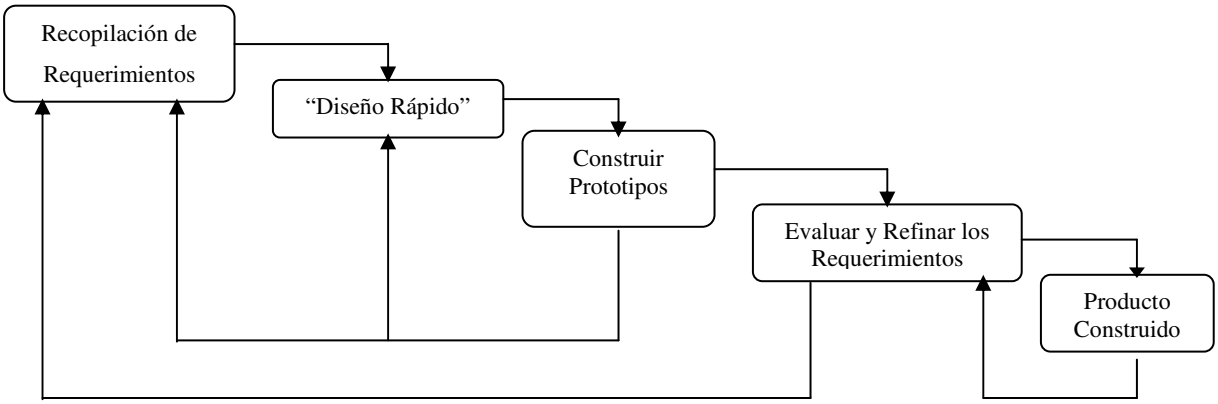


Figura 19. Fases Representativas del prototipado.

Construcción del prototipo: Implantación de un modelo no operacional del sistema básico para el mecanismo de refinamiento de los requerimientos del sistema final.

Evaluación y refinamiento del prototipo: Facilita al desarrollador una mayor comprensión en el análisis de necesidades. Permite llegar al producto final deseado.

Producto de ingeniería: Resultado, objetivo de la aplicación de la metodología; un producto final cumpliendo los requisitos planteados inicialmente y los emergentes durante el proyecto.

Para el presente proyecto se propone la construcción de tres prototipos.

Primer prototipo: Será capaz de recrear una partida de BattleTech utilizando el primer nivel de reglas, contenido en el manual de reglas (Rulebook) de la caja de BattleTech cuarta edición (BattleTech box set fourth edition); en este prototipo cada robot, así como el escenario serán representados por mapas de bits (bitmaps).

Segundo prototipo: Este prototipo incluirá un modelo preliminar en 3D del mapa en el cual se desarrollara el juego, además de los modelos en 3D de 4 de los robots que estarán en capacidad de intervenir en una partida del juego. Además de lo anterior se incluirá la opción de guardar la información del estado de una partida que no haya sido terminada en una sola sesión y se implementaran las ayudas de la herramienta y un prototipo de comunicación TCP/IP.

Tercer prototipo: Además de incluir las características de los anteriores prototipos, incluirá animaciones básicas (caminar, correr y saltar) para los robots durante la partida, el modelo definitivo del escenario en 3D y también serán mejoradas la estabilidad y la velocidad de la aplicación.

3. DISEÑO

En este capítulo se mostrara el diseño final del sistema, sin entrar en mucho detalle de cómo se llegó exactamente a este. Los pasos que se siguieron para pasar de un prototipo a otro se muestran en el capitulo siguiente, desarrollo del sistema.

El sistema se diseñó en dos capas lógicas (figura 20), permitiendo de esta forma aprovechar mejor la filosofía de orientación a objetos. Al estar diseñado en estas dos capas lógicas, que permiten tener la interfaz separada del motor de reglas, es posible cambiar la interfaz gráfica de usuario sin necesidad de cambiar el motor de reglas. De igual manera el motor puede sufrir cambios, mejoras, aumento de reglas, sin necesidad de hacer cambios en la interfaz gráfica. Una explicación más detallada de este diseño en dos capas lógicas se dará más adelante en este capítulo.

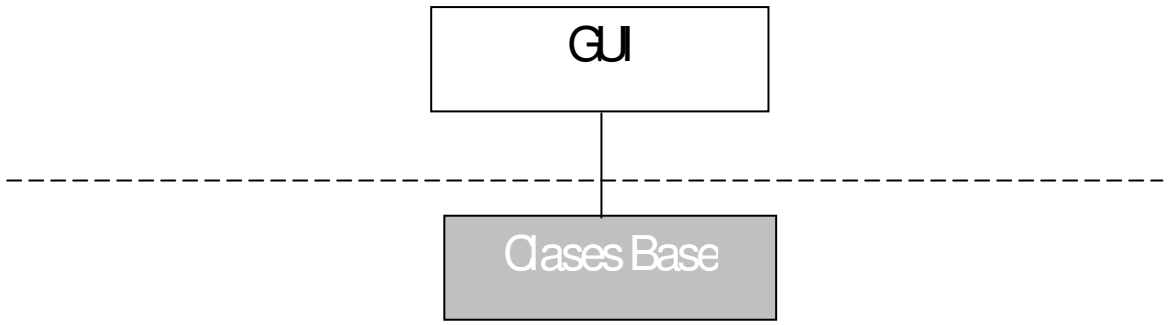


Figura 20. Capas lógicas del sistema.

La capa lógica que comprende las clases base se refiere a las clases que tiene que ver con todo el motor de reglas. Estas clases se encargan de mantener la información necesaria para poder desarrollar un partida el juego y también se encargan de validar los cambios en esta información que suceden durante el desarrollo del juego, es decir aplican todas las reglase estipuladas en el sistema de reglas de BattleTech cuarta edición. El apéndice A muestra unas reglas introductorias al juego que son suficientes para entender el desarrollo de una partida.

3.1 CASOS DE USO DE DISEÑO.

El primer acercamiento al sistema y los requerimientos de este se logró gracias a los diagramas de casos de uso de las primeras etapas de desarrollo del primer prototipo. Estos diagramas fueron creciendo hasta obtener el diagrama de casos de uso final que es el que representa el sistema implementado. Este diagrama de casos de uso se muestra en la figura 21.

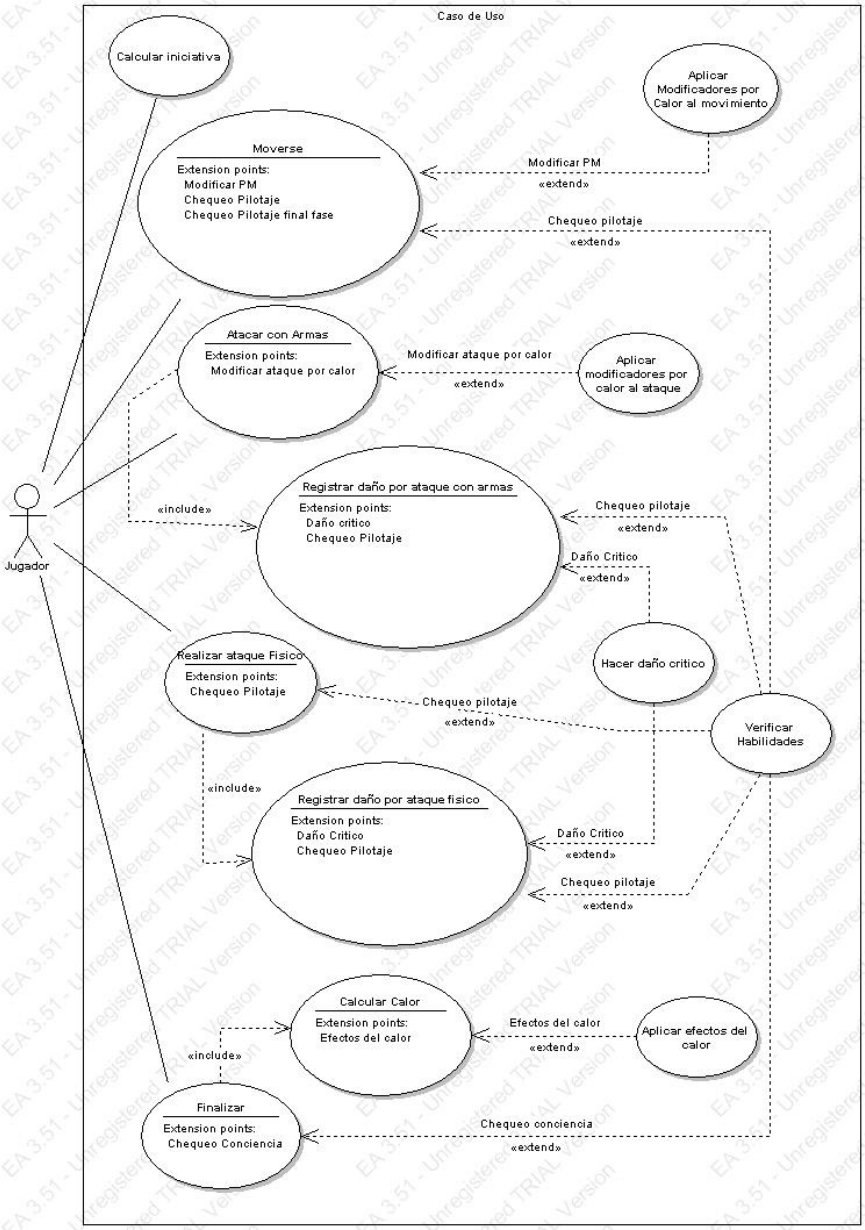


Figura 21. Casos de uso Diseño.

Es importante aclarar que los casos de uso no son solo esos pequeños óvalos con una inscripción dentro atados de una manera lógica a otros. Un caso de uso describe lo que hace un sistema o un subsistema sin especificar como.

Debido a esto es poco probable que el dibujo del ovalo y el verbo que lo describe sean suficientes, por eso es necesario describir el flujo de eventos que ocurren dentro del caso de uso. Esta descripción debe incluir cuando el flujo comienza, cuando termina, cuando el caso de uso interactúa con los actores, cuando hay intercambio de objetos y debe mostrar el flujo base y los flujos alternativos.

Como el diagrama de casos de uso no es suficiente para explicar un diagrama de casos de uso, aquí tenemos un escenario que nos muestra como es que este diagrama de casos de uso se comporta y nos ayuda a comprender la lógica del sistema y los requerimientos de este.

La documentación de este diagrama se hizo con Entreprise Architect[32] (EA) y se generó desde este programa. Esta opción del programa es muy útil, genera el gráfico del diagrama en diversos formatos de imagen (bmp, jpg, emf, wmf, gif o png) y genera un documento en formato de texto enriquecido (rich text format, *.rtf) que contiene toda la información que el realizador del diagrama introdujo dentro de este. Esto evita que el diagrama esté en una herramienta y la documentación en otra. Para este caso, toda la información se encuentra dentro del programa bajo el estándar de UML.

Antes de entrar en detalles sobre cada uno de los casos de uso del diagrama, hay que explicar lo que se muestra en la documentación generada por el programa EA. La explicación de la documentación generada por EA se muestra en la tabla 1.

La información mostrada en la tabla 1 es lo que generalmente aparece en la documentación, sin embargo es posible que aparezca información adicional que en su momento se explicará que significa y por qué aparece. UML es muy versátil y tiene muchas opciones que no siempre son necesarias utilizar.

[32] Enterpraise Architect 3.51 de Sparx Systems. Herramienta CASE para modelado con UML.

Tabla 1. Descripción documentación producida por EA

Aplicar efectos del calor	Nombre del caso de uso
Type: *public* **Use Case**	Tipo. Puede ser publico, privado, protegido. Puede ser clase, caso de uso, etc.
Status: Approved. Version 1.0. Phase 1.0.	Estado. EA permite llevar un control del estado de cada una de las partes de un diagrama. El estado puede ser propuesto, aprobado, obligatorio, validado, implementado. Además permite llevar control de versiones y de fases.
Package: Use Case Model Análisis	Paquete que contiene el diagrama. Esta es una organización lógica de los diagramas dentro de EA.
Constraints ▪ Mandatory Pre-condition. Calor > 14.	Restricciones. Pueden ser precondiciones o poscondiciones.
Connections ▪ Extend link to use case *Calcular Calor*	Conexiones. Conexiones lógicas que existen para el caso de uso. Pueden ser cualquier tipo de relación UML. Las más comunes son las relaciones de inclusión o de extensión.
Scenarios Basico {Basic Path}. 1. Si calor > 14, cálculo de lanzamiento para evitar desconexión 2. Si lanzamiento falla, aplica efectos de desconexión 3. Si calor > 19, cálculo de lanzamiento para evitar explosión de munición 4. Si lanzamiento falla, aplica efectos de explosión de munición	Escenarios. Escenarios para el caso de uso. Para la descripción de los escenarios no existe un estándar, sin embargo se esta usando la estructura presentada en la guía de usuario de UML. En el escenario cuando se muestra una frase entre paréntesis se refiere a una extensión. Si se encuentra la palabra include corresponde a una inclusión.

La descripción de los casos de uso de la figura 21 se muestra a continuación.

Use Case model Design

Type: *public* **Package**
Status: Proposed. Version 1.0. Phase 1.0.

Package: Use Case View

Jugador

Type: *public* **Actor**
Status: Proposed. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Association link to usecase *Moverse*
- Association link to usecase *Atacar con Armas*
- Association link to usecase *Realizar ataque Físico*
- Association link to usecase *Finalizar*
- Association link to usecase *Calcular iniciativa*

Aplicar efectos del calor

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Extend link to usecase *Calcular Calor*

Scenarios

Aplica efectos por exceso de calor {Basic Path}.
Debe realizar unos chequeos para evitar la desconexion o la explosion de municion. si estos fallan, dependiendo de la escla de calor desconecta el mech o aplica los daños producidos por la explosion de la municion.

Aplicar modificadores por calor al ataque

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Extend link to usecase *Atacar con Armas*

Scenarios

Aplica modificador al disparo por calor {Basic Path}.
Aplica modificador al disparo dependiendo de la escala de calor.
8 +1
13 +2
17 +3
24 +4

Aplicar Modificadores por Calor al movimiento

Type: *public* **Use Case**

Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Extend link to usecase *Moverse*

Scenarios

calor en 5 o mayor {Basic Path}.
Debe tener en cuenta que el calor mayor a 5 disminuye la cantidad de puntos de movimiento disponibles.
Aplica los modificadores correspondientes por exceso de calor para el movimiento. El modificador varia desde -1 hasta -5 dependiendo de la escala de calor
5 -1
10 -2
15 -3
20 -4
25 -5

Atacar con Armas

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Association link from actor *Jugador*
- Extend link from usecase *Aplicar modificadores por calor al ataque*
- Include link to usecase *Registrar daño por ataque con armas*

Scenarios

Disparar Armas {Basic Path}.
Selecciona las armas que desea disparar y los objetivos a los cuales disparara. Dispara las armas y aplica el daño que corresponda a los objetivos escogidos.

Exceso de calor (>8) {Alternate}.
Si el calor es mayor a 8 debe aplicar el modificador al disparo que corresponda según la escala de calor.
<<extend>> Aplicar modificadores por calor al disparo

Registrar daño recibido {Alternate}.
Si fue atacado e impactado debe registrar el daño que recibió.
<<extend>> Registrar Daño recibido por ataque con armas

Calcular Calor

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Extend link from usecase *Aplicar efectos del calor*
- Include link from usecase *Finalizar*

Scenarios

Calcular el calor {Basic Path}.
Calcula el calor producido en el turno dependiendo del movimiento realizado, de las armas disparadas, del calor no disipado en el turno anterior y de la cantidad de radiadores disponibles.

Calor >= 14 {Alternate}.
El exceso de calor produce unos efectos especiales.
si calor >= 14
<<extend>>Aplicar efectos del calor

Calcular iniciativa

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Association link from actor *Jugador*

Scenarios

Calcular iniciativa {Basic Path}.
los 2 jugadores lanzan 2D6 y el jugador que obtenga el mayor resultado será el ganador de la iniciativa. Este jugador moverá después del jugador que perdió la iniciativa.

Finalizar

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Association link from actor *Jugador*
- Extend link from usecase *Verificar Habilidades*
- Include link to usecase *Calcular Calor*

Scenarios

Finalizar turno {Basic Path}.
Calcula el calor.<<extend>>Calcular Calor
Los mechs que pivotaron torso regresan a su posición original.
Realiza los chequeos que correspondan. (conciencia o reconexion).
<<extend>> Verificar habilidades

Hacer daño critico

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Extend link to usecase *Registrar daño por ataque con armas*
- Extend link to usecase *Registrar daño por ataque físico*

Scenarios

Hacer daño critico {Basic Path}.
Daña el componente interno que corresponda y aplica los efectos generados por este daño.

Moverse

Type: *public* **Use Case**
Status: Proposed. Version 1.0. Phase 1.0.
Package: Use Case model Design

Constraints

- *Mandatory Pre-condition* . Debe corresponder al orden de iniciativa.

Connections

- Association link from actor *Jugador*
- Extend link from usecase *Aplicar Modificadores por Calor al movimiento*
- Extend link from usecase *Verificar Habilidades*

Scenarios

moverse {Basic Path}.
Permite que el mech se mueva dentro de la retícula hexagonal. Debe escoger el tipo de movimiento que desea realizar (caminar, correr o saltar). Debe gastar los puntos de movimiento según corresponda al terreno y al los cambios de encaramiento. Debe conservar el tipo de movimiento que se realizo para futuras referencias dentro del turno. Valida la regla de apilamiento y de paso por hexágonos ocupados por mechs aliados.

moverse 2 {Alternate}.
1. Escoge el mech que desea mover
2. Escoge el tipo de movimiento. (quieto, caminar, correr, saltar)
3. (Modificar PM) Ocurre si Calor > 5
3. Revisa que el hexágono por el que desea pasar no este ocupado por un mech enemigo
4. (Chequeo de pilotaje) Si el hexágono por el que pasa es agua
4. Gasta los pm dependiendo del tipo de movimiento.
5. Calcula el modificador ganado por número de hexágonos movidos
6. Revisa que el hexágono en el que termine el movimiento no este ocupado por otro mech.
7.(Chequeo de pilotaje) Si hay daño en cadera o giró y corrió

calor mayor a 5 {Alternate}.
<<extend>> Aplicar modificadores por calor al movimiento

daño en el gyro o hip {Alternate}.

Si corre o salta y tiene el giro o la cadera dañada debe realizar un chequeo de pilotaje al final del turno.<<extend>> Verificar habilidades

pasa por hexagono de agua {Alternate}.
Debe realizar un chequeo de pilotaje cada vez que pase por un hexágono de agua con un modificador de -1 al pilotaje.
<<extend>> Verificar habilidades

Realizar ataque Físico

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Association link from actor *Jugador*
- Extend link from usecase *Verificar Habilidades*
- Include link to usecase *Registrar daño por ataque físico*

Scenarios

Realizar Ataque físico {Basic Path}.
Realiza el ataque físico dependiendo de la situación.
posibles ataques:
Patada o puño. Depende de las acciones realizadas durante el turno y del terreno en que se encuentre el atacante y el objetivo.

Falla una patada {Alternate}.
Si el ataque realizado fue una patada no exitosa, debe realizar un cheque de pilotaje.
<<extend>> Verificar habilidades

Registro de daño por ataque físico {Alternate}.
Si fue atacado por otro jugador con éxito debe registrarse el daño recibido por el ataque físico.
<<extend>>Registrar daño recibido ataque físico

Registrar daño por ataque con armas

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Extend link from usecase *Verificar Habilidades*
- Extend link from usecase *Hacer daño critico*
- Include link from usecase *Atacar con Armas*

Scenarios

Registra el daño recibido {Basic Path}.

Registra el daño recibido dependiendo del arma y de la localización impactada. Controla la transferencia de daño.

Hacer daño critico(estructura interna) {Alternate}.
<<extend>> Hacer daño critico
Si se produce daño critico aplica el daño a la parte interna que corresponda.

Daño recibido > 20 {Alternate}.
Con un daño recibido mayor o igual a 20 puntos es necesario chequear pilotaje (habilidad).
<<extend>>Verificar habilidades

Registrar daño por ataque físico

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Extend link from usecase *Verificar Habilidades*
- Extend link from usecase *Hacer daño critico*
- Include link from usecase *Realizar ataque Físico*

Scenarios

Registra el daño recibido por ataque físico {Basic Path}.
Registra el daño recibido dependiendo del tipo de ataque (patada o puño) y de la localización impactada. Controla la transferencia de daño.

Apilca daño interno {Alternate}.
Si se produce daño critico aplica el daño a la parte interna que corresponda.
<<extend>> Hacer Daño Critico

Verificar Habilidades

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case model Design

Connections

- Extend link to usecase *Moverse*
- Extend link to usecase *Registrar daño por ataque con armas*
- Extend link to usecase *Realizar ataque Físico*
- Extend link to usecase *Registrar daño por ataque físico*
- Extend link to usecase *Finalizar*

Scenarios

chequeo de habilidad {Basic Path}.
Realiza un chequeo de la habilidad que corresponda. Dependiendo de la situación se debe chequear pilotaje o conciencia.

3.2 DIAGRAMA DE CLASES DISEÑO.

Partiendo de los respectivos diagramas de casos de uso que se elaboraron durante el desarrollo de cada uno de los prototipos se hicieron modificaciones a la estructura de clases que representa el sistema de reglas, hasta llegar al diagrama de clases mostrado en la figura 22.

A este diagrama se llegó utilizando varias técnicas de diseño orientado a objetos presentadas en los libros de Grady Booch[33] y Meilir Page-Jones[34]. Dentro de estos lineamientos que se tuvieron en cuenta para el diseño de las clases se pueden señalar:

- Alta cohesión dentro de las clases
- Bajo acoplamiento entre clases
- Principio de la Encapsulación
- Principio de Connascence. Meilir Page-Jones utiliza el término connascence, que significa "haber nacido juntos" y cuya posible traducción sería congénito, para describir un tipo de acoplamiento entre clases que viene dado por la forma como las clases acopladas nacen dentro del proceso de diseño del sistema y dentro de la aplicación misma.
- Principio de la sobrecarga de Clases (encumbrance como lo llama Meilir) como medida de la sofisticación de una clase. Entre mas especializada sea la clase, esta tendrá mas nexos con otras clases, lo que hará que bajen las posibilidades de reutilización de la clase. Se entiende por nexo las relaciones de herencia, de agregación o composición entre otras.

[33] Diseño Orientado A Objetos. Grady Booch. Traducción: Jaime O. Albarracin. Universidad Industrial de Santander. Bucaramanga. 1998.
[34] Fundamentals Of Object-Oriented Design In Uml. Meilir Page-Jones. Addison Wesley. New York 2000.

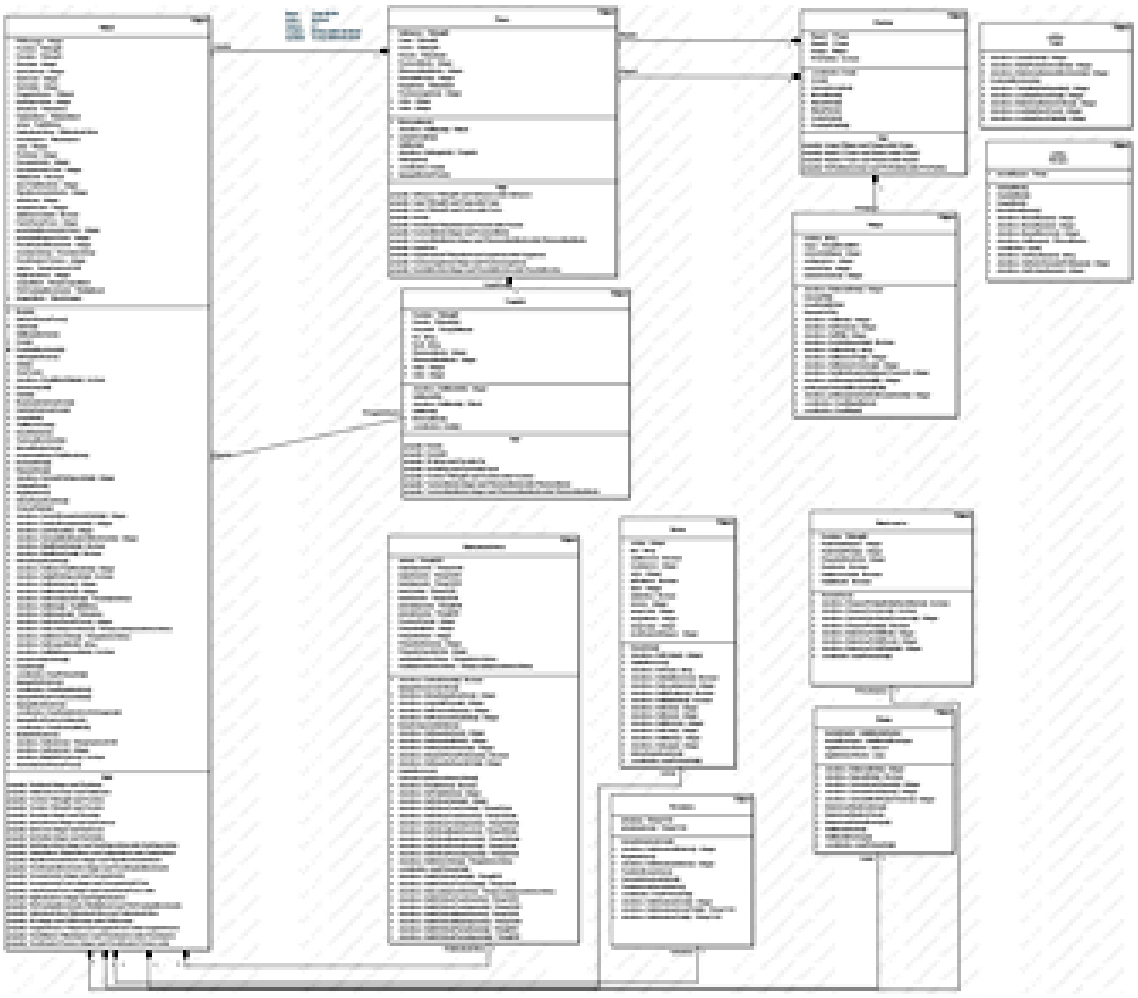


Figura 22. Diagrama de Clases.

- La ley de Demeter. La ley de Demeter dice lo siguiente: Para un objeto *obj* de la Clase *C* y para cualquier operación *op* definida para *obj,* cada objetivo de un mensaje dentro de la implementación de *op* debe ser uno de los siguientes objetos:
  1. El objeto *obj* mismo.
  2. Un objeto referenciado dentro de la signatura de *op.*
  3. Un objeto referenciado por una variable de *obj*.
  4. Un objeto creado por *op*.
  5. Un objeto referenciado por una variable global

- Principio de espacio de Estados y comportamiento.

Durante el desarrollo del modelo de clases para el primer prototipo se llegó a la solución de crear un sistema con dos capas lógicas dentro del juego, una de ellas seria la que representan las clases base de la figura 22 que constituyen el motor de reglas y la otra capa es la que tiene que ver con la interfaz gráfica. Una ventaja adicional de tener estas dos capas es la de poder cambiar la interfaz gráfica sin tener que hacer ningún cambio a las clases base, encapsulación de la información. Para el desarrollo del proyecto este aspecto fue muy importante ya que la interfaz gráfica del primer prototipo es en 2D y la del segundo y tercero es en 3D.

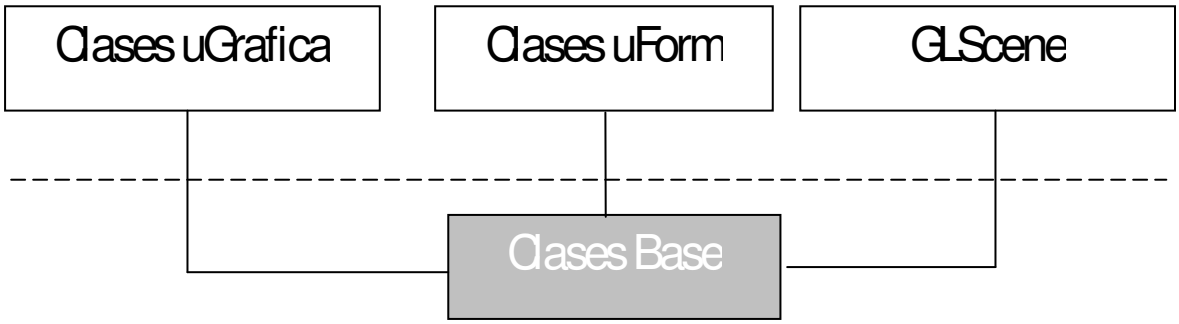


Figura 23. Capas lógicas del sistema.

Es importante aclarar algunos aspectos de la figura 23. Las clases de uGrafica, son clases especialmente diseñadas para mostrar información del sistema. La mayor parte de estas clases son descendientes de la clase TGraphic de Delphi.

Las clases de uForm son las clases descendientes de TForm (esta es la clase formulario de Delphi) y de los clases de la VCL[35] de Delphi utilizados.

Las clases de OpenGL se refieren a las clases de GLScene[36] y que se usan para mostrar las imágenes en 3D.

Por ultimo las clases base son las que se muestran en la figura 22.

[35] Visual Component Library. Librería de componentes Visuales de Delphi.
[36] Librería para Delphi basada en OpenGL. http://glscene.org.

Dentro del comportamiento de las clases base es importante destacar los cambios de estado por los que pasa TPartida que es la clase que controla el desarrollo del juego y que muestra el ciclo principal del juego. El diagrama de estados de esta clase se muestra en la figura 24.

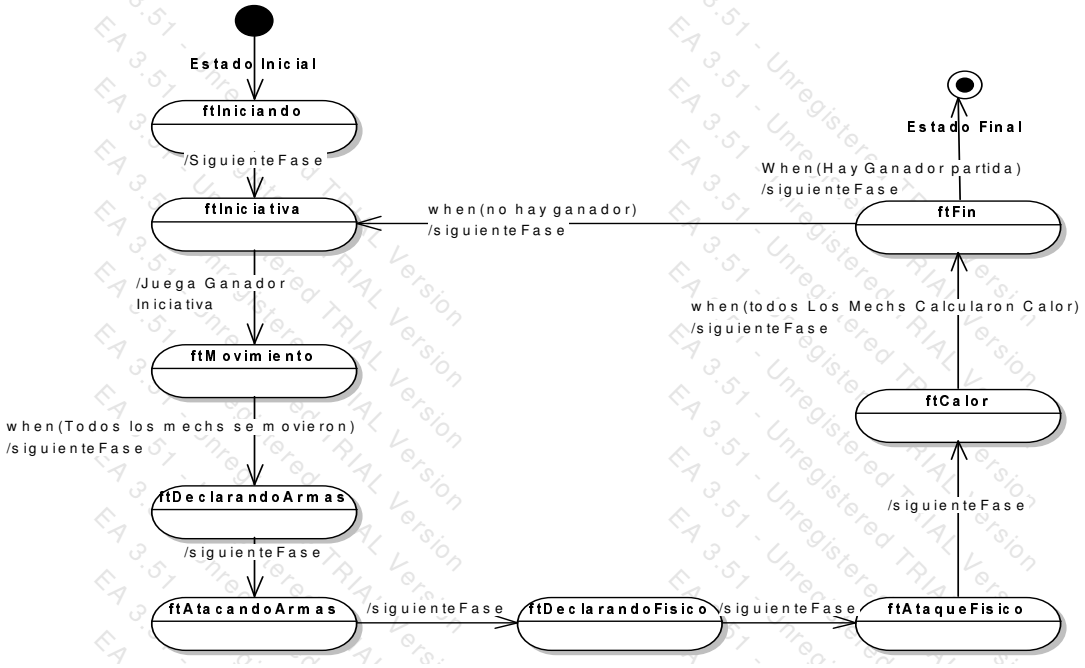


Figura 24. Estados de partida.

Los estados de la clase partida corresponden casi exactamente a las fases de un turno dentro del juego de battletech.

Un aspecto muy importante del diseño de la herramienta fue la utilización de Spider Containers And Persistent Classes[37] para permitir que las clases base del sistema fueran objetos persistentes, es decir se guardan en disco como objetos en un archivo binario, no hay ningún tipo de traducción a una base de datos entidad relación o similar. Otra de las ventajas de Spider Containers And Persistent Classes es que al ser una librería de clases para Delphi, ésta queda incorporada dentro del sistema al compilar, al quedar incorporada la librería dentro de la aplicación, evita que se tengan

[37] Copyrighted © 1996,97,98 Michel Brazeau. Interval Software. http://www.cam.org/~mibra/spider

que utilizar herramientas adicionales de bases de datos para guardar toda la información de las partidas, permitiendo así que el sistema sea independiente de cualquier otra herramienta.

La estructura de los objetos guardados con esta herramienta se conserva, así que si se desea guardar una clase que es un agregado de otras clases, solo es necesario guardar la clase que las contiene y las clases contenidas serán guardadas. Desde luego al momento de codificar esta funcionalidad se deben crear funciones de escritura para cada una de las clases que se desea que sean persistentes, pero esto solo se hace una vez.

3.3 DISEÑO CLIENTE / SERVIDOR.

Para el desarrollo del sistema de comunicación se utilizó un componente llamado ICS - Internet Component Suite[38] desarrollado por François Piette. Se escogió porque utiliza una jerarquía de objetos, es gratuito con código fuente y ha sido ampliamente probado por la comunidad en Internet.

Otras de las características de este componente son:

- La clase básica de ICS es la clase TWSocket que encapsula el paradigma de los sockets de Windows, esto demuestra que la estructura de ICS es orientada a objetos.
- TWSocket es un componente completamente asíncrono (no-bloqueante), y manejado por eventos.

El diseño del sistema de comunicaciones de la aplicación mantiene copias de las clases base del lado del cliente y del servidor para así poder ofrecer la funcionalidad necesaria para hacer cumplir las reglas, si embrago, la evaluación de las reglas en algunos casos se hace del lado del cliente y en otros del lado del servidor. Esta característica se agregó para evitar la sobrecarga del servidor y permitir el diseño de un sistema de mensajes cortos.

Para entender mejor el funcionamiento del servidor, la figura 25 muestre el diagrama de estados del servidor.

[38] http://www.overbyte.be. François Piette francois.piette@overbyte.be

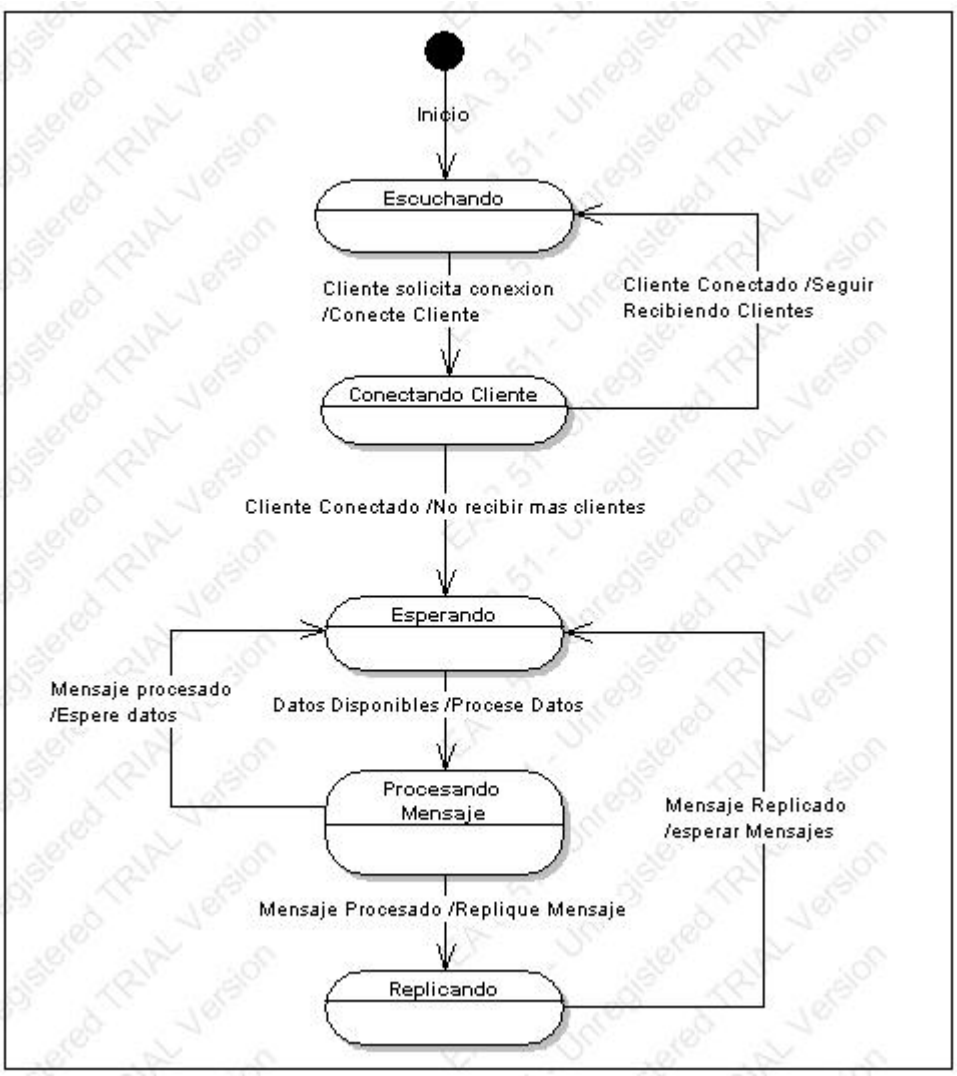


Figura 25. Diagrama de estados del servidor.

Lo que sucede con el servidor es lo siguiente:

- El servidor esta escuchando, listo para recibir clientes.
- Si algún cliente solicita conexión, lo conecta.
- Cuando el cliente esta conectado puede recibir la orden de seguir recibiendo clientes.
- Cuando el cliente esta conectado puede recibir la orden de no recibir mas clientes.
- En caso de no recibir mas clientes, se queda esperando por información en el buffer de datos recibidos
- Si hay datos en el buffer, los procesa.
- Si hay necesidad de reenviar los datos recibidos a todos los jugadores los reenvía.

El servidor después de procesar un mensaje siempre pasa al estado esperando.

El servidor recibe un mensaje que tiene la estructura que muestra la figura 26 en el nivel de la aplicación.

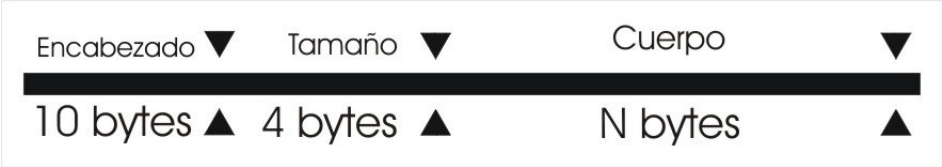


Figura 26. Estructura de un mensaje.

Las partes de este mensaje son:

- Encabezado: Este es un identificador de 10 bytes de tamaño. Lo usa el servidor para clasificar los mensajes y dependiendo del tipo de mensaje lo procesa.
- Tamaño: Este valor de tamaño se refiere al tamaño del cuerpo del mensaje (siguiente sección del mensaje), no del mensaje total
- El cuerpo del mensaje es de tamaño variable, su tamaño exacto esta definido en tamaño, y dependiendo del tipo de mensaje puede ser cero.

Al recibir un mensaje el servidor lo primero que hace es examinar la cabecera del mensaje que contiene un identificador de mensaje, este identificador le permite al servidor saber como debe procesar el mensaje. Dependiendo de esta cabecera el servidor sabe si debe leer el resto del contenido del mensaje y a donde enviar la información contenida en este.

Dependiendo del tipo de mensaje, si lo amerita, el servidor luego leerá los siguientes 4 bytes del mensaje que corresponden al tamaño del cuerpo del mensaje, es decir corresponden a los bytes que vienen después del byte numero 14 hasta el byte final del mensaje que será el byte numero N + 14.

Por ultimo si el mensaje contiene un cuerpo, el servidor leerá los N bytes que le falta leer para haber leído por completo el contenido del mensaje.

El cliente posee la misma funcionalidad, la diferencia fundamental es que no debe replicar ninguna información.

El diseño final del sistema cliente y servidor se muestra en el diagrama de despliegue que aparece en la figura 27.

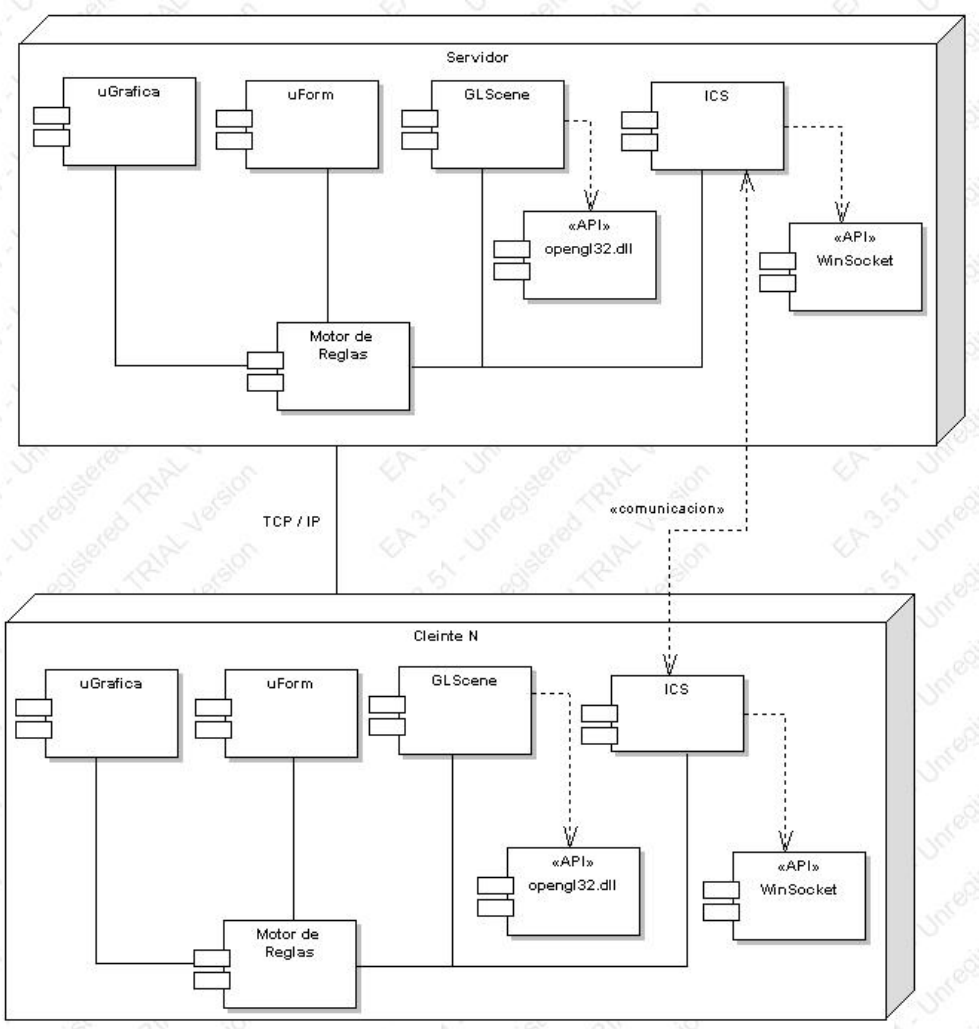


Figura 27. Diagrama de despliegue*[39] de la aplicación.

[39] Esta palabra se traduce de diferentes formas dentro de los libros de UML, el nombre de este diagrama dentro de UML es Deployment Diagram.

3.4 DISEÑO AMBIENTE TRIDIMENCIONAL.

Para el desarrollo del ambiente tridimensional se utilizó una librería llamada GLScene[40]. GLScene es una librería basada en OpenGL para Delphi que provee componentes visuales y objetos que permiten la geneneración de escenas tridimensionales que se distribuye como código abierto bajo la licencia Mozilla Public Licence[41]. Esta librería ha crecido tanto que ya no es solo una librería de utilidades o una envoltura (wrapper) de OpenGL sino que es un grupo de clases base para un motor genérico tridimensional.

Además de esta librería se utilizó 3ds max[42], como herramienta de modelado en 3d y que permitió además generar animaciones, utilizando la técnica de esqueletos, de cada uno de los cuatro robots incluidos en el juego. Estos modelos y animaciones fueron incorporados utilizando el formato de archivo SMD utilizado por el juego Half-life[43]. Estos archivos son archivos de texto que guardan la geometría del modelo o las animaciones de este de forma independiente, es decir, se tiene un archivo para la geometría (malla) y un archivo por cada animación que se tenga disponible. Tener archivos independientes permite hacer un mejor manejo de la memoria del sistema.

La manera de obtener los archivos SMD, partiendo de 3ds max, fue a través exportaciones hechas con plug-ins y scripts liberados extraoficialmente por usuarios de este programa. Uno de los plug-ins utilizado es una variación del plug-in incluido con el software development kit (SDK) de Half-Life.

Las clases que se encargan de mostrar el ambiente tridimensional se mantuvieron aparte de las clases base, como muestra la figura 23. La mayoría de objetos utilizados para generar este ambiente son descendientes de una clase de GLScene.

Cada uno de los cuatro modelos realizados para el sistema fueron rediseñados (variaciones de los presentados en la caja del juego) y realizados, al igual que cada una de sus 10 animaciones, por

[40] http://glscene.org
[41] http://www.mozilla.org/MPL
[42] Programa de modelamiento y animación en 3D. Copyright © 2002 Autodesk, Inc. All rights reserved.
[43] Valve software. www.valvesoftware.com.

Camilo Vargas, quien fue el diseñador industrial que estuvo colaborando en el desarrollo del ambiente en 3D al igual que en el desarrollo de la interfaz gráfica.

Cada uno de los robots realzados para el sistema posee 14 huesos y entre 2500 y 3000 polígonos. Uno de los modelos se muestra en la figura 28.

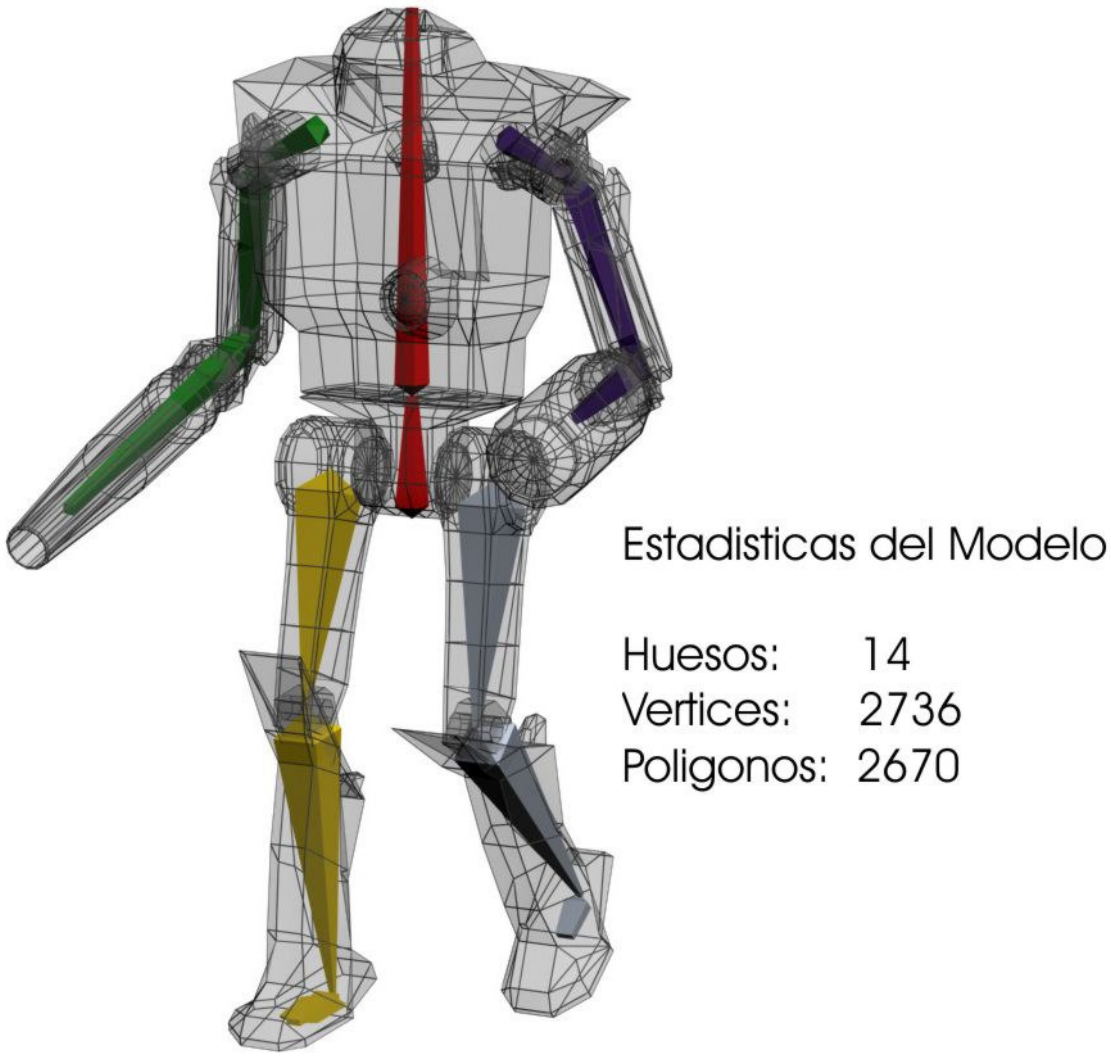


Figura 28. Mech con esqueleto a la vista.

Para crear cada una de las 10 animaciones que forman parte de cada uno de los mechs, fue necesario realizar un diagrama de estados en el que se representa el cambio entre animaciones, así como cada una de las animaciones usadas dentro del sistema.

La figura 29 muestra el diagrama de estados de las animaciones.

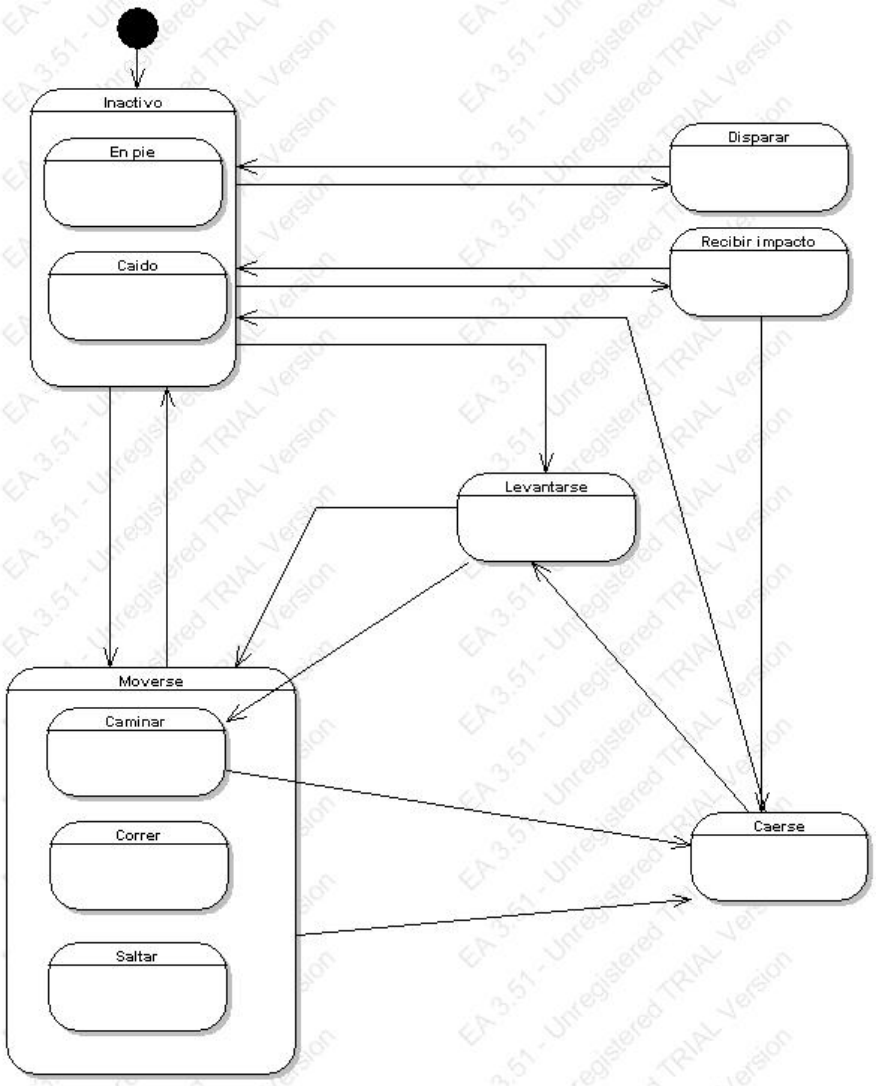


Figura 29. Diagrama animación de movimientos

3.5 HERRAMIENTAS UTILIZADAS

Las herramientas utilizadas para el desarrollo de este sistema fueron:

- Lenguaje de desarrollo: Borland Delphi 5 Professional.

- Desarrollo Interfaz Grafica de Usuario (GUI):
  - VCL (Visual Component Library) de Delphi: Librería visual contenida dentro de delphi 5. Además de los componentes visuales que vienen por defecto en esta librería se utilizaron componentes diseñados especialmente para el sistema, algunos componentes de distribución gratuita con código fuente y modificaciones a componentes gratuitos.
  - OpenGL.
  - 3D Studio max 5: Herramienta de modelado tridimensional que permite la creación de lugares, personajes y objetos de cualquer tipo. Permite la animación y visualizacion de los objetos modelados. Esta herramienta es la herramienta lider en modelado en 3D para juegos en la industria, el 80% de las empresas productoras de juegos usan esta herramienta.
  - Corel Draw 10: CorelDRAW es una aplicación para el diseño gráfico, con esta herramienta es posible crear anuncios y material gráfico para imprenta, Web o en diferentes tipos de archivos como jpg, bmp, entre otros.

- Asesoría y diseño de objetos en 3D e interfaz en 2D a cargo de Camilo Vargas. Diseñador Industrial de la UIS.

- UML: Enterprise Architect version 3.51. Herramienta CASE para diagramación en UML. Permite generación de código, actualización de diagramas a partir de código, realización de diagramas. Generacion de documentación de los diagramas en creados en esta herramienta.

4. DESARROLLO

4.1 METODOLOGIA

La metodología con la cual se planteó el desarrollo del proyecto fue la del prototipado. Esta metodología se escogió porque permitía ver resultados parciales y además permitía escalar el software a medida que se pasaba de un prototipo a otro, sin embargo durante el desarrollo del proyecto esta metodología cambió y tomó prestadas varias fases del Proceso Unificado[44]. Esto ocurrió debido a que durante el planteamiento del proyecto no se tenía suficiente conocimiento sobre el proceso unificado, pero a medida que se desarrollo el proyecto se empezó un estudio de éste y debido a esto se descubrió que había muchos aspectos que podían ayudar en el desarrollo del presente proyecto. Además, el proceso unificado presenta lineamientos muy claros sobre la elaboración de ciertos diagramas de UML en ciertas etapas y esto fue de gran ayuda ya que para el desarrollo del proyecto se planteó la utilización de UML como lenguaje de modelado.

La metodología realmente utilizada durante el desarrollo del proyecto se presenta en la figura 30.

Aunque algunos desarrolladores de software prefieren apegarse por completo a una metodología y seguir cada uno de sus pasos lo más fielmente posible, en este caso pareció pertinente hacer estos cambios por varias razones.

Una de las razones por las que se hicieron cambios tiene que ver con el uso de UML. Al empezar a conocer UML sin colocarlo dentro de ninguna metodología especifica (UML no depende ni tiene que ser utilizado con ninguna metodología, es independiente de la metodología) se puede inferir que el diagrama de casos de uso, al igual que el diagrama de clases se deben realizar dentro de las primeras fases de desarrollo y que el diagrama de clases, de objetos, de interacción (secuencia, colaboración) y de estados (en algunos casos) deberían realizarse dentro de etapas intermedias y

[44] The Unified Software Development Process. Ivar Jacobson, Grady Booch, James Rumbaugh. Addison Wesley Object Technology Series. 1999.

quizás finales del desarrollo, de hecho algunos autores hacen estas sugerencias dentro de los libros que tratan UML. Sin embrago a veces inferir esto al empezar a conocer UML no es suficiente y al momento de desarrollar el sistema se puede perder el significado del diagrama en el afán de reconocer en que momento realizarlo. En este momento es de gran ayuda el proceso unificado que muestra qué diagramas realizar y por qué realizarlos, que fue lo que ocurrió en este caso como lo muestra la figura 30. Diagramas de casos de uso durante la elaboración (etapa inicial), clases, interacciones durante la construcción.

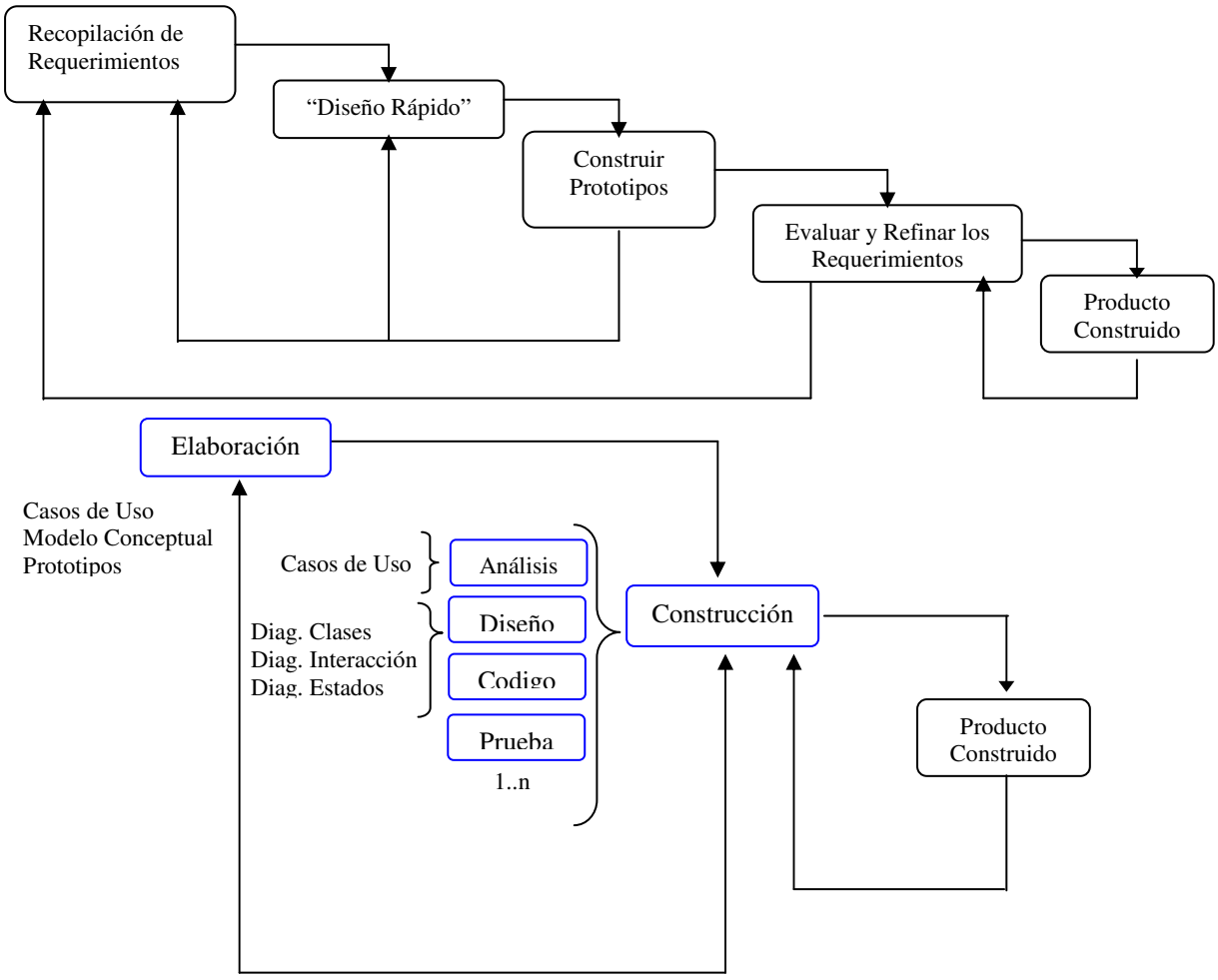


Figura 30. Metodologías.

Otra de las razones por las que se modificó la metodología de prototipado fue la de beneficiarla con el prototipado mismo. En el momento de empezar a utilizar prototipado lo primero que se hace es

fijar un número determinado de prototipos; con características muy específicas, para este caso fueron tres prototipos, sin embargo a veces es necesario realizar prototipos intermedios que no quedan documentados por no cumplir con ninguno de los requisitos establecidos. En el caso del proceso unificado es posible fijar un número determinado de prototipos antes de llegar a una de esos tres prototipos importantes. En este caso la modificación se realizó en la fase de construcción que inicialmente era solo de diseño rápido pero que se modificó para incluir análisis, diseño y prueba. En el prototipado se plantea un diseño rápido pero a veces se termina haciendo algo de análisis y se adoptó la idea de realizar pruebas en esta etapa por la posibilidad de propagar errores difíciles de rastrear entre cada uno de los prototipos planteados. Esta adopción es exacta a como funciona el proceso unificado y también tiene que ver con los consejos dados por algunos autores con Martín Fowler (en Refactoring Software[45] y en UML Distlled[46]) de hacer pruebas exhaustivas de partes de código antes de ser incluidas dentro del sistema.

Además de lo anterior, el proceso unificado ofrece la posibilidad de realizar prototipos de métodos u objetos identificados como críticos o de un grado alto de dificultad en cualquier fase de desarrollo del sistema. Aunque el prototipado no dice que no se puedan hacer, el proceso unificado incluye estos prototipos dentro de la metodología, permite que queden documentados y que se realicen en la etapa de elaboración. Para el caso de este proyecto ocurrió con la clase Tmapa, específicamente con el método que calcula los hexágonos por los que pasa una línea recta trazada desde el centro de un hexágono al centro de otro que esta a más de un hexágono de distancia. Este método se diseñó e implemento incluso antes de tener completamente realizado el diagrama de clases de análisis del sistema que es uno de los primeros diagramas que se deben realizar. La ventaja de haber realizado este método antes de todo lo demás tiene que ver con la identificación temprana de cuellos de botella en el desarrollo del proyecto y con la posibilidad de incluir ayuda de otros desarrolladores para el desarrollo del proyecto y no quedar atascado demasiado tiempo en un problema específico por no haberlo identificado en una etapa temprana.

Otras razones menores por las cuales se incluyeron aspectos del proceso unificado dentro de la metodología inicialmente utilizada tienen que ver con la orientación a objetos que es evidente en el proceso unificado y que ayudo a aclarar dudas de diseño e implementación durante el desarrollo del

[45] Refactoring: Improving the design of existing code. Martin Fowler. Addinson Wesley. 1999.
[46] UML Distilled: A brief guide to the standard object modeling. Martin Fowler. Addison Wesley. 2000.

proyecto y que no es evidente en el prototipado por ser una metodología que puede ser utilizada con cualquier filosofía de programación.

Aunque ninguna de las dos metodologías habla de alguna fase especifica para actualizar los diagramas y los diseños planteados (debería aparecer al final de la fase de construcción por ejemplo) durante el desarrollo se realizaron actualizaciones a los diagramas de clases cada vez que se obtenía un producto de importancia, esta labor no tomaba mucho tiempo gracias a Enterprise Architect[47] herramienta CASE que permite actualizar estos diagramas a partir del código en Delphi.

La conclusión que se puede obtener de la metodología utilizada para el desarrollo de este proyecto es que lo realmente importante en el desarrollo moderno de software es ser lo mas practico posible y establecer una metodología que permita escalar con rapidez el producto de software que se esta realizando. Ya no es tan importante tomar una gran cantidad de requerimientos sino mas bien tomar los requerimientos que representan la esencia del sistema y poder hacer crecer el sistema para que de solución a nuevos problemas relacionados con el sistema original o a aquellos requerimientos que inicialmente no fueron tenidos en cuenta pero que pueden ser incluidos en posteriores versiones finales o prototipos. Se necesita un desarrollo iterativo e incremental del software y aunque suene a proceso unificado no necesariamente debe utilizarse esta metodología para llegar a un buen resultado. Para el caso de este proyecto se tuvo en cuenta esto y por eso se hicieron las modificaciones ya explicadas.

Otro aspecto importante que merece ser mencionado, tiene que ver con UML y su impacto en el desarrollo de la metodología utilizada. UML es de gran ayuda para poder hacer fluir el proyecto por la metodología. A pesar de que puede llegar a ser una herramienta muy útil en todas las fases de la metodología puede también convertirse en un problema y puede retrasar el desarrollo del sistema si no se tiene cuidado. UML es un lenguaje que permite crear diagramas que pueden ser tan vagos o tan detallados como lo quiera el equipo de desarrollo lo cual es un arma de doble filo, se puede caer en exceso de diseño creando diagramas demasiado detallados que consumen tiempo precioso de desarrollo, que para el caso de este proyecto ocurrió en el desarrollo del primer prototipo, o diagramas demasiado vagos que no aportan nada al proyecto y pueden generar demoras al momento de tratar de mejorar esos diagramas en etapas futuras. Los diagramas demasiado

[47] Enterpraise Architect 3.51 de Sparx Systems. Herramienta CASE para modelado con UML.

detallados no son tan útiles en las etapas de análisis y diseño pero puede ser de utilidad en las etapas finales, pero en estas etapas no se esta diseñando sobre el diagrama, lo que realmente se esta haciendo es actualizando el diagrama desde el código lo cual acerca el diagrama a la realidad y lo hace útil para futuras fases de desarrollo mientras que el exceso de detalle en etapas tempranas lo único que genera es retraso por que no se puede predecir por completo lo que pasara en al realidad, esto es al momento de generar el código que corresponde al diagrama. El caso contrario no es menos perjudicial para el proyecto, se pueden producir diagramas demasiado vagos (no confundir con la capacidad de ocultar información en los diagramas dependiendo del objetivo de estos) que se pueden convertir en un problema en el momento en que el proyecto empiece a crecer y sea imposible recordar exactamente lo que se hizo. La única solución en este caso es tratar de reconstruir lo que se hizo, ya sea desde código o generando la solución de nuevo si el problema esta en una de las fases iniciales.

Es importante aclarar que los cambios que se hicieron a la metodología para obtener el resultado que muestra la figura 30 no se realizaron todos en bloque y en el mismo intervalo de tiempo, esto fue un proceso que ocurrió a medida que se iba pasando por las fases y se necesitaba generar algún diagrama, se necesitaba construir un prototipo que no estaba previsto en la metodología original o se tenía alguna duda sobre la filosofía de objetos. El objetivo de este proyecto no era crear ninguna nueva metodología y tal ves hubiera sido mejor utilizar el proceso unificado desde el principio pero en su momento no parecía necesario y no se tenía el suficiente conocimiento de este como para adoptarlo como metodología, sin embargo el resultado final fue bastante satisfactorio y aunque se invirtió tiempo adicional al hacer este tipo de cambios en la metodología, este tiempo adicional fue bastante valioso porque se ganó conocimiento y experiencia en lo que tiene que ver con metodologías de desarrollo de software.

4.2 DESARROLLO DE LOS TRES PROTOTIPOS

4.2.1 Primer prototipo.

4.2.1.1 Fase de elaboración Primer Prototipo. En esta fase la principal preocupación es explorar el problema en detalle, entendiendo los requerimientos y las necesidades del cliente y del negocio

para poder desarrollar un plan ulterior. Es necesario obtener una visión amplia del sistema que se quiere realizar y entender aspectos generales de este. Esto es lo que Kruchten48 llama "a mille wide and inch deep view", una vista de una milla de ancho y una pulgada de profundidad. En esta etapa se debe evitar entrar en detalles de implementación.

Otro aspecto importante de esta fase es el de la elaboración de prototipos de áreas identificadas como difíciles o problemáticas dentro del proyecto. Los prototipos realizados en esta fase de la metodología puede ser reutilizados o simplemente desechados después de que hallan cumplido con su objetivo, que es el de dar solución a situaciones muy especificas del problema. Para este proyecto en esta fase se desarrollo un prototipo que se reutilizó en el primer prototipo y también en el sistema final como se verá mas adelante.

Los requerimientos para este primer prototipo se pueden resumir de la siguiente manera:

- Construir un prototipo de un sistema de entretenimiento en línea, capaz de recrear una partida del juego de estrategia BattleTech.

Desde luego de este requerimiento se desprenden una seria de requerimientos adicionales que tienen que ver con el cumplimiento de las reglas que contiene el manual de reglas de la caja de BattleTech de la cuarta edición. Una introducción a las reglas del juego se presenta en el apéndice A, introducción a las reglas de BattleTech. Estas reglas que se presentan son introductorias ya que las reglas completas ocupan aproximadamente 47 páginas, y puede que no sean muy claras si no se han entendido bien las reglas de introducción.

Un aspecto importante que hay que destacar es que para este caso los requerimientos que tienen que ver con las reglas y el desarrollo lógico del juego ya están levantados por completo dentro del libro de reglas, el trabajo en esta etapa esta en interpretarlas y modelarlas de acuerdo al resultado que se quiere obtener, ya que existen diversas soluciones posibles para modelar este problema.

Pero además este prototipo esta limitado por las características que se dieron al primer Prototipo que eran:

[48] The Rational Unified Process An Introduction. Second Edition. Krutchten Philippe. Addison Wesley. 2000.

- Será capaz de recrear una partida de BattleTech utilizando el primer nivel de reglas, contenido en el manual de reglas (Rulebook) de la caja de BattleTech cuarta edición (BattleTech box set fourth edition); en este prototipo cada robot, así como el escenario serán representados por mapas de bits (bitmaps).

Las limitaciones que se plantean para el primer prototipo se podrían resumir como limitaciones de funcionalidad del sistema y de interfaz gráfica. Estos requerimientos adicionales son:

- Es una aplicación que corre en una sola maquina
- Robots y Mapa serán mapas de bits. Más específicamente sprites que son mapas de bits con partes transparentes.

Para comprender mejor como debe comportarse el sistema y poder capturar sus requerimientos en su primera versión se realizaron algunos diagramas de casos de uso. El diagrama mostrado en la figura 31 es una de las primeras aproximaciones.

La figura 31 muestra el primer acercamiento al comportamiento y los requerimientos del sistema, este diagrama de casos de uso permite ver lo que debería hacer el sistema para poder recrear un apartida de BattleTech incluyendo la configuración inicial. Desde luego esta es una visión demasiado general. El siguiente paso es analizar mas en detalle lo que sucede cuando el sistema esta en cada uno de estos casos de uso.

Para este diagrama de casos de uso del sistema no se va ha entrar en detalle y no se va a mostrar ningún escenario, sin embargo para el siguiente que es el diagrama de casos de uso de análisis de jugar partida si se mostrara con un escenario.

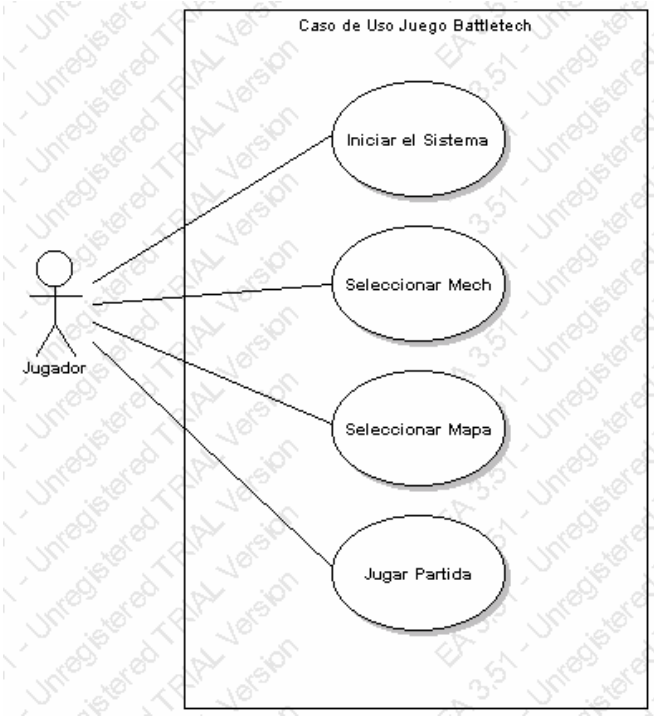


Figura 31. Diagrama de casos de uso.

La figura 32 muestra el diagrama de casos de uso que muestra lo que sucede cuando el sistema esta en jugar partida, esto es lo que algunos autores llaman “explotar” un caso de uso. Cada caso de uso de un diagrama puede ser muy complicado (puede representar otro sistema o un subsistema) así que puede generar un nuevo diagrama de casos de uso que muestra su comportamiento, que es lo que sucede en este caso.

Como el diagrama de casos de uso no es suficiente para explicar un diagrama de casos de uso aquí tenemos un escenario que nos muestra como es que este diagrama de casos de uso se comporta y nos ayuda a comprender la lógica del sistema y los requerimientos de este.

La documentación de este diagrama se hizo con Entreprise Architect (EA).

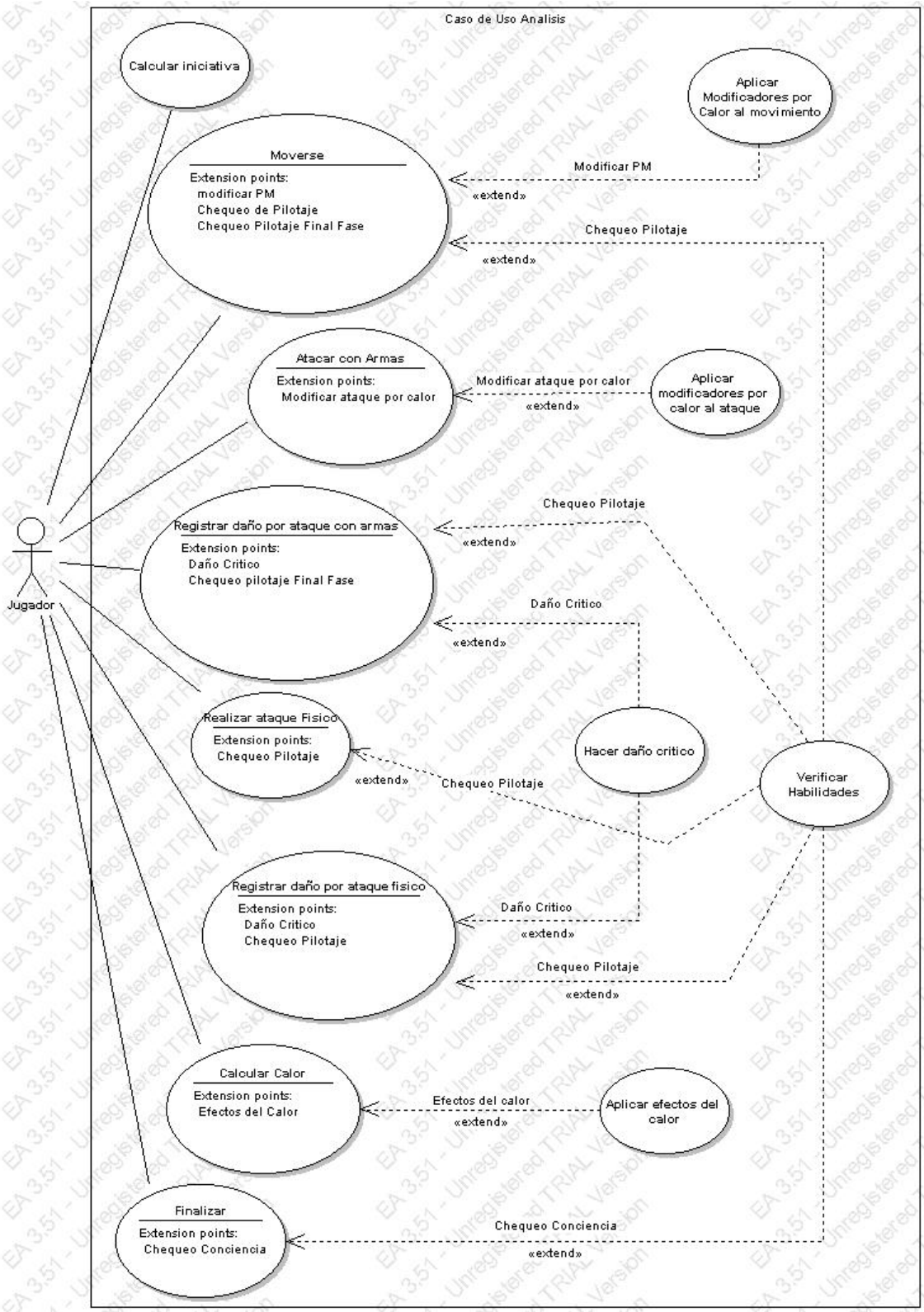


Figura 32. Casos de uso de análisis.

A continuación se muestra la documentación generada por EA que corresponde al diagrama de casos de uso de la figura 32.

Jugador

Type: *public* **Actor**
Status: Proposed. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 05/03/03. Author: Moldorf

Connections

- Association link to usecase *Calcular iniciativa*
- Association link to usecase *Realizar ataque Fisico*
- Association link to usecase *Moverse*
- Association link to usecase *Atacar con Armas*
- Association link to usecase *Registrar daño por ataque con armas*
- Association link to usecase *Registrar daño por ataque fisico*
- Association link to usecase *Calcular Calor*
- Association link to usecase *Finalizar*

Aplicar efectos del calor

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 12/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . Calor > 14.

Connections

- Extend link to usecase *Calcular Calor*

Scenarios

Basico {Basic Path}.
1. Si calor > 14 calculo de lanzamiento para evitar desconexion
2. Si lanzamiento falla aplica efectos de desconexion
3. Si calor > 19 calculo de lanzamiento para evitar explosion de municion
4. Si lanzamiento falla aplica efectos de explosion de municion

Aplicar modificadores por calor al ataque

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 11/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . calor > 8.

Connections

- Extend link to usecase *Atacar con Armas*

Scenarios

basico {Basic Path}.
1. si 8 =< calor < 13 +1 a la tirada de impacto
2. si 13 =< calor < 17 +2 a la tirada de impacto

3. si 17 =< calor < 24 +3 a la tirada de impacto
4. si 24 =< calor +4 a la tirada de impacto

Aplicar Modificadores por Calor al movimiento

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 11/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . Calor > 5.

Connections

- Extend link to usecase *Moverse*

Scenarios

basico {Basic Path}.
1. si 5 =< calor < 10 pm = pm - 1
2. si 10 =< calor < 15 pm = pm - 2
3. si 15 =< calor < 20 pm = pm - 3
4. si 20 =< calor < 25 pm = pm - 4
5. si 25 =< calor pm = pm - 5

Atacar con Armas

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 11/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . Debe haber finalizado la fase de movimiento.

Connections

- Extend link from usecase *Aplicar modificadores por calor al ataque*
- Association link from actor *Jugador*

Scenarios

basico {Basic Path}.
1. Escoge el mech con el que desea atacar
2. Escoge el mech que desea atacar
3. Escoge las armas que desea utilizar
4. Calcula la tirada que necesita para impactar
5. (modificar ataque por calor)
6. Lanza para saber si impacta
7. Si impacta y es un arma de misiles lanza para número de misiles
8. Si impacta lanza para localizar daño
9. Si hace critico lanza para dañar componente interno.
10. Si daña componente interno, lanza para localizar el componente y dañarlo
11. Calcula el calor por armas

Calcular Calor

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.

Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 12/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . Debe haber finalizado fase de ataque físico.

Connections

- Extend link from usecase *Aplicar efectos del calor*
- Association link from actor *Jugador*

Scenarios

Basico {Basic Path}.
1. Suma calor por movimiento + calor por armas y le resta el número de radiadores disponibles
2. (Efectos del calor)

Calcular iniciativa

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 08/04/04. Author: Moldorf

Calcula la iniciativa y determina quien se debe mover primero.

Connections

- Association link from actor *Jugador*

Scenarios

Basico {Basic Path}.
1. Lanzar 2D6
2. Se compara con el lanzamiento del otro jugador
3. Menor mueve primero

Calcular Modificadores Movimiento

Type: *public* **Use Case**
Status: Proposed. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 11/04/04. Modified on 11/04/04.

Connections

- Include link from usecase *Moverse*

Finalizar

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 12/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . Debe haber finalizado el calculo del calor.

Connections

- Extend link from usecase *Verificar Habilidades*
- Association link from actor *Jugador*

Scenarios

Basico {Basic Path}.

1.(Chequeo de conciencia)

Hacer daño critico

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 11/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . Si se aplica daño en estructura interna.

Connections

- Extend link to usecase *Registrar daño por ataque con armas*
- Extend link to usecase *Registrar daño por ataque fisico*

Scenarios

Basico {Basic Path}.

1. Aplica efectos por daño en componente interno

Moverse

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 11/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . Debe corresponder al orden de iniciativa.

Connections

- Extend link from usecase *Aplicar Modificadores por Calor al movimiento*
- Extend link from usecase *Verificar Habilidades*
- Association link from actor *Jugador*
- Include link to usecase *Calcular Modificadores Movimiento*

Scenarios

moverse {Basic Path}.

1. Escoge el mech que desea mover
2. Escoge el tipo de movimiento. (quieto, caminar, correr, saltar)
3. (Modificar PM)
4. Gasta los pm dependiendo del tipo de movimiento.
5. (Chequeo de pilotaje)
6. Calcula el modificador ganado por número de hexágonos movidos
7. (Chequeo de Pilotaje)

Realizar ataque Fisico

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 12/04/04. Author: Moldorf

Constraints

- *Mandatory Pre-condition* . Debe haber finalizado la fase de ataque con armas.

Connections

- Association link from actor *Jugador*
- Extend link from usecase *Verificar Habilidades*

Scenarios

Basico {Basic Path}.

1. Escoge el mech con el que desea atacar
2. Escoge el mech que desea atacar
3. Escoge el tipo de ataque fisico
4. Calcula la tirada que necesita para impactar
5. Lanza para saber si impacta
6. (Chequeo de pilotaje)
8. Si impacta lanza para localizar daño
9. Si hace critico lanza para dañar componente interno.
10. Si daña componente interno, lanza para localizar el componete y dañarlo

Registrar daño por ataque con armas

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 12/04/04. Author: Moldorf

Internal Requirements

- *Approved* . Registrar el daño recibido al mech y aplicar los efectos que este daño genere. (*Difficulty:* Medium; *Priority:* High)

Connections

- Extend link from usecase *Verificar Habilidades*
- Extend link from usecase *Hacer daño critico*
- Association link from actor *Jugador*

Scenarios

Basico {Basic Path}.

1. Registra el daño recibido
2. (Daño Critico)
3. (Chequeo de pilotaje) Al final de la fase y si daño recibido > 20

Registrar daño por ataque fisico

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 12/04/04. Author: Moldorf

Connections

- Extend link from usecase *Hacer daño critico*
- Extend link from usecase *Verificar Habilidades*
- Association link from actor *Jugador*

Scenarios

Basico {Basic Path}.

1. Registra el daño recibido
2. (Daño Critico)
3. (Chequeo de pilotaje) Al final de la fase y si fue pateado

Verificar Habilidades

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Details: Created on 05/03/03. Modified on 12/04/04. Author: Moldorf

Internal Requirements

- *Approved* . Verificacion de habilidades. Realiza chequeos de pilotaje y conciencia. (*Difficulty:* Medium; *Priority:* High)

Constraints

- *Mandatory Pre-condition* . Se mueve y entra a hexagono de agua.
- *Mandatory Pre-condition* . Daño recibido > 20.
- *Mandatory Pre-condition* . Fallo patada.
- *Mandatory Pre-condition* . Fue pateado.

Connections

- Extend link to usecase *Moverse*
- Extend link to usecase *Registrar daño por ataque con armas*
- Extend link to usecase *Registrar daño por ataque fisico*
- Extend link to usecase *Finalizar*
- Extend link to usecase *Realizar ataque Fisico*

Scenarios

Chequeo Conciencia {Alternate Path}.
1. Chqeueo de conciencia
2. Si fallo chequeo entonces Aplica efectos por fallo de chequeo

Chequeo Pilotaje {Alternate Path}.
1. Aplica modificadores al pilotaje
2. Realiza chequeo de pilotaje
3. Si fallo chequeo entonces Aplica efectos por fallo de chequeo

El diagrama de casos de uso anterior es lo que se llama el modelo de casos de uso y muestra como el sistema se ve desde afuera. La suma de estos casos de uso es el sistema que queremos realizar y se asume que no hay huecos, es decir que todo lo que no se encuentre dentro de este diagrama no pertenece al sistema que se esta elaborando.

El siguiente paso después de crear el modelo de casos de uso es el de crear un modelo conceptual del sistema, este modelo describirá los principales conceptos del sistema y como estos están relacionados.

Para crear este modelo conceptual se utiliza el diagrama de clases. Este modelo conceptual no tiene que ser el modelo definitivo de clases que se implementara posteriormente. Lo importante de este modelo es capturar los conceptos o ideas importantes para el sistema.

Hay varias formas de encontrar los conceptos que son de importancia para el sistema, por ejemplo un posible concepto podría ser un objeto tangible, un lugar, un rol etc. No se entrara en detalle sobre la forma de capturar estos conceptos porque hay varios autores que describen como hacerlo (Booch[49] por ejemplo). Lo importante para destacar aquí es que se quiso que este diagrama fuera lo mas parecido a la realidad del juego de mesa, así que uno de los lineamientos que se tuvo en cuenta fue el de tratar de mantener la mayor cantidad de objetos tangibles y roles como clases.

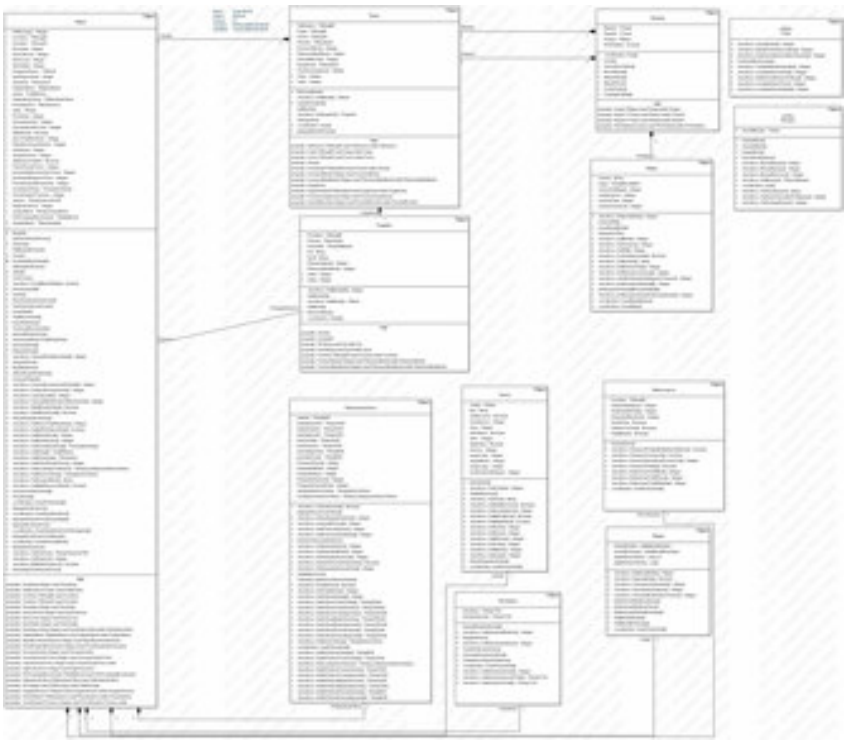


Figura 33. Modelo conceptual.

El resultado obtenido de este análisis es el que se muestra el la figura 33.

La figura 33 muestra parte del modelo conceptual que se obtuvo en esta fase. Como se ve en el modelo se trato de mantener una gran cantidad de objetos tangibles del juego de mesa como clases, por ejemplo mech que en el juego de mesa existe físicamente y esta representado por una hoja y su respectiva ficha o como en el caso de mapa que representa la retícula hexagonal de 25 x 17 pulgadas que se utiliza en el juego de mesa. Además es importante notar que en este modelo no se

[49] Análisis Y Diseño Orientado A Objetos Con Aplicaciones. Grady Booch. Addison-Wesley/Diaz de Santos. Willimntong, Delaware. E.U.A. 1996.

pretendía capturar todos los atributos y métodos de cada clase, ni siquiera se pretendía capturar la totalidad de clases que tendría el sistema ya que esta seria una tarea imposible.

El modelo completo consta de 10 clases con algunos métodos y atributos que se detectaron en esta etapa inicial de desarrollo.

Como se dijo anteriormente en esta etapa también se pueden realizar algunos prototipos de aspectos que se consideran problemáticos. Para este caso se realizó un prototipo que se incorporo dentro del sistema final como métodos de la clase Tmapa.

En esta fase de elaboración se detectaron dos problemas difíciles de resolver que eran parte fundamental del motor de reglas del juego. Los problemas eran los siguientes:

- El primer problema tenía que ver con los cálculos de distancia que se realizan dentro del juego. En la figura 34 se muestran dos mechs que están a 7 hexágonos de distancia. Se pueden contar los hexágonos que tienen puntos rojos (desde el mech azul hasta el amarillo) y además se incluye el hexágono ocupado por el mech amarillo. Esta distancia esta medida en hexágonos de distancia.
- El segundo problema tiene que ver con la línea de visión (LOS por sus siglas en ingles Line Of Sight). Se debe determinar por que hexágonos (coordenadas) pasa la línea roja que va del mech azul al amarillo. Para el ejemplo de la figura la línea roja pasa por los hexágonos 0810, 0809, 0909, 0808, 0908, 0807, 0907 y 0906. Este debe ser el resultado que se debe obtener cuando se desea calcular la LOS.

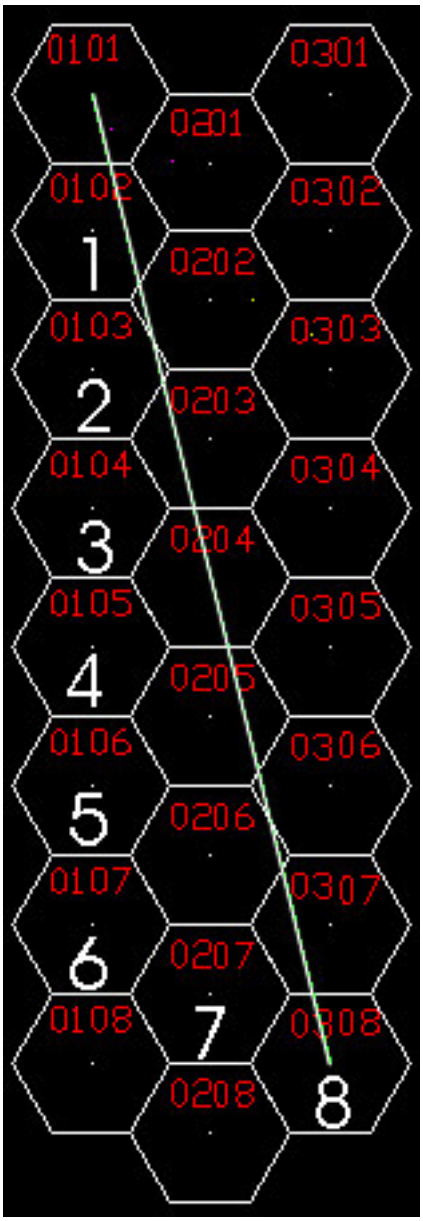


Figura 34. LOS.

El desarrollo de este prototipo tardo 4 semanas,pque es un tiempo bastante largo pero que de no detectarse en esta etapa hubiera demorado el desarrollo del proyecto. Para el desarrollo de este prototipo se tuvieron que probar varias soluciones para llegar a la que se implemento que permite hacer estos cálculos solo tomando como parámetros las coordenadas de los hexágonos, esto permite que el motor de reglas sea independiente de la interfaz gráfica del sistema.

La figura 35 muestra una imagen del prototipo terminado en su versión 2.1 que ya incluye otros algoritmos agregados durante el desarrollo del segundo y tercer prototipo. Los que primero se desarrollaron en esta fase de elaboración fueron los algoritmos que corren al presionar el botón RunLOS y DistranciaEntre, (encerrados en azul claro) que en la gráfica 35 están dando solución al problema planteado en la gráfica 34.

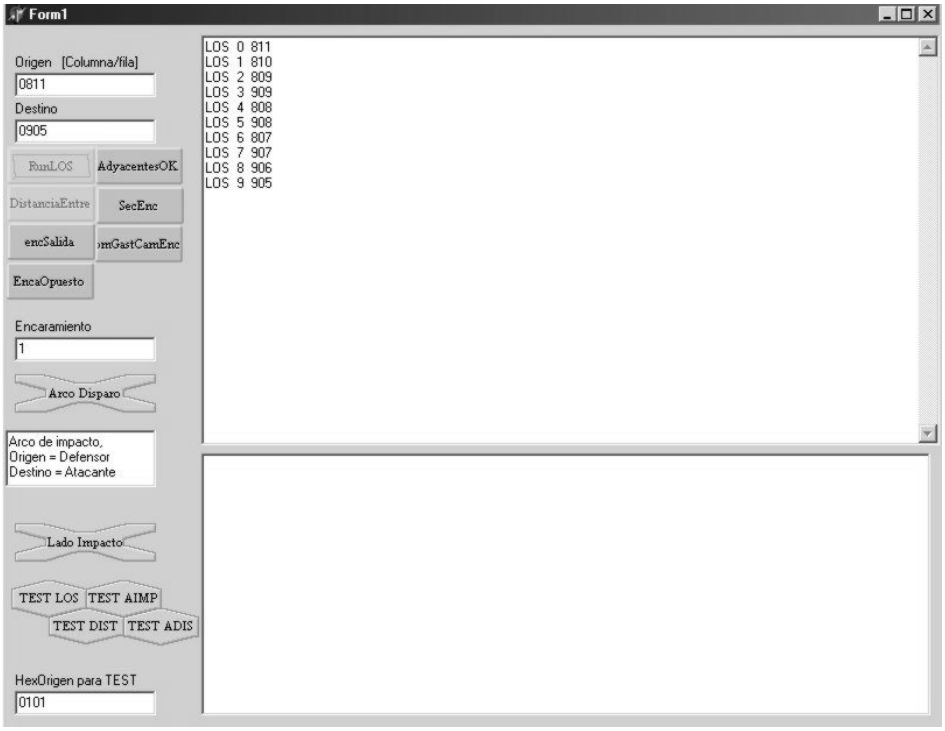


Figura 35. Prototipo LOS, Distancia entre.

Con estos dos modelos ya realizados y la elaboración del prototipo para LOS y distancia entre, damos por finalizada la fase de elaboración. Lo que sigue es la fase de construcción.

4.2.1.2 Fase De Construcción Primer Prototipo. En esta fase el objetivo es construir el producto y obtener una versión que pueda ser liberada. La estrategia para obtener esta versión del producto es seguir una serie de pequeñas cascadas, como la que muestra la figura 36, con un pequeño número de casos de uso desarrollados en cada iteración. El número de veces que se pasara por la cascada dependerá del número de casos de uso que se desee codificar en cada iteración o simplemente obedecerá a una decisión del equipo de desarrollo. Un punto importante al escoger el número de casos que se codificaran en cada iteración es que el objetivo de estas iteraciones es disminuir la

complejidad centrándose en pocos casos de uso en cada iteración, de otra forma no tendría ningún sentido estas iteraciones y se estaría utilizando simplemente una metodología de cascada.

Para este caso se paso por la cascada 5 veces como se explicara mas adelante.

En un caso ideal se obtendrá un producto corriendo al final de cada iteración, aunque desde luego limitado.

Cada fase de la cascada producirá un grupo de diagramas UML

- En análisis se producirán diagramas de casos de uso expandidos o completos
- En diseño se producirán diagramas de clases, interacción y de estados
- En la codificación se producirán unidades de código corriendo.

Al final de cada iteración se harán pruebas y se actualizaran algunos diagramas desde el código, principalmente el diagrama de clases.

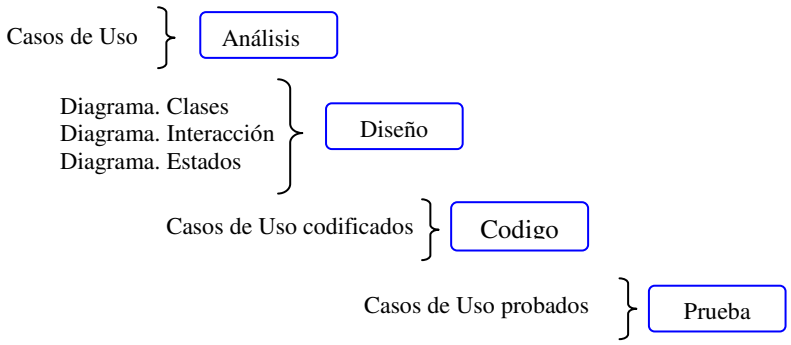


Figura 36. Cascada de construcción.

4.2.1.2.1 Análisis. En esta primera fase se revisan los casos de uso que se están construyendo en la iteración, se expanden y completan, de esta forma se reduce la complejidad del sistema.

Como se dijo anteriormente se escogió pasar por la cascada 5 veces, cada una de estas iteraciones se relaciona con un número determinado de casos de uso que se querían atacar. La figura 37 muestra el nuevo diagrama de casos de uso que se obtuvo después de las 5 iteraciones y que corresponde al primer prototipo. Se puede ver que se hicieron algunos cambios en las relaciones entre los casos de uso, estos cambios obedecen a la forma como el usuario interactúa con el sistema. En este caso el sistema puede hacerse cargo de algunas funciones que en el momento de modelar el juego de mesa tenían que ser realizadas por el jugador, por ejemplo, el jugador en el juego de mesa tiene que hacerse cargo de registrar el daño que recibió en la hoja del mech que pilota, en el caso del sistema éste se puede hacer cargo de esta acción y registrar el daño recibido. Por esta razón ahora Registrar Daño por Ataque con Armas se incluye (include) dentro de Atacar con Armas, desde luego la precondición para que se ejecute es que el mech haya recibido daño en ese turno.

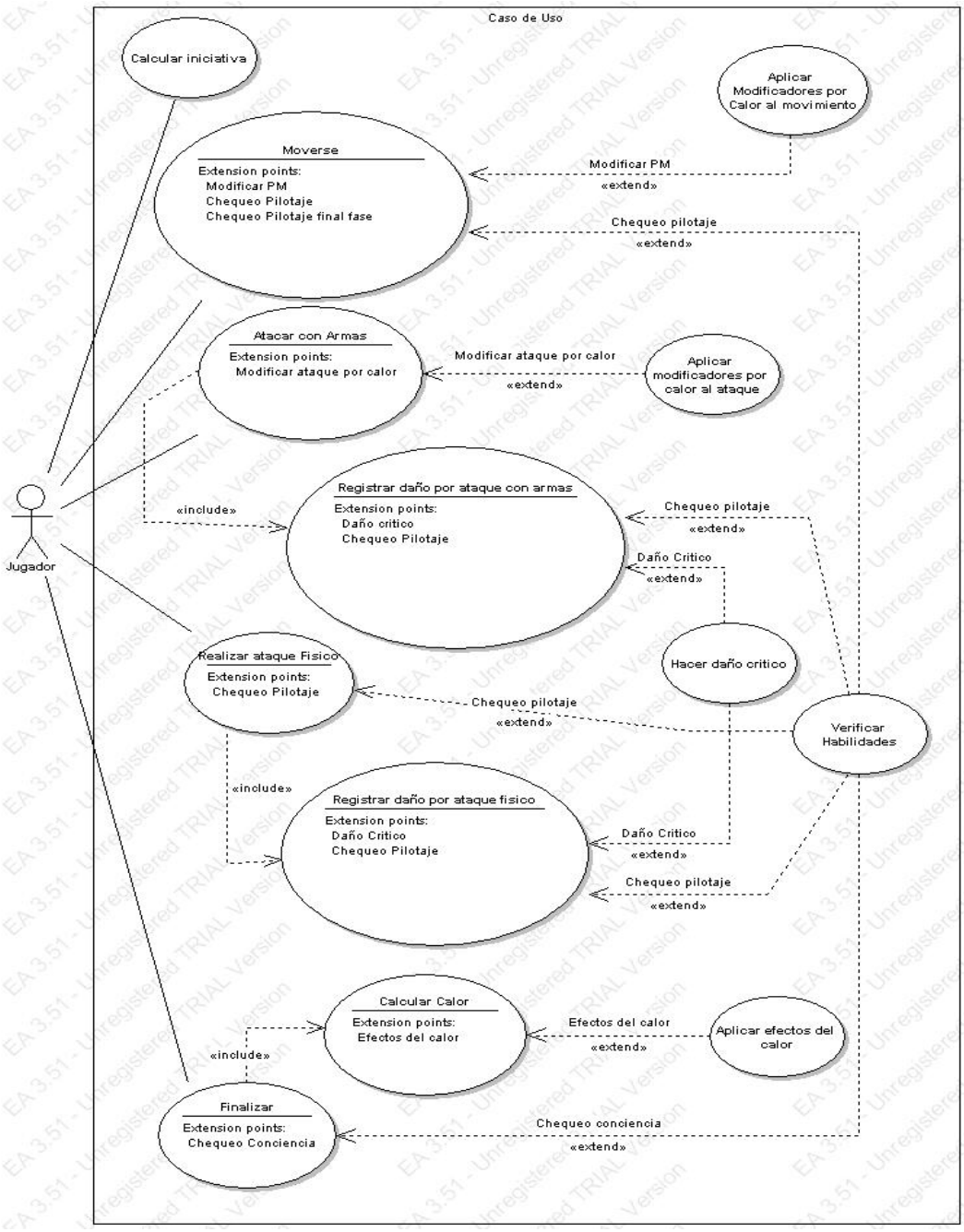


Figura 37. Diagrama de Casos de Uso**.**

La primera iteración que se realizó tenía como objetivo llegar a una aplicación que fuera capas de calcular iniciativa y moverse. Para esta primera iteración no se incluyo el caso de uso que se relaciona con moverse, cuyo objetivo es aplicar los modificadores por movimiento al calor. Esta

exclusión aplica para otros casos de uso que tienen que ver con modificadores y con verificaciones de habilidades que se dejaron para una ultima iteración, la número 5.

La segunda iteración tuvo como objetivo los casos de uso Atacar con Armas y registrar daño por ataque con armas. La relación que existe entre estos dos casos de uso cambio de la fase de elaboración ya que el registro del daño recibido es una tarea que puede realizar el sistema. De nuevo se dejo por fuera el caso de uso que tiene que ver con los modificadores por calor ya que depende de otro caso de uso que todavía no esta implementado, Calcular Calor. El otro caso de uso que se incluyo en esta iteración fue el caso de uso Hacer Daño Critico.

En la tercera iteración se codificaron los casos de uso Realizar Ataque Físico y Registrar Daño por Ataque Físico. De nuevo en este caso la relación existente entre estos dos casos de uso es una inclusión y no una relación directa de Registro Daño por Ataque físico con el jugador, como se mostraba en el diagrama de casos de uso de la fase de elaboración, ya que es una tarea que puede ser realizada por el sistema que agiliza el desarrollo del juego y libera al jugador de esta tarea.

La cuarta iteración introdujo la funcionalidad de los casos de uso Finalizar, Calcular Calor y Aplicar efectos del Calor. De nuevo aparece un cambio en las relaciones entre Calcular Calor y el actor que ahora no actúa directamente sobre este caso de uso. Este cambio obedece a la funcionalidad del sistema y a la decisión de liberar al jugador de tareas tediosas que retrasan el desarrollo del juego y que pueden ser realizadas por el sistema sin afectar el ambiente de juego.

Es importante notar que no solo se realizaron cambios en las relaciones y la estructura del juego, también se enriquecieron los casos de uso al entrar mas en detalle en cuanto a sus escenarios. Por ejemplo podemos compara el caso de uso Moverse de la fase de Elaboración con el producido en esta fase en la primera iteración.

Este es el caso de uso de la fase de elaboración:

Moverse

Type: *public* **Use Case**
Status: Approved. Version 1.0. Phase 1.0.
Package: Use Case Model Analysis
Constraints

- *Mandatory Pre-condition*. Debe corresponder al orden de iniciativa.

Connections

- Extend link from usecase *Aplicar Modificadores por Calor al movimiento*
- Extend link from usecase *Verificar Habilidades*
- Association link from actor *Jugador*
- Include link to usecase *Calcular Modificadores Movimiento*

Scenarios

Moverse {Basic Path}.

1. Escoge el mech que desea mover
2. Escoge el tipo de movimiento. (quieto, caminar, correr, saltar)
3. (Modificar PM)
4. Gasta los pm dependiendo del tipo de movimiento.
5. (Chequeo de pilotaje)
6. Calcula el modificador ganado por número de hexágonos movidos
7. (Chequeo de Pilotaje)

Este es el caso de uso de la primera iteración de la fase de análisis:

Moverse

Type: *public* **Use Case**
Status: Proposed. Version 1.0. Phase 1.0.
Package: Use Case model Design

Constraints

- *Mandatory Pre-condition* . Debe corresponder al orden de iniciativa.

Connections

- Association link from actor *Jugador*
- Extend link from usecase *Aplicar Modificadores por Calor al movimiento*
- Extend link from usecase *Verificar Habilidades*

Scenarios

moverse 2 { Basic Path}.

1. Escoge el mech que desea mover
2. Escoge el tipo de movimiento. (quieto,caminar, correr,saltar)
3. (Modificar PM) Ocurre si Calor > 5
3. Revisa que el hexagono por el que desea pasar no este ocupado por un mech enemigo
4. (Chequeo de pilotaje) Si el hexágono por el que pasa es agua
4. Gasta los PM dependiendo del tipo de movimiento.
5. Calcula el modificador ganado por número de hexágonos movidos
6. Revisa que el hexágono en el que termine el movimiento no este ocupado por otro mech.
7. (Chequeo de pilotaje) Si hay daño en cadera o gyro y corrio

Las principales diferencias tienen que ver con los cambios en la descripción del caso de uso que es un poco mas detallada. Además se agregaron las precondiciones de los puntos de extensión a la descripción para mostrar mejor el comportamiento del caso de uso.

Los cambios en los demás casos de uso varían mucho, en algunos se tuvo que hacer una descripción aun mas detallada para poder entender bien lo que sucedía dentro de cada uno de ellos y además se hicieron cambios en sus relaciones como ya se había hecho. No es necesario mostrar de nuevo las descripciones detalladas de todos los casos de uso.

En esta fase también se suelen realizar algunos diagramas de secuencia para lograr un mayor entendimiento de los casos de uso. En este caso no se consideró necesario.

Con estos diagramas de casos de uso detallados se da por terminada la fase de análisis.

4.2.1.2.2 Diseño. En este momento del desarrollo del sistema se tiene un completo entendimiento del problema que se quiere resolver, para la iteración en la que se encuentre (primera iteración, segunda,…, quinta iteración). Se han desarrollado los casos de uso parar la iteración con un nivel de profundidad y detalle mayor que permite empezar a diseñar una solución para el problema presentado.

Los casos de uso son satisfechos por la interacción de objetos, así que en esta fase del desarrollo del proyecto hay que decidir cuales objetos son necesarios para la iteración que corresponde, las responsabilidades de esos objetos y como estos objetos necesitan interactuar.

UML posee los diagramas de interacción, ya sea de colaboración o secuencias que permiten modelar las interacciones entre los objetos. Estos dos diagramas de interacción (colaboración y secuencias) son isomorficos lo que permite pasar de un diagrama a otro con relativa facilidad, la diferencia radica en el objetivo que persigue cada uno de ellos, uno se enfoca en el tiempo y el otro en las relaciones que existen entre los objetos.

En el momento en que se decide que objetos son los que interactúan dentro de los casos de uso que se están trabajando dentro de la iteración, el diagrama de clases de UML permite capturar esta información. Gran parte del trabajo de producir el diagrama de Clases ya esta hecho en el modelo conceptual que se había realizado en la fase de elaboración.

Otro de los modelos que es de gran utilidad en esta etapa de diseño es el diagrama de estados como se verá mas adelante.

En conclusión los diagramas que se van a producir en esta fase van a ser los diagramas de interacción, de clases y de estados.

Antes de mostrar el resultado final de las 5 iteraciones en el diagrama de clases y en algunos de los diagramas de iteración y de estados hay que tener en cuenta algunas consideraciones que tienen que ver con el diseño de juegos en general.

Al empezar el diseño del sistema lo primero que se tuvo en cuenta es que en general un juego tiene un comportamiento como el mostrado en la figura 38.

La figura 38 muestra la transición de Estados para el ciclo de un juego. Esta transición de estados para un juego se cumple más o menos de la misma forma para todos los juegos.

El primer estado es el estado de inicialización del juego, en este estado se hace todo lo tradicional en la inicialización de una aplicación, por ejemplo asignación de memoria, cargar información del disco, etc.

El siguiente estado es el del menú del juego, que es un menú típico que permite navegar por las posibles opciones que este permite (configuración de GUI, salvar partida, nivel de dificultad, etc.).

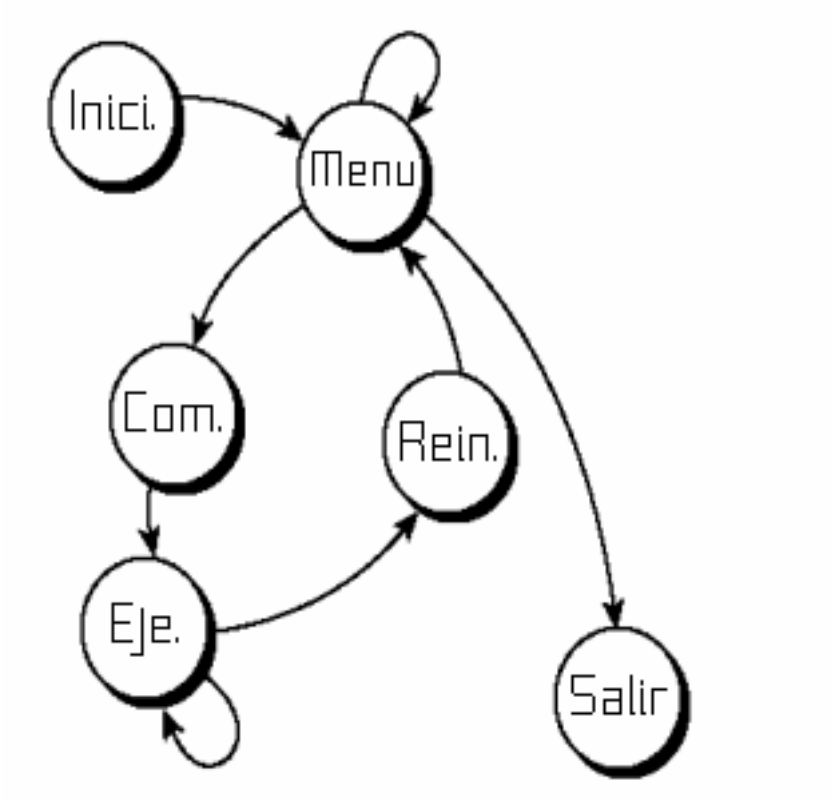


Figura 38. Transición de Estados para el ciclo de un juego.

El tercer estado del juego es el de Comenzando o arrancando el juego. En este estado se hacen tareas como cargar la IA de los enemigos, cargar el ambiente en el que se desarrolla el juego, cargar el avatar del jugador, etc.

El cuarto estado del juego es el ciclo del juego (game loop) y en este es en el que se desarrolla la acción del juego. En este estado debe capturarse de alguna forma los mensajes del sistema para poder mantener al jugador siempre dentro del desarrollo del juego. Se captura la entrada del teclado, de un joystick, etc., se generan las imágenes necesarias para recrear el ambiente etc. El juego debe permanecer en este ciclo hasta que el jugador pierda, gane o decida salir del juego para reiniciarlo por ejemplo o para dar por terminada la partida.

El otro estado del juego es reiniciando, en este estado se debe hacer todo lo necesario para poder comenzar a jugar de nuevo.

El sistema que se desarrollo no era la excepción a la regla y cumple con la misma transición de estados. Teniendo en cuenta esto se empezó a diseñar la solución que permite participar en una partida de BattleTech.

Uno de los aspectos en los que mas se invirtió tiempo fue en pensar como este sistema se iba a mantener dentro de ese ciclo del juego, ya que este es un juego a típico en el sentido en que los estado futuros del juego dependen de una interacción entre los jugadores que están en la partida, pero esta interacción incluye pequeñas demoras. Además de esto el jugador debe poder realizar ciertas labores mientras espera a que los demás jugadores realicen sus jugadas en el turno correspondiente. La solución a este problema esta dada por el comportamiento de la clase TPartida que es la encargada de controlar la secuencia de la partida pasando por varios estados. Esta clase crea ese ciclo en el que se encuentra el sistema mientras se desarrolle la partida.

El diagrama de clases final (después de la quinta iteración) que se obtuvo es el que muestra la figura 39.

A este diagrama se llego utilizando varias técnicas de diseño orientado a objetos presentadas en los libros de Grady Booch[50] y Meilir Page-Jones[51].

[50] Diseño Orientado A Objetos. Grady Booch. Traducción: Jaime O. Albarracin. Universidad Industrial de Santander. Bucaramanga. 1998.
[51] Fundamentals Of Object-Oriented Design In Uml. Meilir Page-Jones. Addison Wesley. New York 2000.

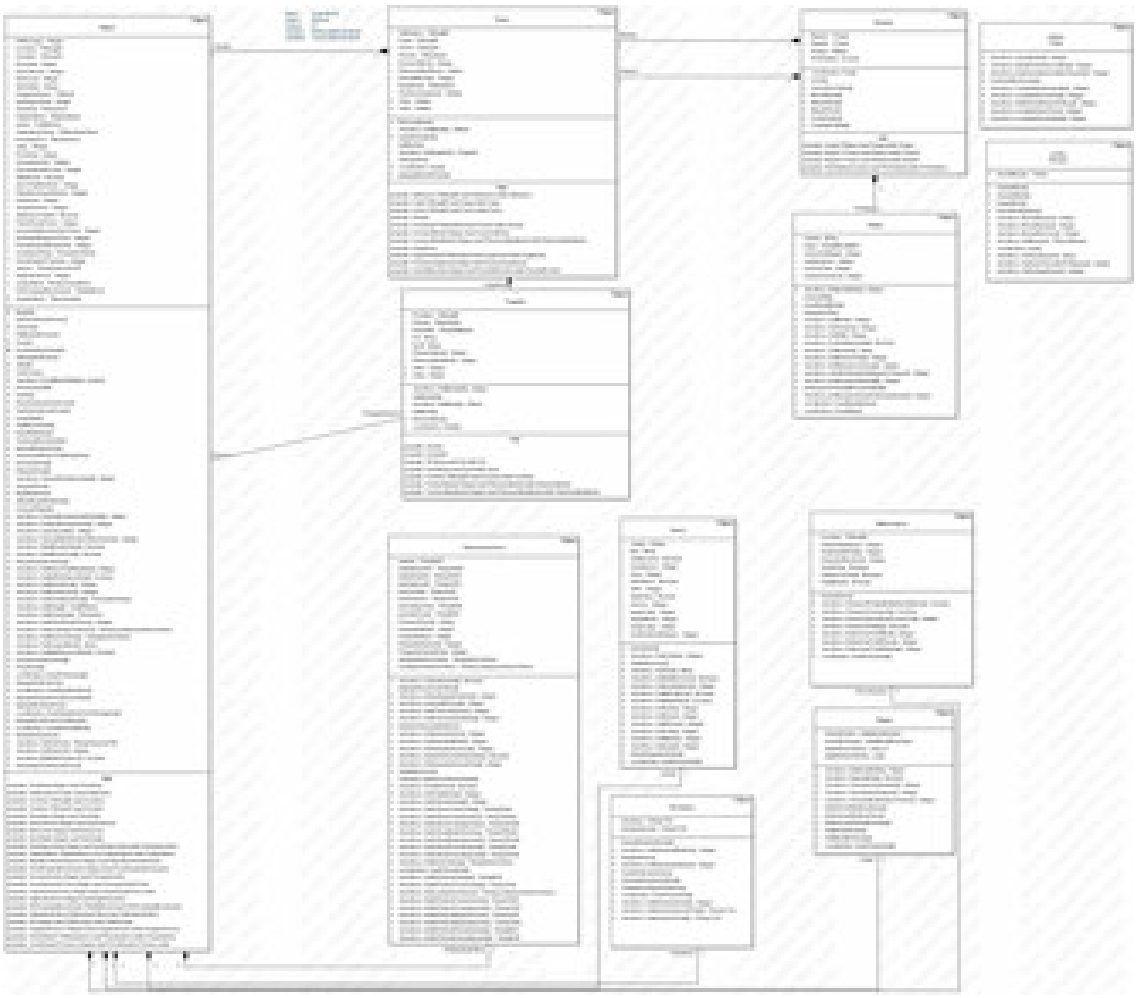


Figura 39. Diagrama de Clases primer prototipo.

Durante el desarrollo de las cinco iteraciones que produjeron el diagrama de clases de la figura 39, apareció un problema que tenía que ver con la interfaz gráfica de usuario. ¿Como se iba a mostrar la información de las clases? ¿Era necesario que cada clase tuviera métodos dedicados a mostrar la información de la clase en la Interfaz gráfica?

La segunda opción no es una solución, es una pésima idea porque estaría en contra de la filosofía de objetos, se estarían creando clases sin cohesión. La solución esta dada por la creación de dos capas lógicas dentro del juego, una de ellas seria la que representan las clases base de la figura 39 que constituyen el motor de reglas y la otra capa es la que tiene que ver con la interfaz gráfica. Una ventaja adicional de tener estas dos capas es la de poder cambiar la interfaz gráfica sin tener que hacer ningún cambio a las clases base, encapsulación de la información. Para el desarrollo del

proyecto este es un aspecto muy importante ya que la interfaz gráfica del primer prototipo es en 2D y la del segundo y tercero es en 3D.

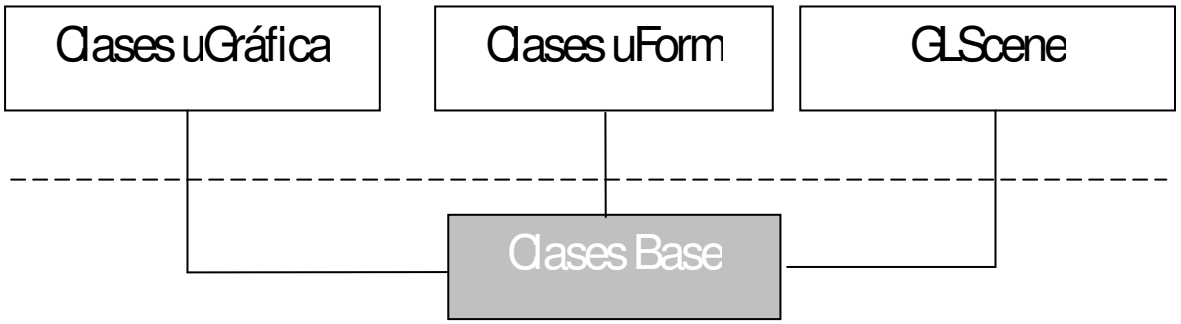


Figura 40. Capas lógicas del sistema.

Es importante aclarar algunos aspectos de la figura 40. Las clases de uGráfica, son clases especialmente diseñadas para mostrar información del sistema. La mayor parte de estas clases son descendientes de la clase TGraphic de Delphi, de estas clases se hablara mas adelante.

Las clases de uForm son las clases descendientes de TForm (esta es la clase formulario de Delphi) y de los clases de la VCL[52] de Delphi utilizados.

Las clases de OpenGL se refieren a las clases de GLScene[53] y que se usan para mostrar las imágenes en 3D.

Por último las clases base son las que se muestran en la figura 39.

Dentro del comportamiento de las clases base es importante destacar los cambios de estado por los que pasa TPartida que es la clase que controla el desarrollo del juego y que muestra el ciclo principal del juego. El diagrama de estados de esta clase se muestra en la figura 41.

[52] Visual Component Library. Librería de componentes Visuales.
[53] Librería para Delphi basada en OpenGL. http://glscene.org.

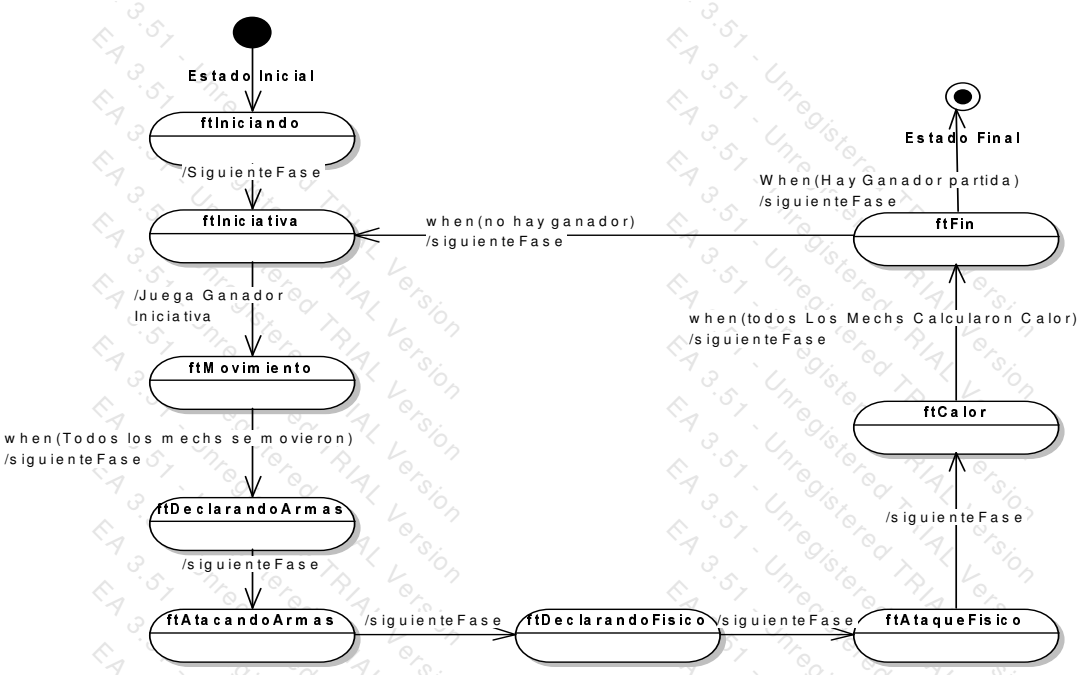


Figura 41. Estados de partida.

Un aspecto muy importante del diseño de la herramienta fue la utilización de Spider Containers And Persistent Classes[54] para permitir que las clases base del sistema fueran objetos persistentes, es decir se guardan en disco como objetos en un archivo binario, no hay ningún tipo de traducción a una base de datos entidad relación o similar. Otra de las ventajas de Spider Containers And Persistent Classes es que al ser una librería de clases para Delphi, ésta queda incorporada dentro del sistema al compilar, al quedar incorporada la librería dentro de la aplicación evita que se tengan que utilizar herramientas adicionales de bases de datos para guardar toda la información de las partidas.

Un aspecto que vale la pena mencionar en esta etapa fue el esfuerzo de diseño que se realizó para poder representar la información de una hoja de Mech como la que muestra la figura 42 en la interfaz gráfica del juego.

[54] Copyrighted © 1996,97,98 Michel Brazeau. Interval Software. http://www.cam.org/~mibra/spider

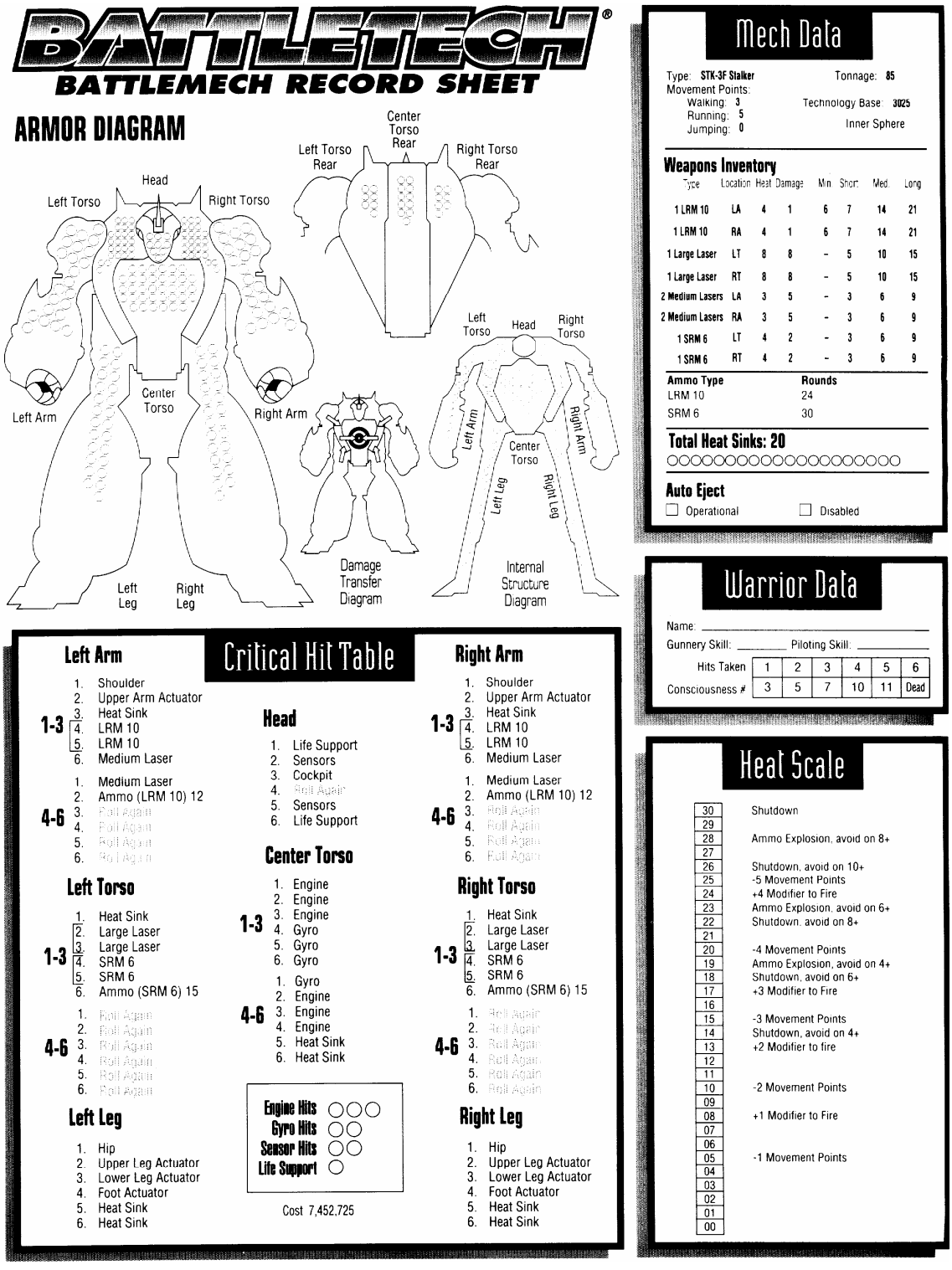


Figura 42. Hoja de Mech.

Además de mostrar la información de la hoja se debía mostrar el mapa y los dados para tener todo lo necesario para el desarrollo de la partida.

Para dar solución a este problema se diseñaron y se utilizaron componentes de delphi rediseñados. La figura 43 muestra como es el esquema de la armadura en la hoja de mech. La figura 44 muestra el componente grafico que representa la armadura dentro del sistema en varias de sus posibles configuraciones.

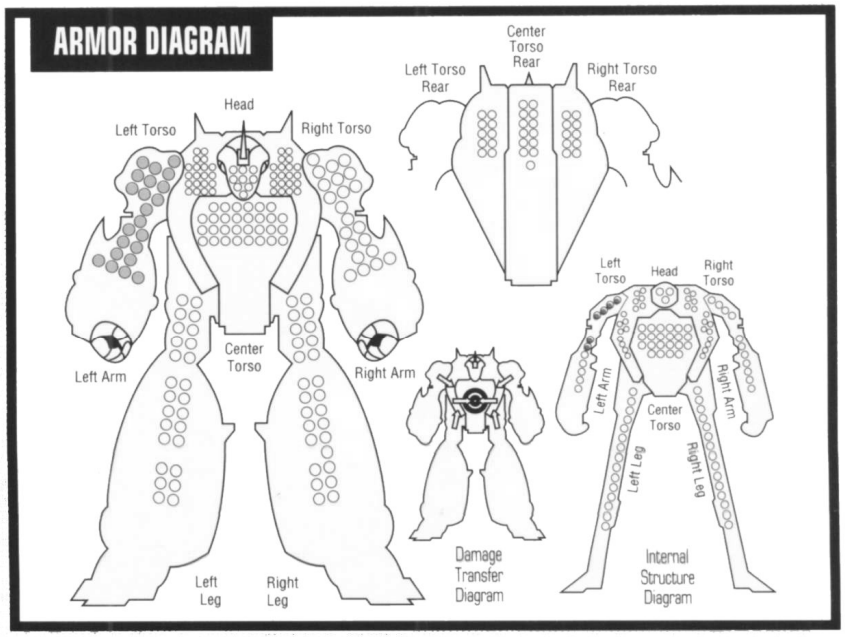


Figura 43. Armadura en hoja de mech.

Esta parte de diseño fue una de las que mas consumió tiempo y energía. Para el caso de los componentes que representan la armadura y la estructura interna cada uno de ellos consumió entre tres y cuatro semanas de trabajo entre diseño, diseño grafico y codificación para poder agregar la funcionalidad necesaria La figura 44 muestra el resultado del componente que representa la armadura y la figura 47 muestra el resultado del componente que representa la estructura critica. Este componente además de mostrar la estructura crítica acepta clic del ratón dependiendo de la fase del juego. Otros componentes que también se diseñaron especialmente para el sistema fueron los que representan el estado de calor del mech y se muestran en la figura 45.

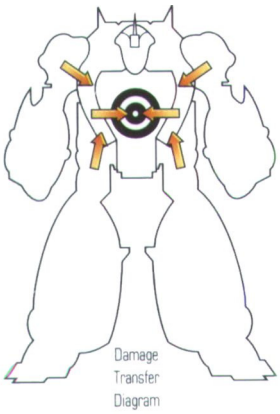


Figura 44. Componente Grafico de armadura.

La figura 44 muestra varios de las formas como se puede ver la armadura dentro del sistema. La primera de ellas (superior izquierda), es la forma inicial en la cual solo se muestra la armadura exterior. La segunda (superior derecha), muestra la armadura exterior del lado izquierdo y la interior del lado derecho, esta forma es utilizada en la configuración del juego al momento de escoger los mechs de la partida. La tercera que se muestra (inferior izquierda), es la que aparece dentro del juego. En esta opción el componente va cambiando a medida que pierde puntos de la armadura. En lugar de tener dos gráficos a parte para mostrar armadura interna y externa, se muestra la armadura interna en color naranja y solo en aquellas partes en las que la armadura exterior se perdió, como es el caso de la pierna y el brazo derecho de la figura. La armadura exterior cambia de color dependiendo de los puntos de armadura que se tengan, si esta roja quiere decir que tiene menos del 25 por ciento de los puntos de armadura totales para la parte del mech.

La figura 45 muestra los componentes para las barras de calor, en la gráfica aparece del lado derecho el diagrama original de la hoja de mech que permite controlar el calor dentro del juego. El lado derecha muestra los tres componentes diseñados para el sistema.

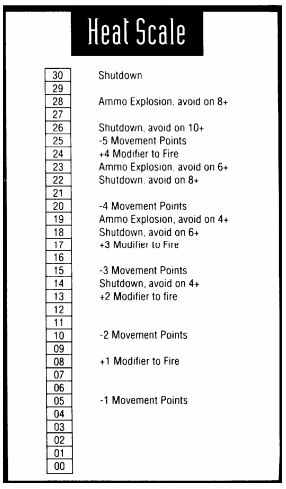


Figura 45. Barras de calor.

La barra de calor total es la que representa exactamente al la de hoja del mech, muestra el calor con el que termine el mech al final del turno. La barra explicación del calor reemplaza el texto que aparece del lado derecho de la escala de calor del diagrama de la hoja de mech. En lugar de describir el efecto del calor, como en la hoja, en el componente se usas iconos que se encienden y apagan dependiendo del efecto de calor.

El diseño y codificación de los componentes barras de calor consumió más o menos 2 semanas. En este caso se diseñó primero el componente de la barra de calor que tiene los números dentro (calor total), que es una modificación a un diseño de una barra creada por Dobrin Dobrev[55]. Basándose en esta se diseñaron las otras dos barras horizontal y vertical con pocos números (explicación calor, calor parcial).

La figura 46 muestra la tabla que muestra la estructura crítica del mech. Esta es una parte fundamental de la hoja del mech. Aquí se tiene el registro de las partes críticas del mech, por esta razón es importante que el jugador tenga esta información a la mano siempre.

[55]http://elektronika.vega.bg

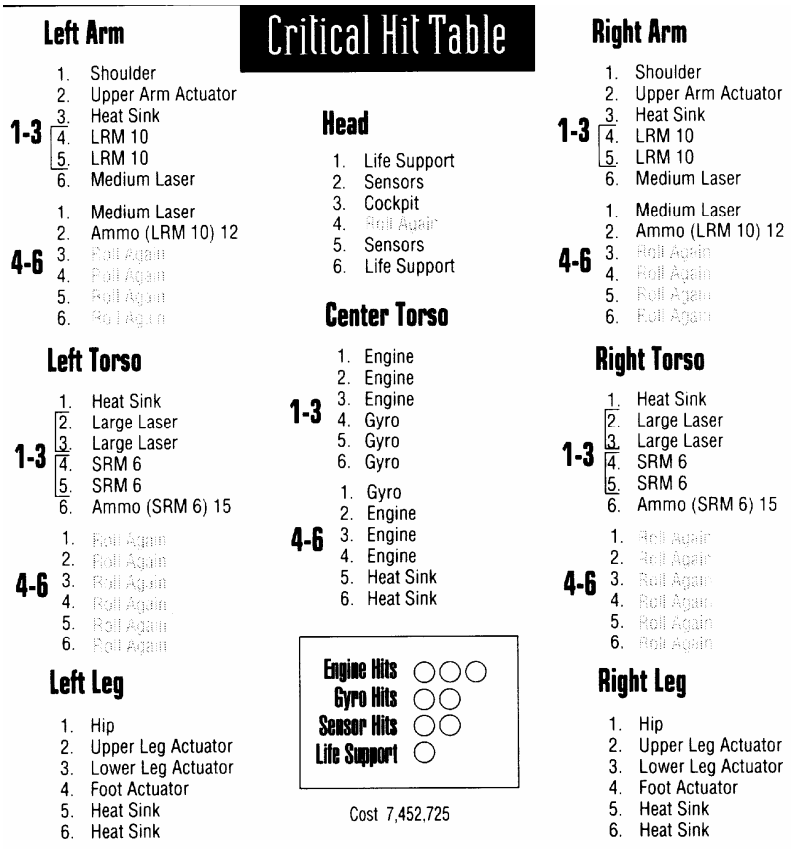


Figura 46. Tabla estructura crítica.

La figura 47 muestra el componente diseñado para el sistema que muestra la estructura crítica del mech. Este componente es muy diferente del que se muestra en la hoja de mech.

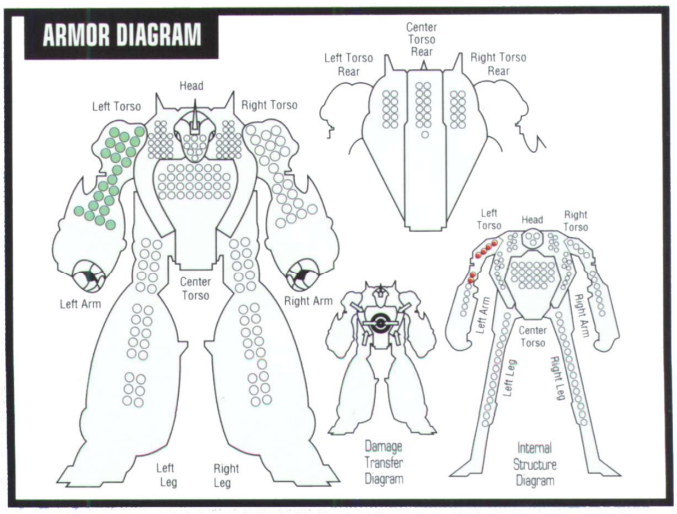
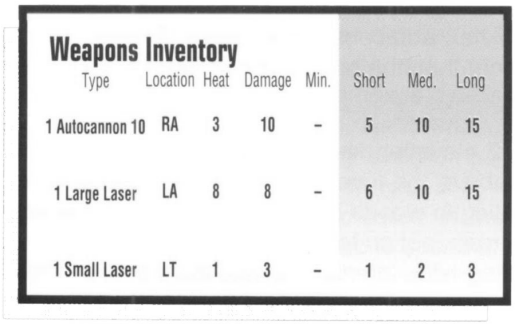


Figura 47. Componente estructura crítica.

Las razones por las que no se mantuvo la misma estructura de la hoja de mech son principalmente agregar claridad, permitir que el jugador lea mas rápidamente el diagrama y además se agrego funcionalidad que no existe en la hoja de mech. Esta funcionalidad es la de poder activar/desactivar las armas del mech haciendo clic sobre el arma dentro del diagrama. Esta función es de gran utilidad ya que dependiendo de la localización del arma dentro del mech se puede determinar a que objetivos se puede disparar teniendo en cuanta la localización del objetivo dentro del mapa. Como ayuda para esta funcionalidad el diagrama puede estar en varios estados, muestra todos los críticos, en forma de paginas de 6 ranuras cada uno, solo los críticos de la pagina 1 o 2, una configuración creada por el jugador (tiene memoria para recordar la configuración), una en la que solo muestra las partes que son únicas para el mech y que no pertenecen al chasis usado por todos los mechs y por último una en la que solo se muestran sus armas y la munición de estas si aplica.

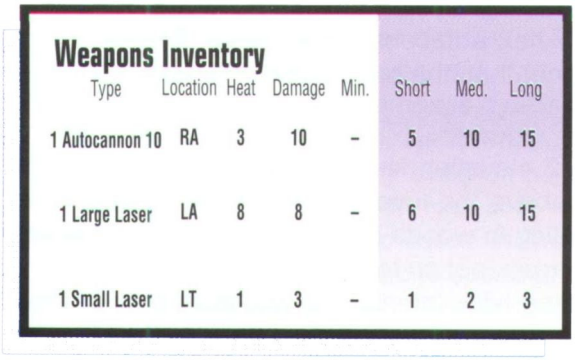
Por último se el componente que muestra el inventario de armas y la munición.



Weapons Inventory

Type	Location	Heat	Damage	Min.	Short	Med.	Long
1 Autocannon 10	RA	3	10	–	5	10	15
1 Large Laser	LA	8	8	–	6	10	15
1 Small Laser	LT	1	3	–	1	2	3

Figura 48. Inventario armas de la hoja de mech.



Weapons Inventory

Type	Location	Heat	Damage	Min.	Short	Med.	Long
1 Autocannon 10	RA	3	10	–	5	10	15
1 Large Laser	LA	8	8	–	6	10	15
1 Small Laser	LT	1	3	–	1	2	3

Figura 49. Componente para inventario armas, munición y ayudas.

La figura 49 muestra el componente que muestra el inventario de armas, este componente es una modificación de un TStringGrid de delphi diseñado e implementado por Andreas Hörstemeier[56]. Esta versión mejorada de la tabla permite entre otras cosas el uso de diferentes formatos de fuente, de color y alineación en cada casilla de la tabla, organización de las filas según alguno de los campos de las columnas y la captura de clicks del ratón en cada una de las celdas. Esta tabla del inventario de armas suprime la información acerca de la localización (esta presente en la estructura critica) y agrega información que no estaba originalmente en la hoja de mech y que tiene que ver con la cantidad de armas del mismo tipo que están activas. A diferencia de la tabla de la hoja, en el componente no aparece dos veces o mas el mismo nombre del arma, para eso esta la columna x en donde aparece el número de armas de ese tipo presentes. Esta información adicional se incorporo para mejorar la experiencia de juego. Los otros dos componentes que aparecen en la figura 40 son unos componentes de ayuda para movimiento (muestra modificador por movimiento total y por terreno y hexágonos), y para el ataque (muestra la distancia entre un hexágono y otro en hexágonos y los posibles lanzamientos para los rangos corto, medio y largo).

[56] http://www.hoerstemeier.com. andy@hoerstemeier.de.

Además de estos componentes para representación de la hoja de mech se hicieron dos prototipos en la primera iteración que también tomaron algo de tiempo, pero que se hicieron porque se consideró que eran de gran importancia para el desarrollo de la herramienta. El primero de ellos es un programa que permite la creación de un mech y la inclusión de este en la base de datos del juego.

La figura 50 muestra una imagen de esta aplicación. Esta aplicaron se realizó no solo para poblar la base de datos del juego sino también para poder validar la creación del objeto mech que es parte fundamental del sistema.

Este prototipo evoluciono durante el desarrollo del sistema y llego a la versión 6.0.

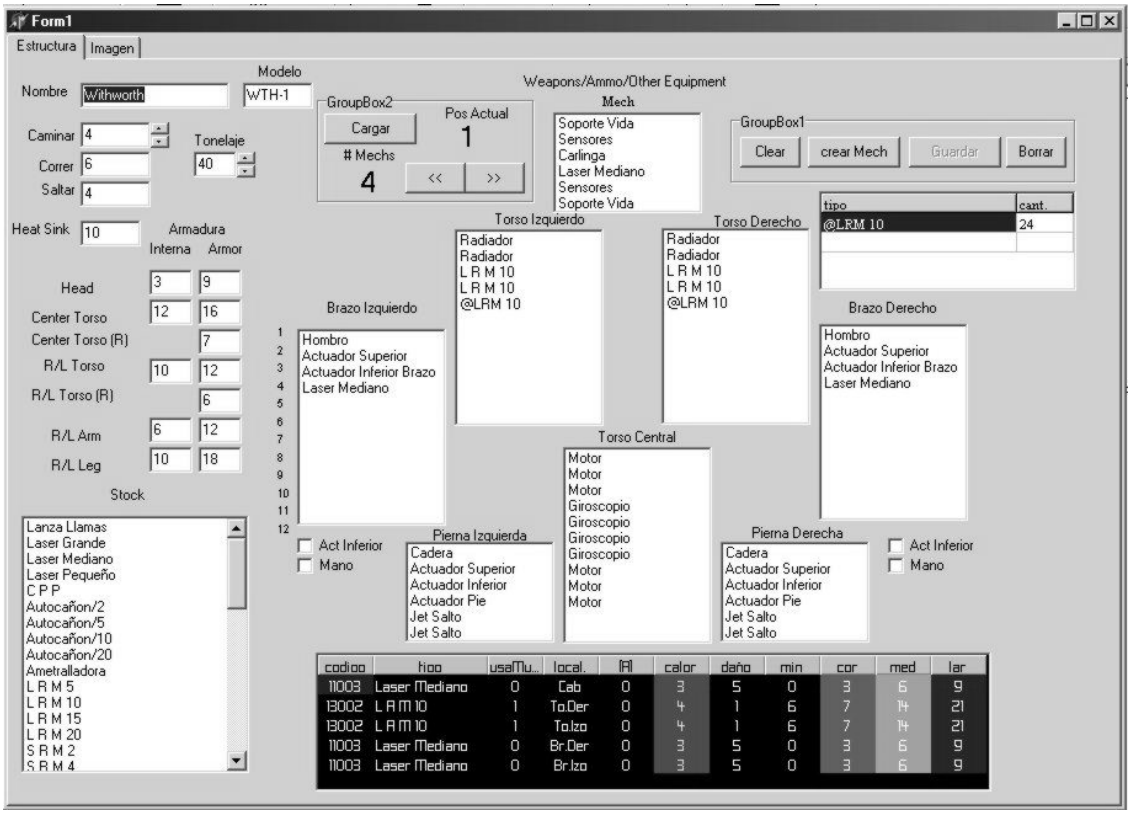


Figura 50. Creador de mechs.

El segundo prototipo desarrollado es el encargado de crear los mapas para el juego. Este prototipo nació para validar la creación de los mapas y para resolver el problema del empalme de los cuatro

que están disponibles, como lo muestra la figura 51, hasta convertirse en parte del sistema desde el segundo prototipo. Este prototipo llego a la versión 2.0 antes de volverse parte del sistema.

El problema se puede resumir en que los mapas que se unen (máximo cuatro) para formar el mapa final tienen 15 columnas por 17 filas y no es posible empalmarlos sin reducir alguno de los mapas o sin aumentar el número de filas o columnas después del empalme. La solución que se tomó fue la de aumentar el número de filas y columnas del mapa producido de la unión de varios mapas. Esta solución se tomó porque a diferencia del juego de mesa en el que lo mas fácil es solapar filas o columnas, en el sistema es posible agregar filas o columnas al mapa antes de generarlo. La limitante de cuatro mapas se agrego porque solo en ocasiones muy especiales se utilizan más de 4 mapas, la mayoría de las veces se usa entre 1 y 4 mapas solamente.

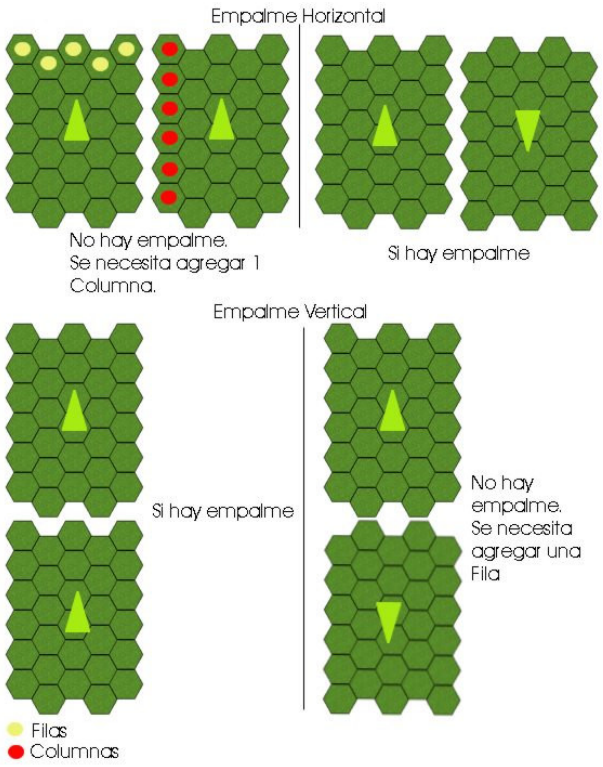


Figura 51. Empalme de mapas.

La primera versión del prototipo GuardarMapa (nombre que se le dio durante el desarrollo) era una versión que mostraba toda la información de los mapas en modo texto para luego pasar a lo que se ve en la figura 52.

La retícula hexagonal (hexMap ver 1.2) utilizada para la representación de los mapas y mas adelante para la representación del mapa en el sistema en este primer prototipo es una versión modificada especialmente para este sistema del componente creado por Patrick Kemp y Bob Dalton[57].

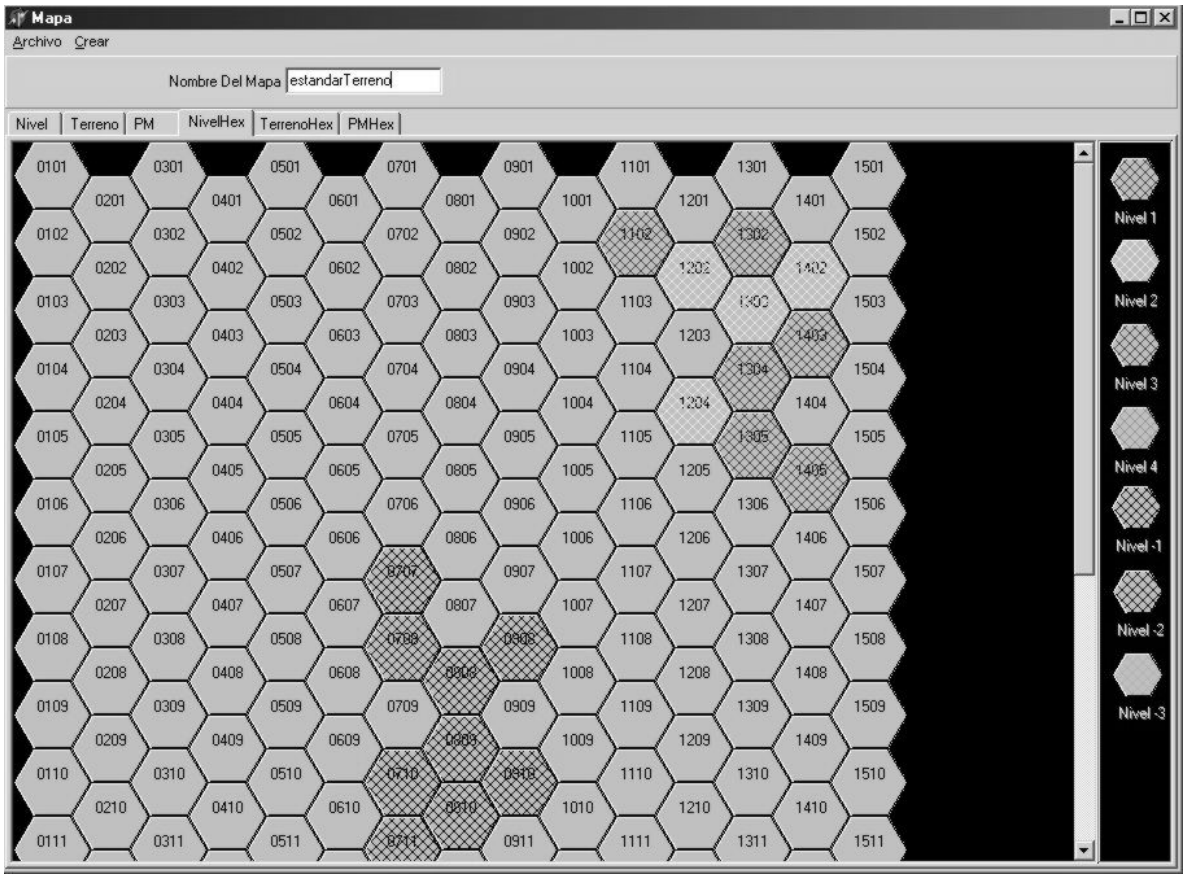


Figura 52. Guardar mapa.

Para diseñar el comportamiento del sistema se usaron diagramas de actividades para algunos de los métodos que pertenecen a las clases que conforman del sistema, estos métodos se detectaron durante el diseño del diagrama de clases. Sin embargo con estos diagramas se cometió un error, este error tuvo que ver con los diagramas y el diseño. Con UML se pueden crear diagramas muy

[57]Mustang Software Inc. www.mustang.com. bob.dalton@mustang.com

detallados, pero es importante saber que detalles agregar y en que momento detener el diseño y pasar a la realidad del código.

En este caso en la primera iteración, en la que se desarrollo el caso de uso movimiento, se cometió el error de demorar demasiado el paso a la codificación creando diagramas demasiado detallados. Este exceso de detalle consumió tiempo de diseño que no oporto al proyecto, no porque el diseño no fuera el adecuado sino porque se entro en mucho detalle que no venia al caso porque no se tenía nada codificado y era imposible predecir inconvenientes que solo se presentan en el momento de pasar al código los diagramas realizados.

Lo importante de haber cometido este error en la primera iteración fue que no volvió a ocurrir esta situación, la conclusión de esta experiencia es que no se debe entrar en demasiados detalles que son específicos de la fase de codificación.

Lo anterior no significa que este tipo de diagramas no sea de utilidad. Para el desarrollo del proyecto se hicieron unos 20 diagramas de actividades que sirvieron para representar diferentes funcionalidades del sistema. La figura 53 muestra uno de estos diagramas, este representa un método de mapa que permite determinar cuando dos hexágonos son adyacentes.

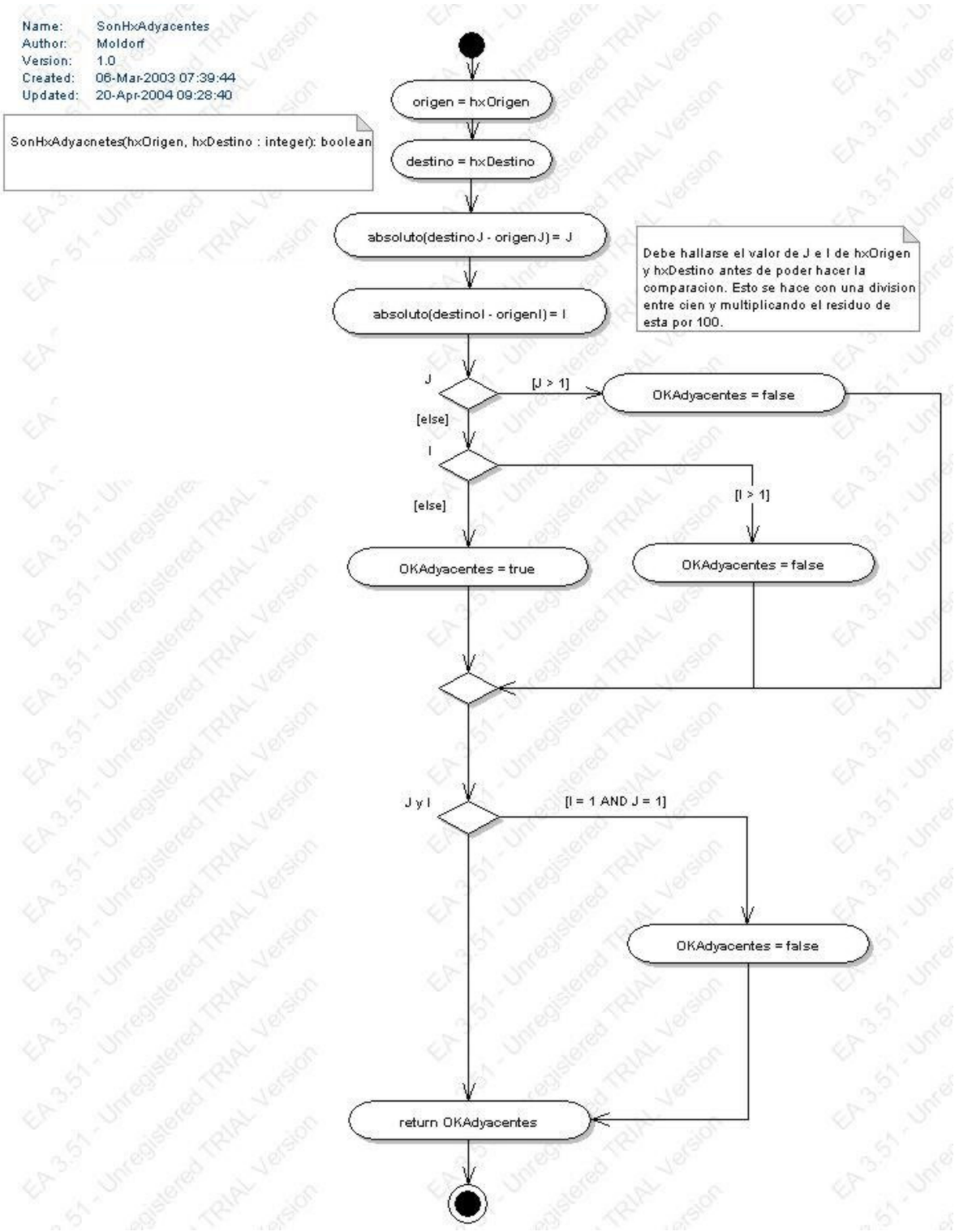


Figura 53. Método son adyacentes.

Lo que sigue después del trabajo anterior es llevar al código el diseño representado en los diagramas generados en esta fase.

4.2.1.2.3 Codificación. El objetivo de esta fase es construir una parte funcional del sistema partiendo de los modelos obtenidos en las fases anteriores.

Las definiciones de las clases procederán del diagrama de clases de diseño. Las definiciones de los métodos provendrán en su mayoría de los diagramas de colaboración y de las descripciones de los casos de uso. Los diagramas de estado serán de gran ayuda en este aspecto también permitiendo capturar condiciones de error.

Para el desarrollo de este proyecto lo primero que se hizo al iniciar esta etapa fue generar el código del diagrama de clases de diseño utilizando la herramienta de generación de código de Enterprise Architect (EA). EA genera código en delphi para los diagramas de clases. El código generado es de gran ayuda para comenzar la codificación de las partes del sistema al pasar por cada una de las iteraciones planteadas. EA genera cada clase en una unidad diferente y genera toda la declaración de las clases y además las definiciones de los métodos, tanto en la parte de la interfase (interface / type) de la unidad como en la parte de implementación (implementation) de la unida de código de Delphi. El código generado por EA para una clase simple, figura 54, se presenta en el siguiente párrafo.

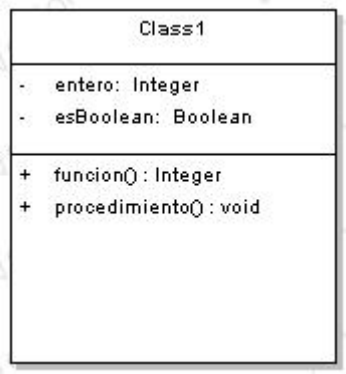


Figura 54. Clase simple

```
{***********************************************************
{*
{* Class1.pas
{* Delphi Implementation of the Class Class1
{* Generated by Enterprise Architect
{* Created on:      16-Apr-2004 23:40:28
{* Original author:
{*
{***********************************************************
{* Modification history:
{*
{***********************************************************}
unit Class1;

interface

type

  Class1 = class (TObject)

private
        entero: Integer;
        esBoolean: Boolean;

public
        function    funcion: Integer;
        procedure   procedimiento;

  end;

implementation

function    Class1.funcion: Integer;
begin

end;


procedure   Class1.procedimiento;
begin


end;
end.
```

Cada una de las clases generadas por EA se guardan en una unidad aparte, lo que corresponde a un archivo con extensión .pas para cada clase en delphi. En los comentarios iniciales del código anterior se puede ver que esta clase fue generada en el archivo clase.pas.
Lo siguiente que se hizo fue obtener la funcionalidad deseada para cada una de las clases del diagrama de clases de diseño.

En esta fase es de vital importancia el conocimiento de la herramienta en la que se esta codificando, para este caso el conocimiento de Delphi. El conocimiento de la herramienta hace que el proceso de codificación sea más rápido.

En esta fase cualquier herramienta que ayude en la producción de código, como por ejemplo el navegador de código para delphi CodeWarp desarrollado por Creative IT - Eric Grange liberado como Free Software que permite ganar tiempo en el momento de buscar una clase, procedimiento o función específica es de gran utilidad. Esta herramienta al igual que un foramateador de código y una herramienta para producir documentación del código de delphi en html fueron utilizadas durante todo el desarrollo del proyecto con el fin de aumentar la productividad en esta fase.

La herramienta para formatear código utilizada fue DelforExp, Delphi Formatter, creada por Egbert van Nes. Esta es una herramienta Freeware que funciona para delphi versión 2 a la 5. La otra herramienta mencionada es DIPasDoc 0.8.7 creada por The Delphi Inspiration - Ralf Junker liberada con licencia GPL que permite generar documentación del código en delphi en html.

Después de codificar el diseño del primer prototipo el resultado obtenido fue el premier prototipo.

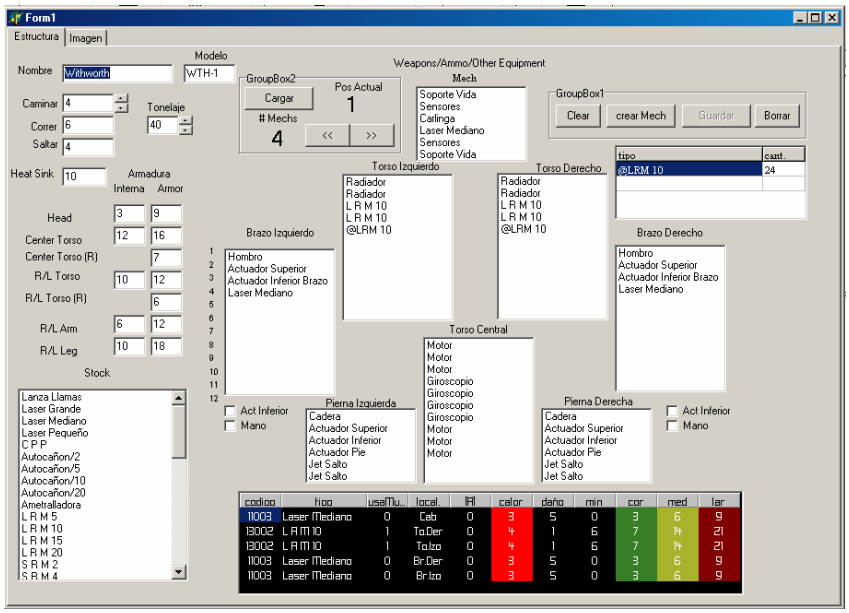


Figura 55. Primer Prototipo 1.

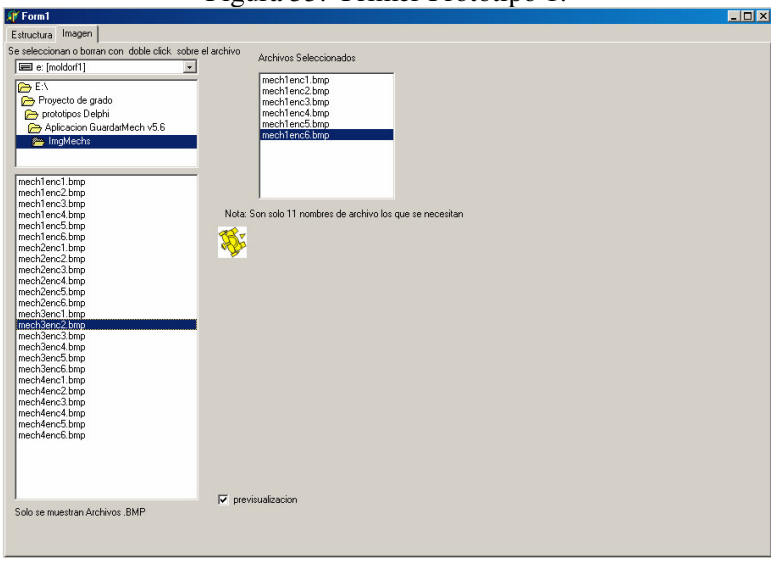


Figura 56. Primer Prototipo 2.

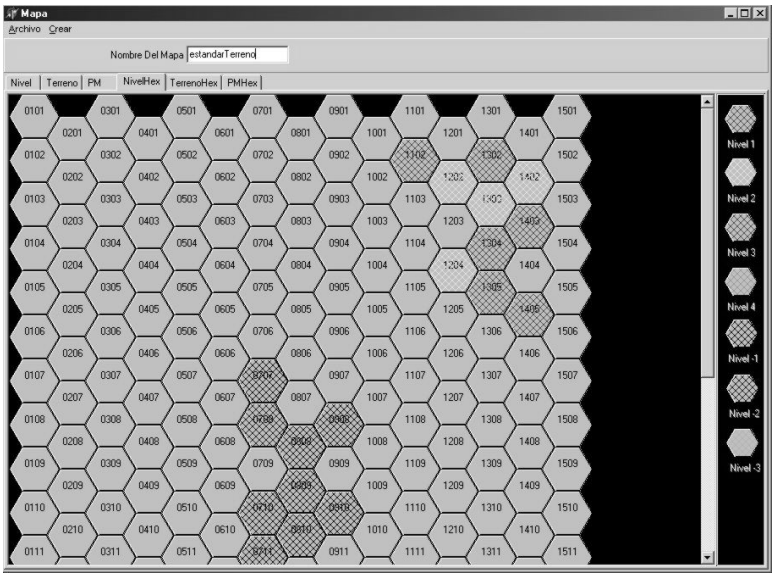


Figura 57. Primer Prototipo 3.

Uno de los grandes problemas del diseño y la codificación es mantener el modelo actualizado con el código. Para solucionar este problema al final de esta fase en cada iteración se actualizaban los modelos hechos en UML. EA fue de gran ayuda para llevar a cabo esta tarea de actualización, su capacidad de hacer ingeniería inversa desde el código en delphi permitió actualizar el diagrama de clases de una forma rápida y eficiente. Esta herramienta incluso introdujo las propiedades (properties) de delphi dentro de estos diagramas a manera de etiquetas (tags). La herramienta también fue capas de actualizar las relaciones entre clases que habían cambiado durante la implementación.

Estos modelos actualizados al igual que la documentación de código hecha con DIPasDoc permitieron manejar la complejidad del sistema a medida que iba creciendo. De no ser por esa documentación hubiera sido bastante difícil desarrollar toda la aplicación ya que al final de esta fase en la quinta iteración al momento de terminar el primer prototipo ya no era posible recordar por completo como estaba el diagrama de clases y mucho menos recordar todos los métodos que se tenían a disposición y que era lo que exactamente hacia cada uno de ellos puesto que ya eran demasiados.

Otro aspecto importante del final de esta fase es que se trato de hacer algo de Refactoring[58], sin embargo se hizo más a manera de prueba y no de una forma muy disciplinada. Los resultados fueron bastante positivos y lo único lamentable fue que el tiempo no permitió ahondar en esta metodología para poder aplicarla de una manera más seria y disciplinada.

Al terminar cada una de las iteraciones de esta fase se pasaba a la fase de pruebas, sin embargo es bueno aclarar que el hecho de tener una fase aparte para las pruebas no significa que en esta fase no se hayan realizado, lo que quiere decir esto es que es mucho mejor tener un espacio dentro del desarrollo dedicado específicamente a probar las partes de software desarrolladas dentro de escenarios pensados que podrían ser críticos. De esta forma es más probable producir un sistema libre de errores.

4.2.1.2.4 Pruebas. Para esta fase de desarrollo no se siguió ninguna metodología específica para pruebas de software. La estrategia que se siguió fue la de ejecutar la pieza de software construida en un escenario diseñado con anterioridad que tuviera aspectos que se consideraron como críticos.

Por ejemplo en el caso de este prototipo en la primera iteración en la cual el objetivo era calcular la iniciativa y permitir que los mechs se movieran un escenario cítrico era el siguiente:

Hacer pasar el mech por un hexágono de agua ocupado por un aliado.

¿Por que es este un escenario critico? Para este caso es crítico porque tiene que aplicar varias reglas juntas en una sola parte del movimiento. Debe calcular el número de puntos de movimiento gastados (incluye cambio de nivel y cambio de terreno), debe identificar al mech que se encuentra en el hexágono de agua como un aliado para luego si permitir al mech que esta jugando pasar por el hexágono. Por último debe cerciorase de que iba a pasar por ahí y que no termina su movimiento en ese hexágono, porque es ilegal tener dos mechs en el mismo hexágono.

Diseñando este tipo de escenarios, que incluyen tareas comunes y repetitivas del sistema así como casos atípicos o poco probables, se probaron cada una de las subversiones y versión final del primer

[58] Refactoring: Improving The Design Of Existing Code. First Edition. Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. Addison Wesley Pub Co. 1999.

prototipo. Al tener estos escenarios se ejecutaba la pieza de software unas dos o tres veces sobre estos escenarios para detectar fallas. A medida que se llegaba a la iteración número 5, estos escenarios se iban haciendo más complicados y una de las ventajas que se tuvo en para esta aplicación fue que al probar el producto de la tercera iteración era obligado ejecutar las partes creadas durante la iteración uno y dos. Por ejemplo, al momento de probar el producto de la iteración 2 que debía agregar la funcionalidad del ataque con armas, era necesario al correr el programa pasar por la fase calculo de la iniciativa y luego por la de movimiento, este aspecto de la aplicación fue ventajoso en esta fase porque además de volver a probar partes ya probadas, permitió detectar errores que tenían que ver con la agregación de nueva funcionalidad al sistema, errores que no tenían que ver con la pieza de software producida en cada iteración sino con la unión de esta pieza con las anteriores ya producidas, mas fácilmente.

Además de estas pruebas realizadas al prototipo final, tamben se realizaron pruebas a partes criticas del software, por ejemplo, en el caso de la clase mapa y su método que calcula la línea de visión (LOS), se realizó una prueba especifica para este método, ya que es critico para la fase de ataque del juego, que consistió en calcular la línea de visión desde cada uno de los hexágonos de una mapa estándar a cada uno de los hexágonos del mapa. Para el caso del hexágono de coordenadas 0101 se calculo la línea de visión desde este hexágono al hexágono 0102, 0103 … 1712, 1715 y de esta forma para todos los hexágonos del mapa.

Este tipo de pruebas se realizaron para varios componentes del sistema entre ellos los componentes gráficos, métodos como hacer daño al mech, cálculo de los puntos de movimiento y creación de objetos entre otros.

4.2.2 Segundo Prototipo. El segundo prototipo planteado para el desarrollo del sistema tenía las siguientes características:

Debía incluir un modelo preliminar en 3D del mapa en el cual se desarrollara el juego, además de los modelos en 3D de 4 de los robots que estarán en capacidad de intervenir en una partida del juego. Además de lo anterior se incluirá la opción de guardar la información del estado de una partida que no haya sido terminada en una sola sesión y se implementaran las ayudas de la herramienta y un prototipo de comunicación TCP/IP.

Estos requerimientos para el segundo prototipo se modificaron para agilizar el desarrollo del mismo. Como en esta etapa se empezaba el trabajo con OpenGL, el diseño del ambiente en 3D y el de los mechs se decidió dejar por fuera de este prototipo la posibilidad de guardar los datos de la partida, al igual que el diseño de las ayudas adicionales. Sin embargo durante el desarrollo del primer prototipo ya se diseñaron e implementaron algunas de las ayudas de juego que se tenían pensadas, este es el caso de las ayudas para la fase de movimiento (modificadores por movimiento), y la ayuda para fase da ataque con armas (calculo de la tirada de impacto).

La razón por la que se dejo por fuera la opción de guardar una partida al igual que la elaboración de las ayudas se debió a que era importante centrar toda la atención principalmente en el diseño de los mechs y el ambiente tridimensional debido a que para el desarrollo de esta parte del sistema se contaría con la ayuda de Camilo Vargas, diseñador industrial de la UIS, y era importante aprovechar al máximo su tiempo.

Además del ambiente tridimensional también se desarrollo el sistema de comunicación que debía permitir desarrollar una partida de battletech entre jugadores conectados a Internet que tuvieran una dirección IP.

La metodología se mantuvo igual a la que se utilizó durante el desarrollo del primer prototipo, planteando dos iteraciones dentro de la fase de construcción. La primera de ellas modificó el sistema para poder incorporar el sistema de comunicación, y en la segunda iteración se incorporo el ambiente 3D y los cuatro mechs con cuerpos tridimensionales.

Para esta sección del capitulo, al igual que para el desarrollo del siguiente prototipo, se va ha cambiar la estructura del documento, como ya se mostró como funcionaba la metodología en la sección anterior, y para este prototipo se sigue la misma, en esta sección no se va ha entrar en detalle acerca del desarrollo de la metodología y no se va ha mostrar que se desarrollo en cada una de estas fases. Esta sección estará dividida en tres partes, la primera de ellas mostrara el desarrollo de un prototipo preliminar que se creo para probar posibles soluciones al problema de comunicación. La segunda parte mostrara cual fue la solución para crear el sistema de comunicación de la aplicación y por último la tercera parte tratara sobre como se desarrollo el ambiente tridimensional, que herramientas se utilizaron y cual fue el aporte de Camilo Vargas.

4.2.2.1 Prototipo preliminar de comunicación. Antes de comenzar el desarrollo del sistema de comunicación y del ambiente tridimensional para la aplicación, se hicieron algunas pruebas preliminares con posibles soluciones para el problema de la comunicación con TCP/IP.

Lo primero que se debía decidir para la comunicación con TCP/IP era que protocolo se debía utilizar en la capa de transporte de TCP.

Las posibilidades para la capa de transporte TCP son dos: TCP o UDP. Cada uno de ellos ofrece ventajas y desventajas.

De UDP se puede decir que es un protocolo no orientado a conexión, no seguro, porque la entrega y la protección contra duplicados no esta garantizada, sin embargo se garantiza la integridad de los datos agregando una suma de verificación. UDP depende de IP para mover los paquetes en la red.

De TCP se puede decir que es un protocolo orientado a la conexión, que depende de IP para mover los paquetes en la red. TCP protege contra la perdida de datos, corrupción de datos, el reordenamiento de paquetes y la duplicación de paquetes adicionando una suma de verificación y un número de secuencia a los datos transmitidos y, en el lado que recibe, devolviendo paquetes de reconocimiento de los datos recibidos.

Además de lo anterior UDP es más rápido que TCP y en redes de área local la posibilidad de perder un paquete es casi cero. Para una aplicación de entretenimiento la velocidad es un factor primordial pero para este caso la confiabilidad de la información también lo es, no se puede perder el movimiento de ningún jugador.

El protocolo que se escogido fue TCP, porque el sistema no esta pensado para correr en redes de área local solamente; no tiene sentido implementar un sistema de secuencia de mensajes o de acuse de recibo cuando TCP ya incorpora uno. Otro aspecto a tener en cuenta es la seguridad que brinda TCP de no perder paquetes. ¿Que pasaría si el mensaje que contiene la información en la que el jugador A destruye la cabeza del mech del jugador B (perder la cabeza significa la perdida del mech y la eliminación del jugador) se pierde? Esta es una situación que se debe evitar. Si se esta utilizando TCP y alguno de los jugadores pierde la conexión es fácil de detectar mientras que en el caso de UDP no es tan sencillo.

Con el protocolo escogido las opciones para desarrollar un sistema de comunicaron usando sockets se pueden agrupar en dos: utilizar directamente la API desde Delphi o utilizar un componente de comunicación por sockets. La primera solución no era la mejor por varias razones. La primera, implicaría tiempo y trabajo para probar el modo como se hace la comunicación antes de entrar a probar los mensajes que tienen que ver con la aplicación. Segundo, esta API no es orientada a objetos así que lo mas lógico seria crear una estructura orientada a objetos, esto requeria tiempo y trabajo adicional.

La segunda alternativa suena un poco mejor, el trabajo que se tiene que hacer es probar alguno de los componentes de comunicación que trae delphi, que funcionan bien (como por ejemplo INDI) o buscar algún componente gratuito.

Después de hacer una búsqueda de componentes de comunicación y evaluar algunos de ellos utilizando demos y haciendo modificaciones al código de estos, el componente escogido fue ICS - Internet Component Suite[59] desarrollado por François Piette. Se escogió porque utiliza una jerarquía de objetos, es gratuito con código fuente y ha sido ampliamente probado por la comunidad en Internet.

El prototipo realizado a manera de prueba era capas de procesar tres tipos de mensajes a nivel de aplicación diferentes que permitían enviar tres clases de información diferente. Se podía enviar un objeto de tipo mech, un arreglo de enteros de tamaño fijo y también permitía el envió de una cadena de caracteres de máximo 80 caracteres.

La figura 58 y la figura 59 muestran el prototipo del cliente y del servidor.

[59] http://www.overbyte.be. François Piette francois.piette@overbyte.be

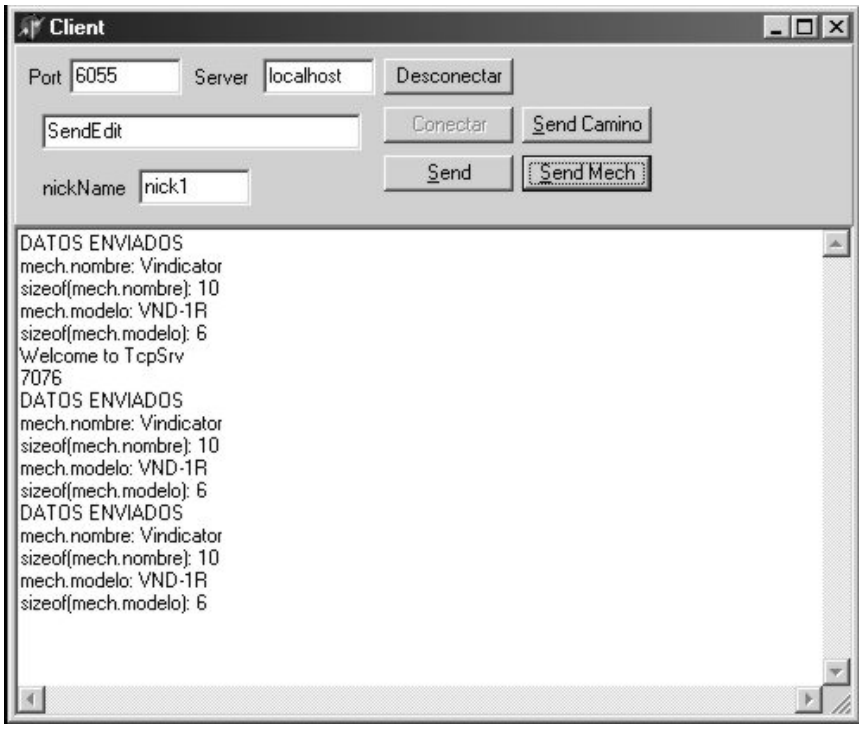


Figura 58. Cliente de prueba.

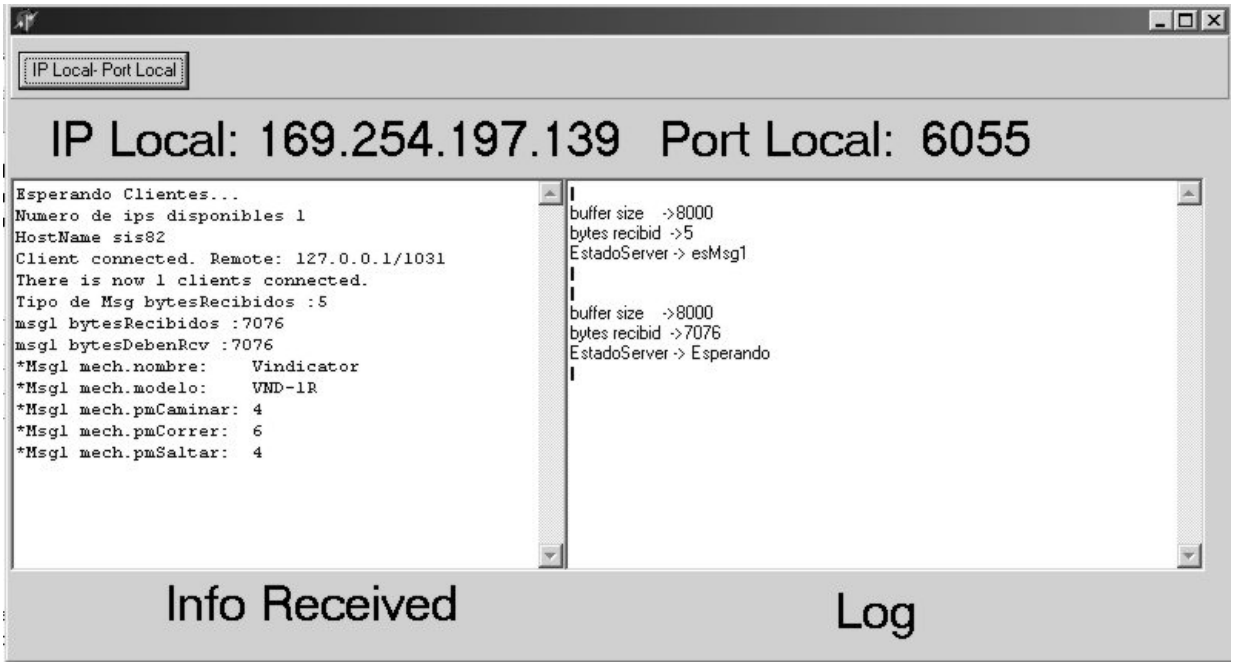


Figura 59. Servidor de prueba.

Es importante recalcar algunos aspectos de este componente que en su momento sirvieron para favorecer su escogencia. La clase básica de ICS es la clase TWSocket que encapsula el paradigma de los sockets de Windows, esto demuestra que la estructura de ICS es orientada a objetos. TWSocket es un componente completamente asíncrono (no-bloqueante), y manejado por eventos. El ser asíncrono, significa que si por ejemplo se le pide a un objeto de la clase TWSocket que se conecte (connect()), el componente comienza la operación que se le pidió que hiciera, pero inmediatamente devuelve el control del programa mientras que el esta haciendo la operación en segundo plano. Cuando la operación es llevada a cabo, para esta caso se establece la conexión, un evento se dispara (OnSessionConnected), lo que significa que es un componente manejado por eventos. Además de estas características también ofrece la opción de trabajar los sockets en hilos diferentes al principal (Multithreading) de manera segura.

Este prototipo además de servir como prueba de ICS, también sirvió para probar la idea que seria la base para el diseño del sistema de comunicación de la aplicación. El servidor se comporta como una maquina de estados que dependiendo de una cabecera de mensaje sabe que tipo de acción realizar.

4.2.2.2 Sistema de comunicación. En este punto del desarrollo ya se tiene el motor de reglas implementado y lo que sigue es determinar que objetos se quedan en el lado del cliente y que objetos en el lado del servidor, pero antes de poder determinar esto es necesario retomar el caso de uso general que representa una partida de battletech y que se muestra en la figura 22 del capitulo anterior. Los primeros tres casos son los encargados de configurar una partida de battletech así que estos tres casos de uso deben estar implementados en el servidor.

El primer caso de uso debe permitir la conexión de los jugadores al servidor y además en este caso de uso se incluyo la configuración de los equipo, esto es que jugadores y cuantos de ellos pertenecen a cada equipo; cada equipo debe pertenecer a alguno de los grandes poderes de la esfera interior.

El segundo caso de uso se refiere a la necesidad de asignar a cada jugador uno o varios mechs para que los use dentro de la partida. Esta opción solo esta disponible en el servidor para que el jugador que este usando esta aplicación pueda acomodar los mechs de acuerdo al escenario que éste previamente diseñó. Además en este punto el sistema se asegura de que ningún jugador se quede

sin mech para la partida, como para cada mech debe existir un mechwarrior, en este caso de uso se deben asignar los valores de las habilidades del mechwarrior de cada mech (habilidad de disparo y pilotaje).

Por último el caso de uso seleccionar mapa permite escoger cualquiera de los mapas disponibles y colocarlos en la configuración deseada de acuerdo a las reglas de battletech. Se pueden usar máximo cuatro mapas en cualquiera de las configuraciones que muestra la figura 60. Para el caso en que están los mapas acomodados de manera horizontal, uno al lado del otro unidos por sus lados mas largos, el hecho de que solo aparezcan dos mapas en la figura no significa que en esa configuración solo se puedan usar dos, para esa configuración se pueden usar tres o cuatro mapas siempre y cuando estén unidos por sus lados mas largos. La misma regla aplica cuando los mapas esta uno arriba de otro unidos por sus lados mas pequeños.

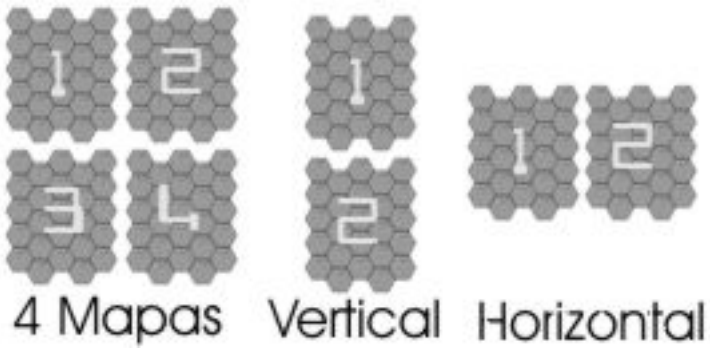


Figura 60. Configuración posible de los mapas.

Después de tener identificados los primeros tres casos de uso como partes del servidor y analizarlos detenidamente el problema inicial que se tenía aparece de nuevo. Que clases de las que pertenecen al desarrollo de la partida deben aparecer en el servidor y cuales en el cliente.

Dar solución a este inconveniente al principio no fue sencillo debido a algunas situaciones especiales que presenta la aplicación. Algunas de las soluciones disponibles para escoger podrían ser desarrollar un cliente liviano, en donde la mayor parte de la información y las reglas se mantienen en el servidor, un cliente pesado o una solución del tipo de n capas. Los aspectos que se tuvieron en cuenta para tomar una decisión sobre el diseño del cliente y del servidor fueron:

- Hay una regla de battletech que dice que todo jugador tiene acceso a la información de cualquier mech que se encuentre en juego en cualquier momento.
- Gráficamente deben existir todos los mechs que estén en juego en el cliente.
- Es deseable enviar mensajes de poco tamaño.
- Algunas reglas se pueden controlar del lado del cliente para agilizar el desarrollo juego y minimizar la cantidad de mensajes enviados al servidor.
- Hay una regla que se aplica en general al diseño de juegos que dice que es preferible guardar que computar. En este caso específico es preferible tener copia de algunos objetos y actualizarlos del lado del cliente y del servidor que pretender crearlos y destruirlos durante el desarrollo del juego luego de enviar la información necesaria al servidor o al cliente.

Teniendo en cuanta las anteriores consideraciones, las clases base estarán tanto en el cliente como en el servidor, la única diferencia será que la clase Tpartida que es la encargada de controlar el juego (controla los turnos y las fases dentro de estos) estará solo en el servidor.

Un diagrama de estados que muestra el comportamiento de esta clase se muestra en la figura 32 estados de partida del capitulo anterior. Es importante destacar que cada uno de estos estados corresponde casi exactamente a las fases que tiene cada turno de un juego de battletech.

Una de las ventajas dada por esta estructura en la que se tiene copia de los objetos de las clases base es que se pudo diseñar un sistema de comunicación que usa mensajes muy cortos como se explicara a continuación.

La figura 61 muestra el diagrama de estados del servidor.

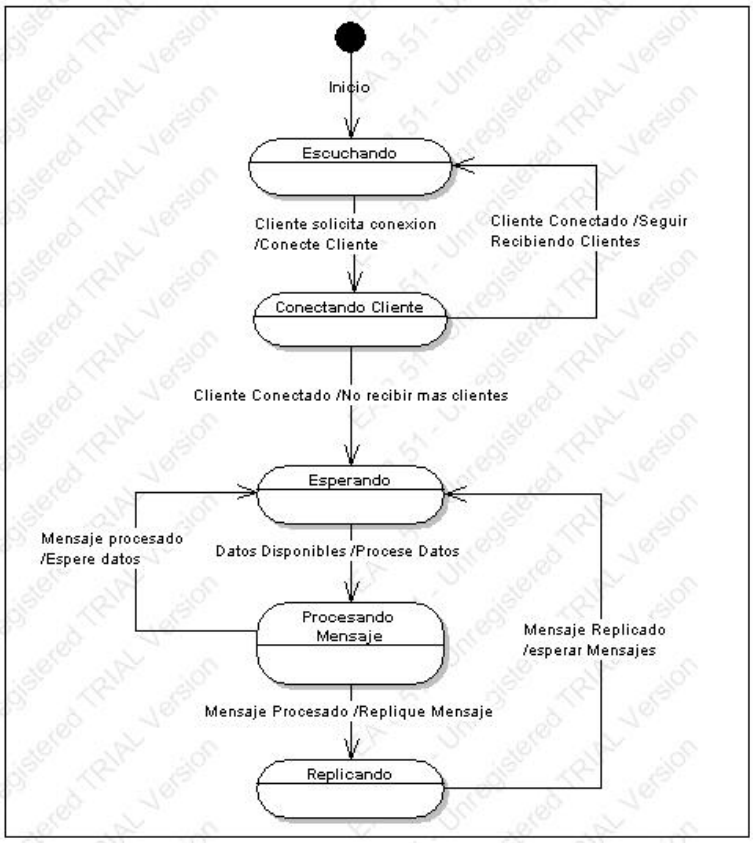


Figura 61. Diagrama de estados del servidor.

Lo que sucede con el servidor es lo siguiente:

- El servidor esta escuchando, listo para recibir clientes.
- Si algún cliente solicita conexión, lo conecta.
- Cuando el cliente esta conectado puede recibir la orden de seguir recibiendo clientes.
- Cuando el cliente esta conectado puede recibir la orden de no recibir mas clientes.
- En caso de no recibir mas clientes, se queda esperando por información en el buffer de datos recibidos
- Si hay datos en el buffer, los procesa.
- Si hay necesidad de reenviar los datos recibidos a todos los jugadores los reenvía.

El servidor después de procesar un mensaje siempre pasa al estado esperando.

El servidor recibe un mensaje que tiene la estructura que muestra la figura 62 en el nivel de la aplicación.

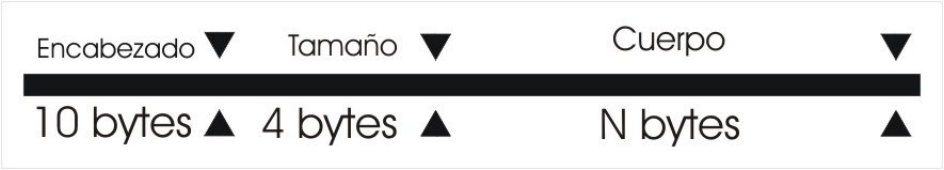


Figura 62. Estructura de un mensaje.

Las partes de este mensaje son:

- Encabezado: Este es un identificador de 10 bytes de tamaño. Lo usa el servidor para clasificar los mensajes y dependiendo del tipo de mensaje lo procesa.
- Tamaño: Este valor de tamaño se refiere al tamaño del cuerpo del mensaje (siguiente sección del mensaje), no del mensaje total
- El cuerpo del mensaje es de tamaño variable, su tamaño exacto esta definido en tamaño, y dependiendo del tipo de mensaje puede ser cero.

Al recibir un mensaje el servidor lo primero que hace es examinar la cabecera del mensaje que contiene un identificador de mensaje, este identificador le permite al servidor saber como debe procesar el mensaje. Dependiendo de esta cabecera el servidor sabe si debe leer el resto del contenido del mensaje y a donde enviar la información contenida en este.

Dependiendo del tipo de mensaje, si lo amerita, el servidor luego leerá los siguientes 4 bytes del mensaje que corresponden al tamaño del cuerpo del mensaje, es decir corresponden a los bytes que vienen después del byte número 14 hasta el byte final del mensaje que será el byte número N + 14.

Por último si el mensaje contiene un cuerpo, el servidor leerá los N bytes que le falta leer para haber leído por completo el contenido del mensaje.

Una funcionalidad importante que incorpora ICS es que posee una clase descendiente de TWSocket que contiene métodos especiales para que obtener la funcionalidad de un servidor. Este servidor (TWSocketServer) crea una instancia de la clase TWsocketCliente cada ves que un cliente se conecta, lo interesante es que gracias a la herencia es posible crear una clase propia descendiente de TWSocketClient y asignársela al servidor. De esta forma al momento de conectarse un cliente es

pueden crear instancias de los objetos relacionados con el cliente y que pertenecen a las clases base de la aplicación.

Utilizando esta característica de ICS el servidor crea copias de las informaron contenida en los clientes, solo de las clases base, obviando la creación de los componentes que tienen que ver con la parte gráfica, es decir no es posible ver el desarrollo de la partida en el servidor. Esta opción no se incluyo por dos razones.

La primera razón tiene que ver con el uso de los recursos. El escenario mas probable para ejecutar el servidor es que éste se encuentre ejecutándose en la maquina de alguno de los jugadores, así que correrá con un cliente dentro de la misma maquina.

La segunda razón es que a diferencia del juego de mesa en el que puede estar presente un juez que no participa en la partida en este caso no es necesario, ya que el objetivo de este juez es dar solución a problemas de reglas, situación que esta resuelta por la aplicación. Debido a esta situación no se hace necesario mostrar el desarrollo del juego en el servidor. Se podría pensar en que seria útil mostrar el desarrollo del juego a manera de espectáculo, dentro de un torneo por ejemplo, pero la mejor solución para este problema seria crear un visor de partidas que permitiera a jugadores en diferentes lugares observar la partida, no solo al usuario cuya maquina esta corriendo el servidor. Lamentablemente esta opción se sale del objetivo del proyecto y no se desarrollo ningún tipo de herramienta que permita ver el desarrollo de una partida.

Para finalizar es importante destacar que el hecho de tener copias de las clases base del lado del cliente y del servidor permitió liberar al servidor de ciertas labores de procesamiento que tienen que ver con la aplicación de algunas reglas, disminuyendo de esta forma el tráfico de mensajes y la cantidad de instrucciones que el servidor debe procesar.

Un ejemplo de esta característica se presenta en la fase de movimiento, cada vez que un jugador desea mover su mech de un hexágono a otro se deben realizar la comprobación de varias reglas por cada hexágono escogido. La secuencia de las reglas aplicadas se muestra en la figura 63.

Este diagrama que se presenta es una de las versiones preliminares que se hicieron para poder entender mejor el caso de uso moverse, pero es perfecto para mostrar las reglas que aplica el cliente.

Cada una de las actividades que se muestra en el diagrama implicaría una llamada al servidor para poder pasar a la siguiente, mientras que en el modelo actual estas reglas son aplicadas en el cliente y solo se envía al servidor la información del camino recorrido por el mech. Esto es un arreglo de hexágonos por los que paso el mech para que este guarde esta información y la replique a los demás jugadores.

La explicación de lo que sucede en el diagrama es la siguiente:

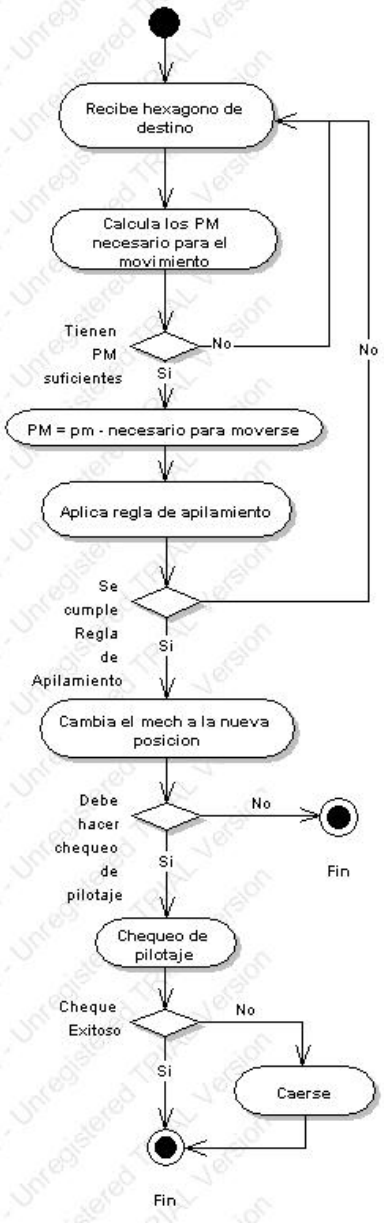


Figura 63. Diagrama Actividades para moverse. Fase Preliminar.

- Se comprueba que el mech que se esta moviendo tiene punto de movimiento suficientes para moverse.
- Si tiene los puntos de movimiento necesarios se descuentan de sus puntos de movimiento. Si no es así, vuelve a pedir un hexágono de destino.
- Después de descontados los puntos de movimiento el sistema verifica que se cumpla la regla de apilamiento. No pueden estar dos mechs en el mismo hexágono, sin embargo es posible que un mech pase por el hexágono de un mech aliado.
- Si la regla de no se cumple pide un hexágono de destino.
- Si la regla de apilamiento se cumple, el mech pasa al nuevo hexágono.
- Por último dependiendo del terreno y del estado del mech, se debe hacer un chequeo de pilotaje.
- Si el chequeo es exitoso, finaliza el movimiento. Si falla el chequeo debe caerse.

Es importante aclarar de nuevo que esta es una versión preliminar y simplificada de lo que ocurre en el caso de uso de movimiento.

Este es solo uno de los varios casos en que el cliente se hace cargo de algunas reglas del juego para evitar sobrecargar el servidor de operaciones y para disminuir el tráfico de mensajes entre servidor y cliente.

El otro caso en que esto sucede que vale la pena destacar es l caso del ambiente tridimensional. Los mechs de los demás jugadores que están actualmente jugando deben moverse en cada una de las interfaces de los clientes, y para solucionar el problema de replicar mensajes por cada uno de los hexágonos por los que pasa cada uno de los mechs el cliente controla esta parte independiente del servidor. Cuando un jugador se mueve envía un arreglo con el amino recorrido y al servidor y este lo replica a los demás jugadores del juego. En cada uno de los clientes existe un objeto cuya única función es animar el movimiento de cada uno de los mechs presentes en el tablero, así que el servidor solo trabaja con una arreglo y unos mensajes mientras que el cliente hace el resto del trabajo que tiene que ver con el movimiento del mech. Esto también es posible gracias a la copia de cada uno de los mechs que esta en juego del lado del cliente.

Sin embargo lo anterior no quiere decir que el servidor solo se limite a repetir todos los mensajes que recibe, no es así, en el servidor se aplican varias reglas del juego de gran importancia como el calculo de la iniciativa que permite saber que jugador mueve primero (dándole la desventaja sobre el que mueve de último), controla el desarrollo del juego haciendo que esta pase por los turnos y por las fases de estos y por supuesto el servidor determina quien es le ganador de la partida.

El diseño final del sistema cliente y servidor se muestra en el diagrama de despliegue que aparece en la figura 64.

Es necesario hacer algunas aclaraciones sobre los componentes que aparecen en el diagrama de la figura 64.

Los diagramas en UML pueden ser diagramas elididos y en este caso se esta haciendo uso de esta característica ya que no aparecen todas las relaciones que tienen los componentes del diagrama, solo se muestran las que son de interés en este momento y esas son: Para el componente GLScene (de esta librería se hablara en la siguiente sección) su relación de dependencia con OpenGL, la relación de dependencia que existe entre el componente ICS y a la API de sockets de Windows y las relaciones que existen entre los componentes propios del sistema (motor de reglas, uGráfica, etc.).

El componente uForm es el que contiene todas las clases que tienen que ver con los componentes que se pueden colocar dentro de un formulario, dentro de estas están la clase TForm, los descendientes de la clase TButton, etc. Este componente tiene una relación de dependencia con la VCL (no aparece en el diagrama). La razón por la cual no aparece la VCL directamente es porque se utilizaron algunos componentes que son modificaciones de los ya existentes en la VCL o nuevos componentes que no aparecen en esta.

El componente Motor de reglas es el componente que contiene todas las clases base que permiten aplicar las reglas de Battletech (figura 39).

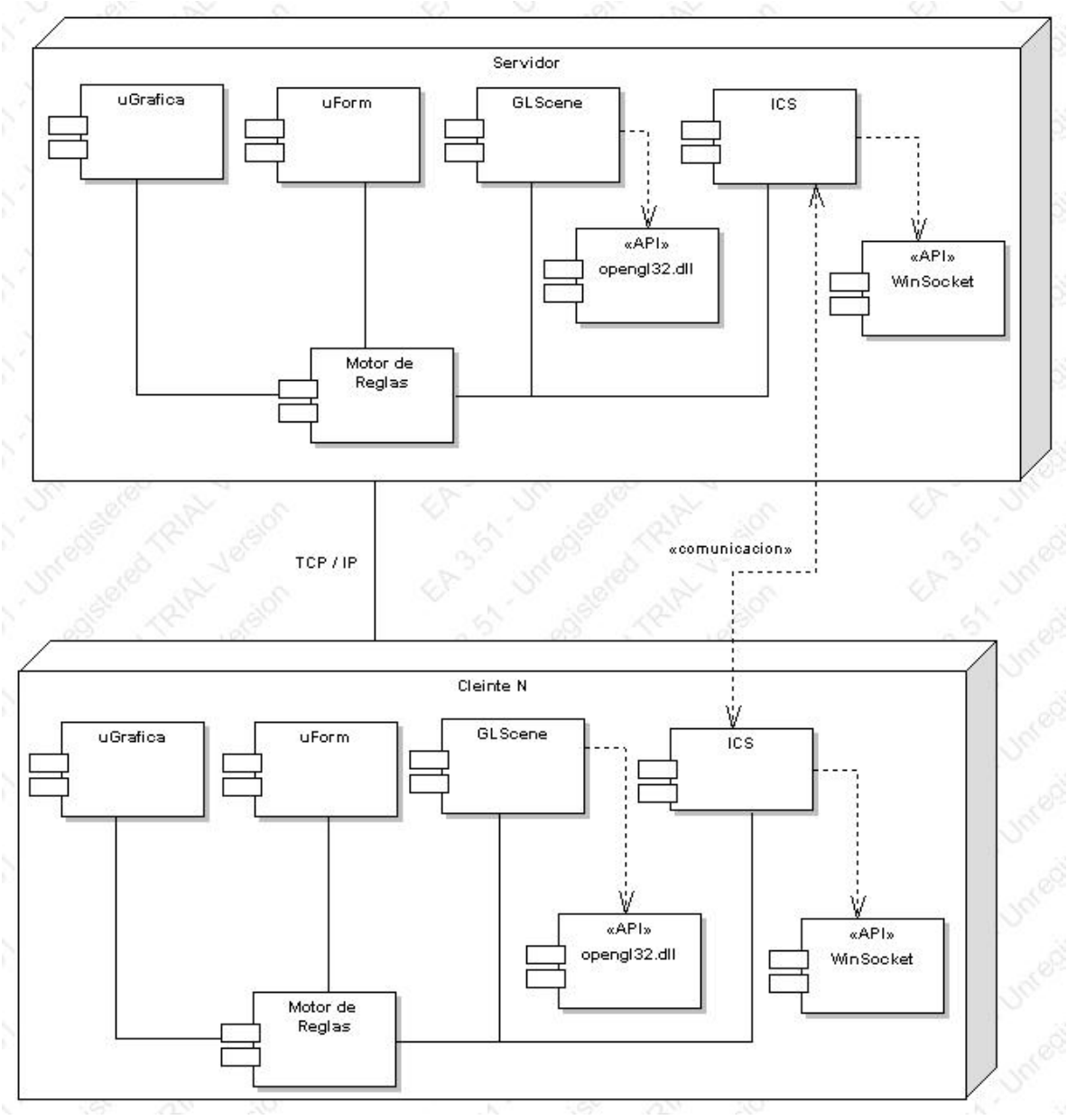


Figura 64. Diagrama de despliegue[60] de la aplicación.

[60] Esta palabra se traduce de diferentes formas dentro de los libros de UML, el nombre de este diagrama dentro de UML es Deployment Diagram.

4.2.2.3 Ambiente tridimensional. Para el desarrollo del ambiente tridimensional se necesitaba dar solución a la forma como se iban a desarrollar los modelos de los mechs y de los objetos que aparecen dentro de la aplicación.

Para dar solución a este problema, se presentan dos opciones. La primera es utilizar directamente OpenGL dentro de delphi utilizando alguna de las traducciones (oficiales o no-oficiales) de la API para Delphi. La segunda es utilizar algún componente que encapsule el funcionamiento de delphi.

La idea inicial era utilizar openGL desde Delphi directamente pero después de hacer algunas búsquedas en Internet se llego a la conclusión de que seria mejor utilizar algún componente de los que están disponibles de manera gratuita y con código fuente, esto evitaría dedicar tiempo adicional en el diseño y prueba del componente especialmente diseñado para la aplicación. Uno de los problemas iniciales era que estos componentes se presentaban de manera aislada, alguno de ellos permitía cargar archivos de tipo 3DS (archivos de escenas de 3ds max[61]), otro implementaba un sistema de partículas, otro era especial para generar superficies de alturas, etc. La única solución completa que se encontró fue GLScene y se empezaron a realizar pruebas y hacer investigaciones sobre su uso con resultados muy satisfactorios.

GLScene[62] una librería basada en OpenGL para Delphi que provee componentes visuales y objetos que permiten la geneneración de escenas tridimensionales que se distribuye como código abierto bajo la licencia Mozilla Public Licence[63]. Esta librería ha crecido tanto que ya no es solo una librería de utilidades o una envoltura (wrapper) de OpenGL sino que es un grupo de clases base para un motor genérico tridimensional.

Debido a todas estas cualidades y a la gran cantidad de personas que están usando GLScene en aplicaciones diversas como aplicaciones de inteligencia artificial, modelado tridimensional, diseño de interfaces y juegos entre otros de manera satisfactoria, se escogió esta librería para desarrollar todo el ambiente tridimensional del sistema.

[61] Programa de modelamiento y animación en 3D. Copyright © 2002 Autodesk, Inc. All rights reserved.
[62] http://glscene.org
[63] http://www.mozilla.org/MPL

Las pruebas iniciales se centraron principalmente en correr algunas aplicaciones que utilizaban GLScene observar el código y la forma como algunos de los componentes que posee eran utilizados. Después de lograr algún conocimiento de la librería las pruebas se centraron en el uso de un componente llamado Actor (clase TActor), que permite cargar archivos en formatos SMD, MD2 y MD3. Este tipo de formatos se usa para guardar modelos que representan objetos o personajes en juegos como Half-life[64] (smd) y Quake[65] (MD2 y MD3). Otro de los objetos en los que se centro la atención durante estas pruebas fue el objeto FreeForm que permite cargar una variedad de archivos que representan mallas (modelos tridimensionales) y que en este caso se probo con archivos con extensión 3DS que son generados por el programas 3ds max.

Después de esta etapa se decidió que los modelos que se necesitaban para representar los mechs y el ambiente tridimensional se harían en 3D Studio. Las razones de esta escogencia fueron varias que tenían que ver con la capacidad de la herramienta y también con su popularidad. Lo primero que se tuvo en cuenta fue el prestigio que tiene esta herramienta en el campo del desarrollo de juegos, esta herramienta es usada por mas del 80% de las compañías que desarrollan juegos actualmente, por esta razón es sencillo encontrar material relacionado con el uso de la herramienta y también es posible encontrar plug-ins o scripts (pequeños programas que corren dentro de 3ds max que sirven para propósitos muy específicos). Debido a su popularidad también era más probable encontrar un diseñador experimentado en el uso de este programa dentro de la universidad.
Otro aspecto que también se considero era la posibilidad de generar alguno de los archivos que soporta GLScene, estos son para el caso de los actores SMD, MD2 o MD3, y esto posible a través de plug-ins o scripts que exportan los modelos de 3ds max a estos formatos. En cuanto a los archivos 3DS, que es uno de los archivos soportados por Freeform, no hay ningún problema ya que es nativo de 3ds max.

Con la herramienta escogida el siguiente paso fue escoger el diseñador industrial con el que se trabajaría. Se escogió un diseñador industrial porque se quería a una persona de la UIS que conociera bien 3ds max y que inicialmente debía crear lo modelos tridimensionales de cuatro mechs que se habían escogido con anterioridad. La persona escogida fue Camilo Vargas.

[64] Valve software. www.valvesoftware.com.
[65] Id Software. www.idsoftware.com.

Al momento de empezar a trabajar en los modelos aparecieron varias preguntas que tenía que ver principalmente con el número de polígonos que debía tener cada modelo y con el tipo de animación que se iba a utilizar.

Primero que todo hay que hacer algunas aclaraciones sobre como es que se realizan las animaciones dentro de GLScene y dentro de 3ds max.

Las animaciones se realizan utilizando una técnica llamada Keyframe interpolation, cuya traducción seria interpolación de cuadros clave, sin embargo de a quien adelante se usara el termino en ingles ya que keyframe puede ser traducido de varias formas.

Keyframe interpolation es una variaron de una técnica de animación llamada keyframe animation, esta técnica consiste en dar la sensación de movimiento a través de la presentación de diferentes posiciones de un personaje rápidamente. Este tipo de animación es la que se logra al dibujar diferentes posiciones de un personaje en un libro, una en cada página y luego pasarlas velozmente. La figura 65 muestra como es hace este tipo de animación.

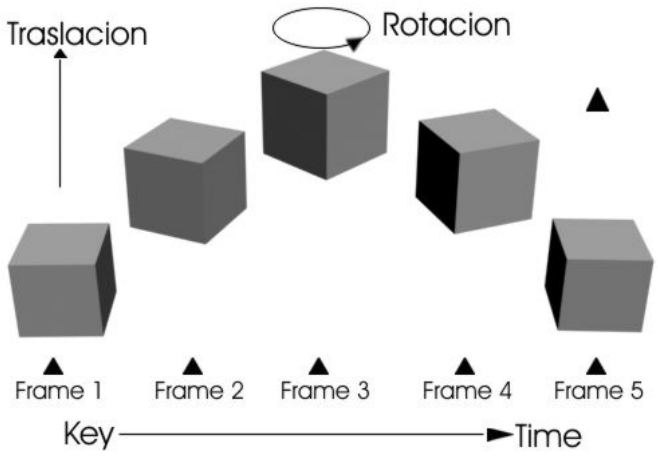


Figura 65. Keyrframe animation.

La figura 65 muestra 5 frames diferentes que serán usados para animar la traslación acompañada de la rotación de un cubo. El problema con este tipo de animación es que a menos que se tenga un gran número de animaciones, que consumirían una buena cantidad de memoria, la animación lucirá espasmódica.

Para el caso de la caja de la figura 65, al pasar del frame 1 al 2 se vería un salto. La solución a este problema es una variación de esta técnica llamada keyframe interpolation, esta técnica utiliza un menor número de frames en cada animación pero para generar una animación fluida hace una interpolación (lineal, B-spline) de los valores de posición y rotación existentes en cada uno de los frames. Para el caso de la caja de la figura 65, se podrían generar nuevos frames interpolando los valores de traslación y rotación de los frames 1 y 2 y de esta forma se produciría una animación más fluida, evitando el salto que se produce en la técnica de keyframes. El efecto de este tipo de animación seria algo como lo que muestra la figura 66, en donde se hace interpolación entre los frames 1 y 2 y entre los frames 2 y 3.

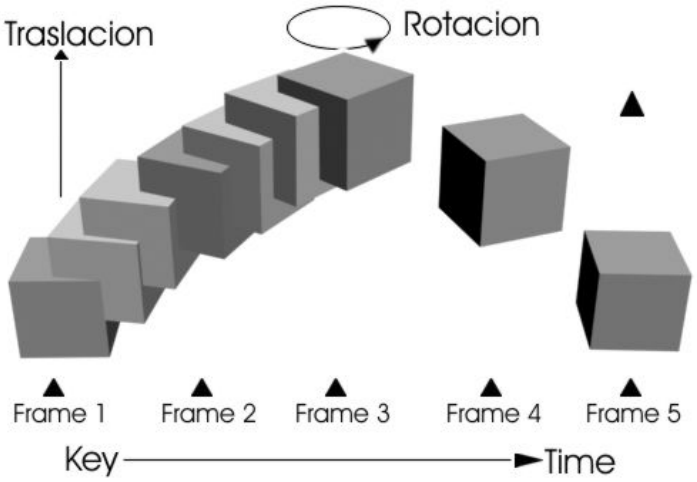


Figura 66. Keyframe interpolation.

Esta es la técnica que utiliza 3ds max y GLScene (aunque en GLScene es posible deshabilitar la interpolación). Sabiendo que esta es la técnica utilizada para las animaciones en GLScene y en 3ds max, lo importante ahora es como generar los frames de referencia (keyframes) para poder crear las animaciones, para esta labor es posible utilizar dos técnicas. Una de estas técnicas es el morphing[66] y la otra es una que utiliza esqueletos.

Es importante hacer esta distinción porque las dos pueden se usadas por GLScene, pero el trabajo que se debe hacer para producir los frames de referencia para cada una de ellas es bien distinto. Además el manejo de la información, en cuanto al uso de memoria también difiere, sin embargo

[66]En español cuando se habla de animación esta palabra no tiene traducción.

ambas técnicas pueden ser usadas dentro de 3ds max. La técnica de morphing es la usada por los archivos MD2 y MD3 y la que usa esqueletos por los archivos SMD.

La técnica de morphing consiste en crear un modelo que será el frame inicial y luego empezar a mover cada uno de los vértices de éste para crear los siguientes frames de referencia o frames objetivo. La figura 67 muestra a Sharky, un modelo creado por Camilo Vargas y animado usando esta técnica.

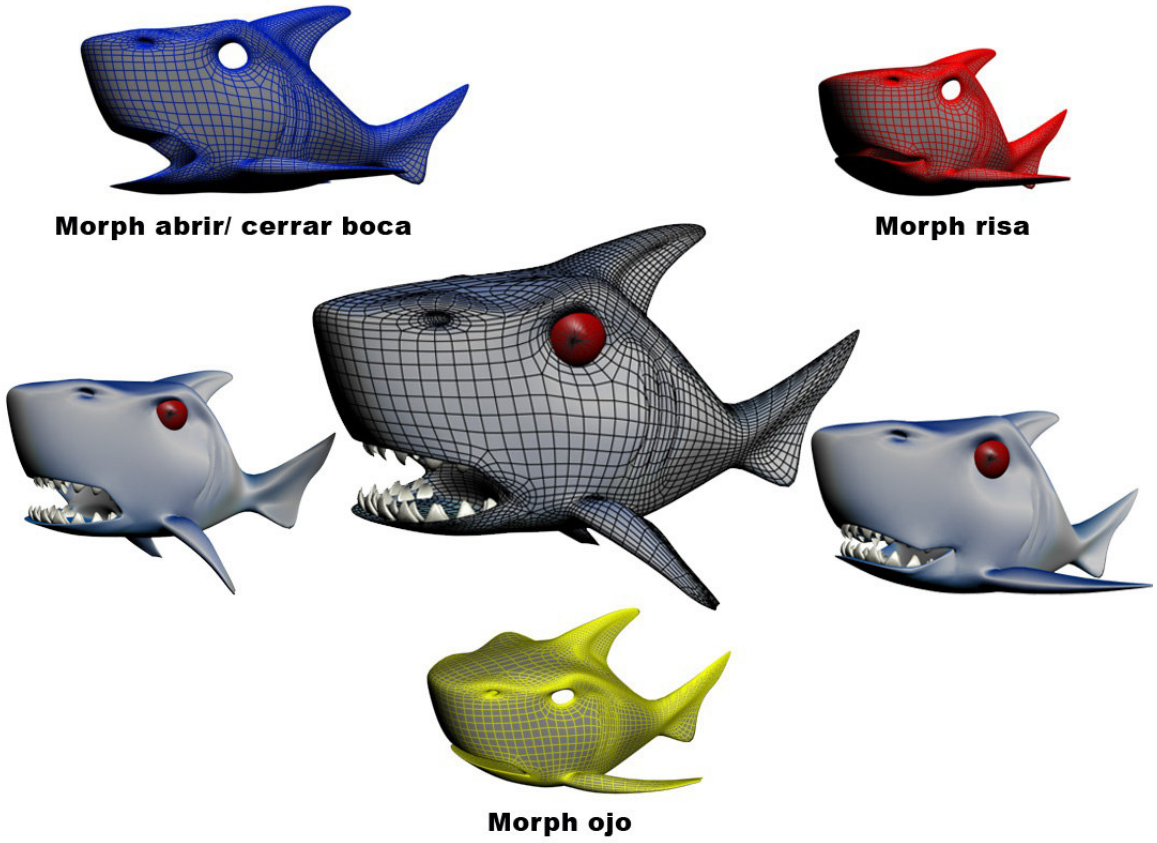


Figura 67. Sharky. Creado por Camilo Vargas.

Para el caso en el que el pequeño tiburón abre y cierre la boca, se debe modificar cada uno de los vértices (puntos de corte de las líneas) del modelo, afectados por la animación, creando primero un tiburón con lo boca cerrada, como el tiburón rojo, y luego uno con la boca abierta, como el tiburón azul. Desde luego es necesario crear frames adicionales entre estos dos para crear una buena animación.

Este tipo de animación es especialmente utilizado para crear expresiones faciales.

La otra técnica es la animación por esqueletos. Un esqueleto es una representación de la información de las relaciones que existen entre las partes móviles de una criatura, por ejemplo, el antebrazo esta atado al brazo y este a su ves esta atado al hombro. Haciendo una comparación con la biología, cada parte móvil es referida como un hueso, las relaciones entre cada hueso se llaman uniones y a la colección completa se le llama esqueleto. La figura 68 muestra uno de los mechs del juego con su esqueleto.

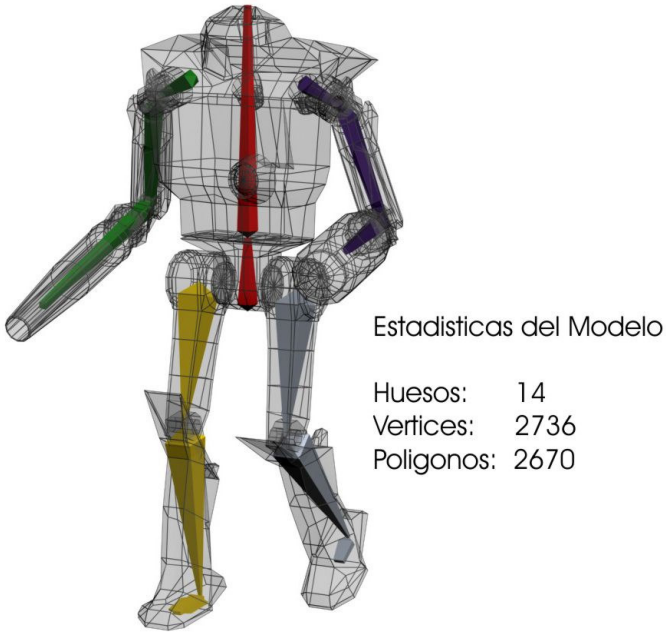


Figura 68. Mech con esqueleto a la vista.

El objetivo del esqueleto es que al igual que el esqueleto de un humano este tiene un área de efecto, es decir que al moverse debe mover la piel y los músculos que se encuentran encima. En el caso del modelo, cada uno de los vértices que forma la malla posee un peso que representa una relación entre un vértice y un hueso. En el momento en que el esqueleto es animado los vértices ponderados son transformados por el movimiento de los huesos. Cada uno de estos vértices se moverá en la dirección en la que se mueva el hueso y en el caso de las uniones se hará un trato especial dependiendo de la cantidad de influencia que tenga cada uno de los huesos que estén en la unión,

permitiendo que algunos de estos vértices se alejen y otros se junten en el caso en que la malla se comporte como la piel humana, comportamiento que muestra el cilindro de la figura 69.

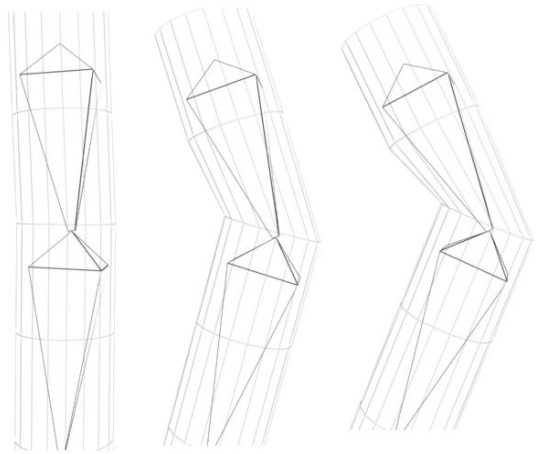


Figura 69. Cilindro con esqueleto.

Al grupo de vértices que forma la malla que esta sobre el modelo se le suele llamar piel digital. La figura 70 muestra el brazo de uno de los mechs del juego con un esqueleto de dos huesos, que representan lo que seria el brazo y el antebrazo. Esta secuencia se logra solo moviendo el hueso del antebrazo. Para este caso el comportamiento de la piel digital no es como el de la piel humana, los puntos de influencia en la zona de la unión se comportan diferentes porque el material de los robots no es deformable.

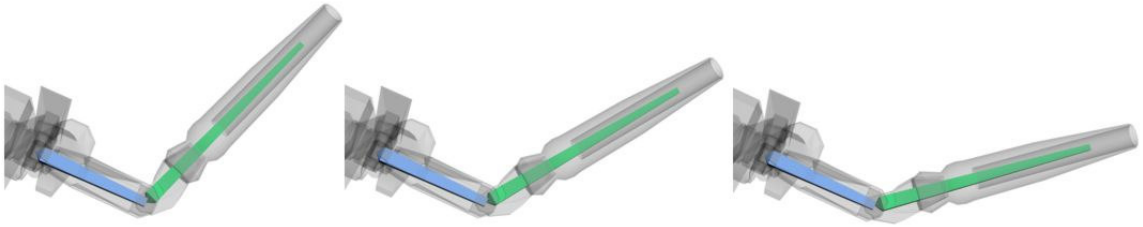


Figura 70. Brazo con esqueleto.

Los modelos de mechs que se utilizaron para el desarrollo del sistema poseen 14 huesos. La ventaja de las animaciones creadas con esqueletos, es que es más sencillo crear o corregir frames de referencia y por esta razón son mas usadas para crear efectos de movimientos de criaturas vivas, como humanos o insectos o como en este caso para crear animaciones de maquinas humanoides.

Teniendo en cuenta lo anterior al momento de crear una animación, fijando los frames de referencia, usando un modelo con un esqueleto y una piel digital configurada acorde a este solo es necesario cambiar los ángulos de los huesos y la malla (piel) que los rodea cambiara acorde con la posición que estos tomen. Desde luego lograr el efecto necesario no es tan sencillo como suena, pero es mucho mas sencillo que crearlo usando morphing.

Por esta razón se eligió la técnica de animación con esqueletos para hacer las animaciones de los robots. Además de esto también se tuvieron en cuenta algunas consideraciones que tienen que ver con el formato de archivo SMD que guarda este tipo de animaciones y que es el único que soporta GLScene que permite animación por esqueletos.

El formato de archivo SMD fue creado por la empresa Valve Software[67] al momento de crear un juego en primera persona llamado Half-Life. Este formato fue creado para guardar los personajes que aparecen dentro del juego. Sin embargo el juego usa una versión empaquetada de archivo (archivos .mdl) que contiene la malla (un archivo SMD), las animaciones (varios archivos SMD), las texturas y un archivo adicional (.qc) que es como una tabla de contenido de todo lo que hay en el archivo .mdl.

Los archivos SMD vienen en dos sabores. El archivo SMD de referencia, que contiene la geometría del modelo, la estructura del esqueleto, las texturas y las relaciones entre los vértices de la malla y el esqueleto. Este archivo de referencia no contiene ningún dato de animaciones.

Los archivos de animación SMD, por otra parte contienen solo información de la animación e información suficiente para que esta corresponda con el esqueleto guardado en el archivo de referencia.

Esta característica de tener un archivo para animación y otro para la malla es muy útil al momento de trabajar con archivos SMD y tratar de crear animaciones, mientras que con otros formatos que utilizan un solo archivo para guardar todo hay que crearlos de nuevo cada ves que se hace un cambio a un animación, para los archivos SMD solo hay que cambiar un archivo. Lo importante es

[67] www.valvesoftware.com

tener un buen diseño del personaje (malla y esqueleto), porque el cambiar la malla inevitablemente hará cambiar todas las animaciones ya creadas.

Otra aspecto interesante de los archivos SMD es que son archivos de texto, esta característica permite ver y editar los archivos sin necesidad de utilizar la herramienta de diseño en 3D con la que se exportaron los modelos, sino simplemente utilizando cualquier programa para edición de archivos de texto.

La tabla 2 muestra un fragmento de un archivo SMD de referencia y de uno de animación.

Tabla 2. Fragmento de archivos SMD de referencia y animación.

SMD referencia	**SMD Animación**
version 1 nodes 0 "Bone01" -1 1 "Bone02" 0 2 "Bone17" -1 3 "Bone18" 2 4 "Bone04" -1 5 "Bone05" 4 6 "Bone06" 5 7 "Bone08" -1 8 "Bone09" 7 9 "Bone10" 8 10 "Bone15" -1 11 "Bone16" 10 end skeleton time 0 0 -1.877116 -0.000000 0.883077 -0.000094 1.568268 -1.570890 1 4.886200 0.000000 -0.000000 -0.000000 0.004533 -0.000000 2 1.863674 -0.000000 0.883293 -3.128706 1.570784 1.581078 3 4.885565 0.000000 -0.000000 0.000000 -0.000021 0.000000 4 -2.123760 -0.000000 7.742714 -3.141591 0.420276 -3.141591 5 2.002910 0.000000 0.000000 1.570796 -0.872664 0.000001 6 4.238230 -0.000000 0.000000 -1.566321 -1.570796 0.000000 7 2.095963 -0.000000 7.742872 1.570796 0.419543 0.000000 8 2.002910 0.000000 0.000000 0.052159 -0.610865 -0.045425 9 4.238229 -0.000000 0.000000 -1.581795 -0.011063 -1.658063 10 0.000000 0.000000 -0.006795 1.570796 -1.570796 0.000000 11 2.457447 -0.000000 0.000000 0.000000 -0.000000 -0.000000 end triangles color1.bmp 11 -2.3234 1.2147 4.2858 -0.0564 0.9602 -0.2734 0.7402 0.7124 11 -0.6052 1.2147 4.2897 -0.0002 0.9893 -0.1456 0.5639 0.7126 11 -0.8909 1.0762 3.3661 -0.0632 0.9866 -0.1506 0.5932 0.6672	version 1 nodes 0 "Bone01" -1 1 "Bone02" 0 2 "Bone17" -1 3 "Bone18" 2 4 "Bone04" -1 5 "Bone05" 4 6 "Bone06" 5 7 "Bone07" -1 8 "Bone11" 7 9 "Bone12" 8 10 "Bone15" -1 11 "Bone16" 10 end skeleton time 0 0 -1.877116 -0.000000 0.883077 -0.000094 1.568268 -1.570890 1 4.886200 0.000000 -0.000000 -0.000000 0.004533 -0.000000 2 1.863674 -0.000000 0.883293 -3.128706 1.570784 1.581078 3 4.885565 0.000000 -0.000000 0.000000 -0.000021 0.000000 4 -2.123760 -0.000000 7.742714 -3.141591 0.420276 -3.141591 5 2.002910 0.000000 0.000000 1.570796 -0.872664 0.000001 6 4.238230 -0.000000 0.000000 -1.566321 -1.570796 0.000000 7 2.095963 -0.000000 7.742872 1.570796 0.419543 0.000000 8 2.004303 -0.000000 -0.000000 -0.000000 -0.000000 -1.149216 9 4.237414 -0.000001 -0.000000 0.000000 -0.000000 -0.001369 10 0.000000 0.000000 -0.006795 1.570796 -1.570796 0.000000 11 2.457447 -0.000000 0.000000 0.000000 -0.000000 -0.000000 time 1

Valve Software al momento de permitir que sus jugadores crearan modelos utilizado el formato de archivo SMD, también diseñó unos lineamientos que se deben seguir para que estos funcionen bien dentro de sus juegos. Estos lineamientos también se tomaron en cuenta para el diseño de los robots, aunque no se siguieron al pie de la letra aquellos que se diseñaron teniendo en cuenta hardware con menor capacidad de memoria y de cómputo (tanto para la tarjeta de video como para el procesador).

Los lineamientos que se tuvieron en cuenta fueron los siguientes:

- Se mantuvo un contorno significante. Se mantuvieron los polígonos que agregan "forma" al modelo, eliminado aquellos que agregaran detalles no significativos para el modelo. Por ejemplo, algunos de los mechs que se modelaron suelen tener algún tipo de antena en su cuerpo, este detalle no agrega forma al modelo así que se elimino.
- Se mantuvieron las coordenadas de las texturas.
- Se mantuvo la forma del objeto animado. Este punto tiene que ver con mantener el robot sin que se deforme durante la animación (ejemplo del brazo de la figura 70), ya que el robot esta hecho de metal y no se debe deformar.

Las anteriores sugerencias no aplican solo para los archivos de Half-Life son sugerencias que aplican para cualquier tipo de animación que se realice. Tener en cuenta estos lineamientos es mas un arte que una ciencia ya que hay casos en los que no hay una respuesta fácil de encontrar, afortunadamente se contó con la colaboración de Camilo Vargas, para el modelamiento de cada uno de los 4 mechs.

Además de los lineamientos anteriores, que tienen que ver con el modelado, también hacen unas sugerencias sobre el número de polígonos que debe tener cada modelo y el tamaño de las texturas. Estas sugerencias son :

- El número de polígonos debe estar entre 500 y 750. Esta sugerencia se cambió ya que se hace esta suposición para un equipo de 166 mhz con 26MB de RAM y una Voodoo con 1MB (video). Para nuestro caso después de ver las especificaciones de las tarjetas actuales y de hacer algunas pruebas se decidió que se iban a mantener los modelos en un rango entre 2500 y 3000, no más de 3000 polígonos.
- Las texturas deben ser cuadradas con tamaños en potencias de 2. Esto es 2x2, 4x4, … , 64x64, 128x128 etc. Esta limitación esta dada por OpenGL.

- Se recomienda que las texturas sean de menos de 256K, esto es un bitmap de 512x512.

Es importante anotar que existe una relación entre la complejidad de la geometría del modelo y la complejidad de las texturas. A mayor complejidad en la geometría del modelo, este dependerá menos de las texturas ya que tendrá un buen nivel de detalle y no dependerá de las texturas para obtenerlo. En el caso contrario, con un modelo con poca complejidad en la geometría deberá tener texturas que sean capaces de generar el detalle necesario para que el modelo se vea bien.

Teniendo en cuenta todo lo anterior se comenzó el diseño de los 4 mechs que ivan a estar dentro del juego. Los cuatro mechs que se escogieron para ser modelados de los 24 modelos disponibles en la caja de battletech 4 edicion fueron:

1. WTH-1 Whitworth
2. VND-1R Vindicator
3. ENF-4R Enforcer
4. TBT-5N Trebuchet

Estos mechs se escogieron teniendo en cuenta que se pudiera realizar una partida de battletech equilibrada en caso de cada bando escogiera dos de estos mechs.

Al momento de comenzar el desarrollo se hizo de vital importancia crear una herramienta que permitiera ver los modelos en formato SMD exportados, al igual que sus animaciones, dentro de OpenGL ya que se temía que existieran variaciones significativas de aspecto entre la versión creada en 3ds max y la exportada a formato SMD, sin embargo esto no fue así. Otra de las razones por las que se hizo necesario desarrollar esta herramienta fue para comprobar que el plug-in o el script que se pensaba utilizar estaba generando archivos SMD validos, esto es que era posible leerlos dentro de GLScene, ya que para poder exportar los modelos estos deben cumplir con ciertas características, por ejemplo, los modelos dentro de 3ds max deben tener una multitextura con todos los identificadores de texturas asignados.

La herramienta que se creo se muestra en la figura 71. Visualizador de modelos SMD.

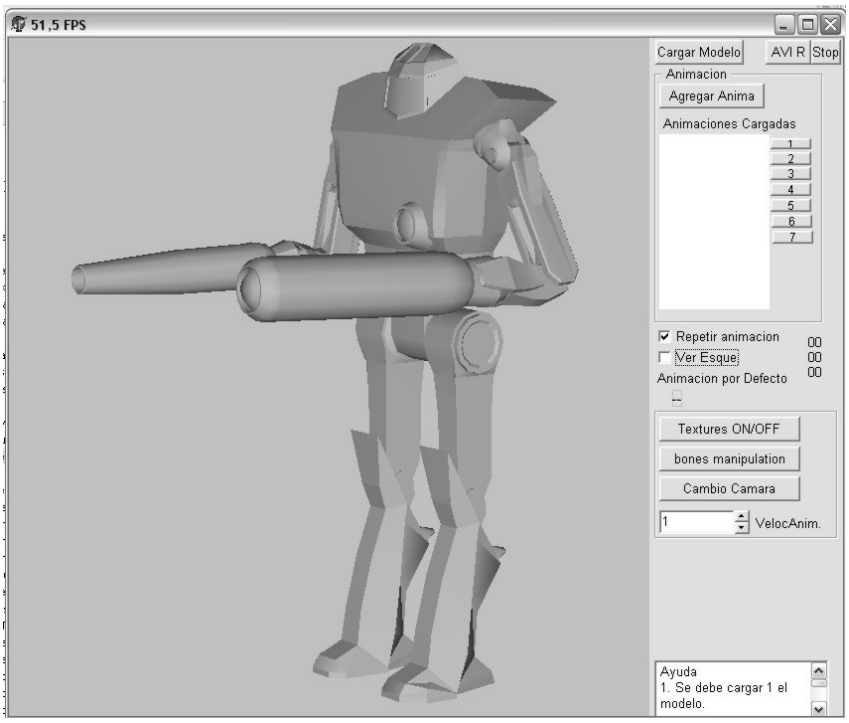


Figura 71. Visualizador de modelos SMD.

Para crear las animaciones que se debían tener dentro del juego se hizo uso de un diagrama de estados en el que se representa cada una de las posibles animaciones que se pueden tener y como se pasa de una a otra.

La figura 72 muestra el diagrama de estados para la animación de movimientos.

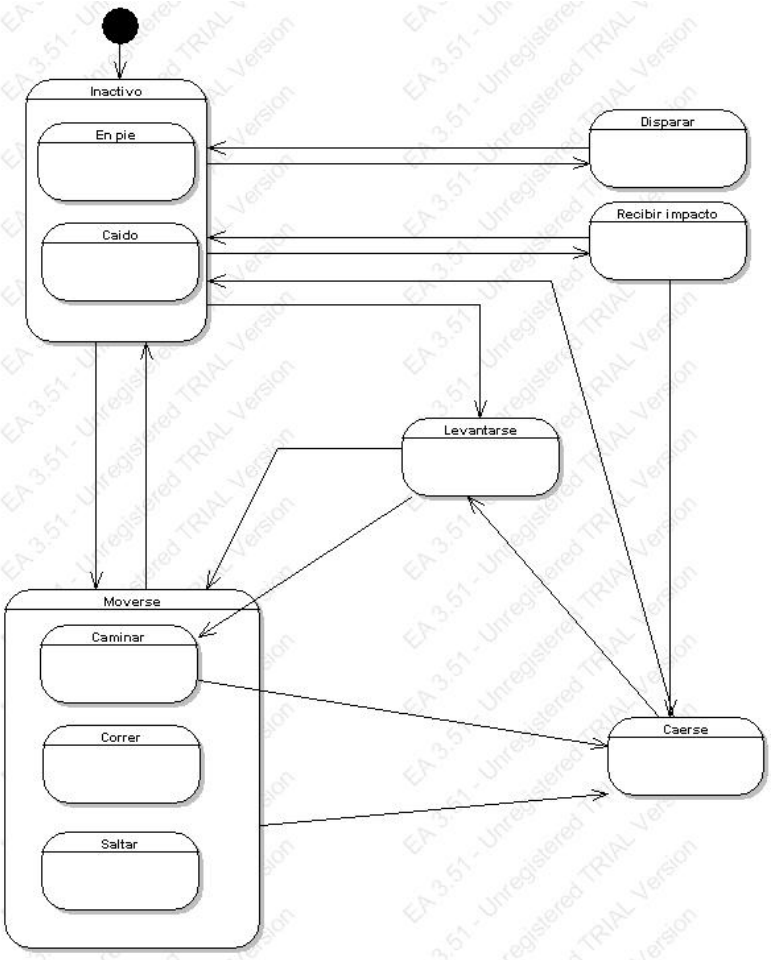


Figura 72. Diagrama animación de movimientos

Teniendo en cuenta el diagrama de la figura 72, se realizaron las 10 animaciones para cada uno de los 4 modelos de mechs que aparecen en el juego.

4.2.3 Tercer prototipo. La principal característica de este prototipo fue la posibilidad de guardar el estado de las partidas. El objetivo que se planteo para este prototipo durante la planeación del desarrollo del proyecto incluía las animaciones de los robots dentro del mundo tridimensional, pero durante el desarrollo se cambio debido a la colaboración del diseñador industrial. Como ya se dijo en la sección anterior se prefirió terminar todo lo que tenia que ver con el mundo en 3D en el prototipo anterior para aprovechar al máximo el tiempo de participación de diseñador en el desarrollo del sistema.

Otro de los objetivos de este prototipo era el de la realización de las pruebas finales de estabilidad y la detección de posibles errores que no se detectaron durante el desarrollo de los prototipos anteriores. Sobre estas pruebas finales se hablara en el capitulo 5.

4.2.3.1 Clases persistentes que guardar el estado de la partida. Para permitir guardar el estado de las partidas para poderlas terminar en otra (o varias) seciones de juego se hizo uso del componente Spider Containers And Persistent Classes[68] .

Este componente permite convertir cualquier objeto hijo de alguna de las clases que tiene el sistema en objetos persistentes, es decir, que permanecen entre ejecuciones del programa. Una forma de poder conseguir la persistencia de los objetos es a través de bases de datos entidad relación o de bases de datos orientadas a objetos. A parte de estas dos opciones es posible guardar la información de los objetos en archivos binarios o en cualquier otro tipo de archivo (formato propio de la aplicación, de text, etc.). Para el caso de esta aplicación se eligió este componente por varias razones.

- No hace ningún tipo de traducción, transformación para guardar los objetos en una base de datos entidad relación, ya que no usa ningún sistema de bases de datos exterior. Es decir no se hace ningún llamado a ningún programa externo.
- Es un componente desarrollado por completo en delphi, con código fuente disponible que hace uso de una de las características más interesantes de esta herramienta. Esta característica es la información de tipos en tiempo de ejecución o RTTI por su sigla en ingles (Run Time Type Information).
- El control de la información, para acceso y creación, permanece dentro de la aplicación misma, ya que es un componente que se compila y queda incluido dentro del ejecutable.
- Al permitir guardar los objetos sin tener que hacer ninguna transformación a tablas, no es necesario gastar tiempo adicional en el diseño de las tablas o formatos que se utilizaran para guardar los objetos. Implica un desarrollo de código adicional, ya que hay que codificar funciones adicionales para cada clase que es persistente, de escritura y lectura. Como se

[68] Copyrighted © 1996,97,98 Michel Brazeau. Interval Software. http://www.cam.org/~mibra/spider

desarrollan funciones para la clase, solo se debe hacer una ves por cada clase y todos los objetos de esta clase podrán ser persistentes.

- Es un componente gratuito para aquellas aplicaciones que no pretendan ser manejadores de bases de datos o competencia para otro componente comercial de esta compañía. Para mayor información es posible ver la licencia en la pagina Web http://www.cam.org/~mibra/spider**.**

Con las clases necesarias para llevar a cabo una partida definidas como persistentes, lo único necesario es definir en que momento de la partida se puede guardar el estado de esta. Aunque se puede pensar inicialmente que se debe poder guardar el estado de la partida en cualquier momento, la experiencia de juego del autor y las sugerencias de los jugadores, permitieron determinar que el mejor y único momento en el que se debía permitir guardar el estado de la partida es al comienzo de un turno de juego, es decir antes de la fase de iniciativa.

El permitir guardar el estado de la partida en otro momento implicaría incomodidades para los jugadores, por ejemplo, guardar el estado de la partida en la fase de ataque, en la cual es posible que un jugador salga de juego implicaría la posibilidad de que este jugador se conectara a una sesión de juego en la cual permanecería unos cuantos minutos nada mas. Otro ejemplo podría ser el de guardar la partida en la mitad de la fase de movimiento, esto implicaría que uno de los equipos, aquel que haya movido menos mechs, tendría una ventaja sobre aquel que ya movió en esa fase.

Por estas y otras razones que no vale la pena mencionar, ya que pertenecen a detalles muy específicos de las reglas, se decidió que solo antes de la fase de movimiento es posible guardar el estado de la partida y que solo lo podrá hacer el jugador que este corriendo el servidor, ya que este es el único que tienen toda la información necesaria para reconstruir por completo una partida. Sin embargo es posible enviar este archivo a otros jugadores para que otro de los participantes puede correr el servidor en otra sesión de juego, ya que el servidor esta en capacidad de cargar cualquier sesión de juego no terminada que este en un archivo valido.

5. PRUEBAS FINALES

En este capitulo se hablara sobre las pruebas finales que se realizaron al tener listo cada un o de los tres prototipos que se plantearon para el desarrollo del sistema.

Es importante aclarar que estas no fueron las únicas pruebas que se realizaron durante el desarrollo, ya que además de probar los prototipos finales se hicieron pruebas de partes de código que podían ser solamente métodos de objetos, pruebas a objetos completos o a grupos de objetos que interactuaban para crear un comportamiento deseado.

Estas pruebas fueron de gran utilidad, pero desarrollar este tipo de pruebas requiere tiempo y esfuerzo adicional. Lo ideal para realizar este tipo de pruebas es tener un grupo de desarrollo que esta dedicado exclusivamente a la prueba de software, sin embargo debido a que en este caso el equipo de desarrollo del proyecto era de solo una persona, fue necesario dedicar tiempo solo para desarrollar programas de prueba.

Lo deseado cuando se crean herramientas para pruebas de software seria poder desarrollar una prueba que diera como resultado "OK", si la prueba fue exitosa o una lista de fallas en el caso contrario, pero desarrollar este tipo de programas no es siempre fácil o no es siempre posible, así que en la mayoría de los casos también hay que incluir un tiempo adicional al de la prueba y solución de problemas en solamente analizar los datos obtenidos del sistema encargado de desarrollar la prueba de la pieza de software.

A continuación se entrara en detalle sobre como se hicieron las pruebas a cada uno de los prototipos terminados y también se mostraran algunas pruebas adicionales, que vale la pena mencionar, a partes de software antes de incluirlos en el sistema final.

5.1 PRUEBAS PRIMER PROTOTIPO.

La prueba final que se realizo al primer prototipo consistió en jugar varias partidas de battletech. Estas partidas que se jugaron eran partidas controladas, lo que esto significa es que se tenían diseñadas con anterioridad jugadas de ambos bandos para de esta forma lograr que el sistema pasara por puntos que se consideraban críticos.

Por ejemplo, en una de las partidas controladas que se jugaron, se pretendía probar principalmente el sistema de reglas de la fase de ataque con armas. En esta fase de ataque con armas es fundamental el cálculo de la distancia y el cálculo de la línea de visión, además de los cálculos de los arcos de disparo y las tiradas de impacto. Se quería probar que todas estas reglas estuvieran funcionando, así que se diseño una partida en la que el único objetivo de uno de los bandos era moverse a lugares en los que la línea de visión no fuera valida, en donde estuviera fuera de alcance, en donde la tirada de impacto fuera mayor a 12 o en donde los arcos de disparo solo fueran validos para un grupo determinado de armas.

Los resultados de este tipo de pruebas fueron bastante satisfactorios, no solo porque el sistema pasó la grana mayoría de estas pruebas, sino porque permitieron detectar errores dentro del sistema. Desde luego esta no es la mejor metodología que existe, ya que lo ideal es crear un grupo de pruebas desde las etapas iniciales del desarrollo que debería ser independiente del equipo de desarrollo, pero como se dijo anteriormente esto no era posible por la limitante de desarrolladores y también por la limitante de tiempo, ya que implementar una metodología de pruebas implicaría un estudio de por lo menos dos de estas metodologías para luego decidir cual de ellas aplicar.

El resultado de estas pruebas también fue bastante grato en términos de funcionalidad de la herramienta ya que no aparecieron errores críticos que implicaran cambios en el diseño o el rediseño de algoritmos, gracias a que la mayor parte de los métodos y objetos se probaron con anterioridad antes de incluirlos en el prototipo final. El hecho de haber utilizado una cascada en la fase de construcción por la que se paso varias veces también ayudo a obtener estos resultados positivos ya que a medida que se escalaba la aplicación era necesario correr de nuevo la parte de la aplicación previamente realizada y a que en este caso el sistema dependía fuertemente de los

desarrollos anteriores hechos dentro del desarrollo del primer prototipo. La fase de ataque se realiza después de la de movimiento, la de movimiento después de la de iniciativa, esto es se tiene una secuencia en cada turno. Desde luego estas pruebas adicionales tuvieron un costo relativamente alto en cuanto a tiempo de desarrollo ya que algunas de ellas implicaron el diseño de pequeños programas de cierta complejidad y además tiempo adicional de evaluación de resultados, pero este gasto adicional de tiempo fue mas que compensado ya que no hubo errores de consideración que retrasaran todo el desarrollo.

Uno de los ejemplos que vale la pena destacar de un programa de prueba realizado durante esta fase del primer prototipo fue el que permitió comprobar que se estaba calculando la línea de visión de A a B y la distancia entre A y B en hexágonos de manera correcta. La figura 73 muestra este programa.

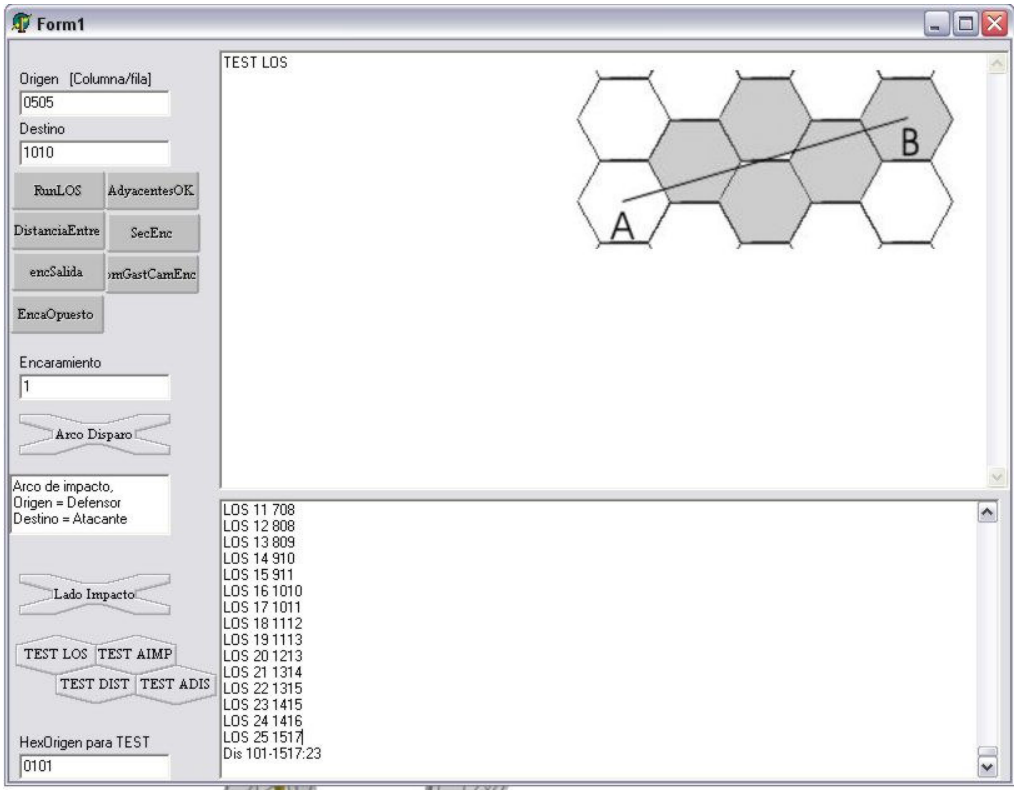


Figura 73. Programa de prueba LOS y distancia entre.

El resultado del programa se obtenía en un archivo de texto en el que se tenía la línea de visión y la distancia del hexágono de origen hasta cada uno de los hexágonos de un mapa de 15 columnas por 17 filas.

El resultado del cálculo para el hexágono 0101 al hexágono 1507 se muestra a continuación:

.....
LOS 0 101
LOS 1 201
LOS 2 302
LOS 3 402
LOS 4 503
LOS 5 603
LOS 6 703
LOS 7 704
LOS 8 803
LOS 9 804
LOS 10 904
LOS 11 905
LOS 12 1004
LOS 13 1105
LOS 14 1205
LOS 15 1306
LOS 16 1406
LOS 17 1507
Dis 101-1507:14
.....

Aunque parezca un poco exagerada la forma en que se probó que estos dos algoritmos funcionaban bien, fue necesario hacerlo, ya que estos dos algoritmos son básicos para el motor de reglas. Sobre este par de reglas descansa el funcionamiento de la fase de ataque con armas.

El desarrollo de esta parte de código, solo la de prueba, tomó dos días y tomó un día mas evaluar los resultados con ayuda de un programa de diseño grafico para poder determinar los hexágonos que exactamente tocaba la línea recta.

El desarrollo de las pruebas del primer prototipo, tomó más o menos cinco días. Durante el transcurso de estos 5 días se realizaron varias partidas controladas y otras en las que se dejo que otro jugador participara compitiendo con el equipo de desarrollo.

5.2 PRUEBAS SEGUNDO PROTOTIPO.

Las pruebas para el segundo prototipo siguieron los mismos lineamientos del primero, es decir se realizaron pruebas con partidas prediseñadas para comprobar que el sistema de comunicación estaba funcionando bien.

Durante el desarrollo de este prototipo también se construyo una pequeña pieza de software que permitía espiar los mensajes enviados entre el cliente y el servidor, para permitir corroborar que estos se estaban armando de manera correcta.

En esta etapa de prueba del segundo prototipo la experiencia no fue tan positiva como con el primero ya que las pruebas iniciales del sistema de comunicación se realizaron en una sola maquina y al realizarlas en una red real aparecieron algunos errores que tenían que ver con los tiempos de respuesta y las demoras que se creían solucionados en el momento de hacer las pruebas en una sola maquina.

A pesar de estos resultados no fue necesario hacer rediseño de algoritmos, solo fue necesario hacer pequeñas modificaciones que lograron solucionar estos problemas. Lo que si quedo demostrado es que el mejor campo de pruebas para un sistema cliente servidor es una red real, no es confiable hacer pruebas dentro de la misma maquina.

En cuanto a las pruebas que tenían que ver con el mundo tridimensional, estas se realizaron principalmente por fuera del prototipo, utilizando una aplicación especialmente diseñada para probar las animaciones y los modelos, y que luego reemplazaría al objeto encargado de animar las acciones en el primer prototipo y que inicialmente lo hacia con imágenes en 2D.

Además de esta pieza de software diseñada para probar el sistema de animación final, fue necesario desarrollar una pieza adicional que fue utilizada por el diseñador industrial para poder probar los modelos ya realizados en el ambiente en que se iban a utilizar, esta herramienta permitía hacer acercamientos, probar cambios de animación, desactivar texturas, observar el modelo desde varias perspectivas moviendo una cámara, cambiar las velocidades de animación, grabar secuencias en videos en formato *.avi, y observar el esqueleto del modelo.

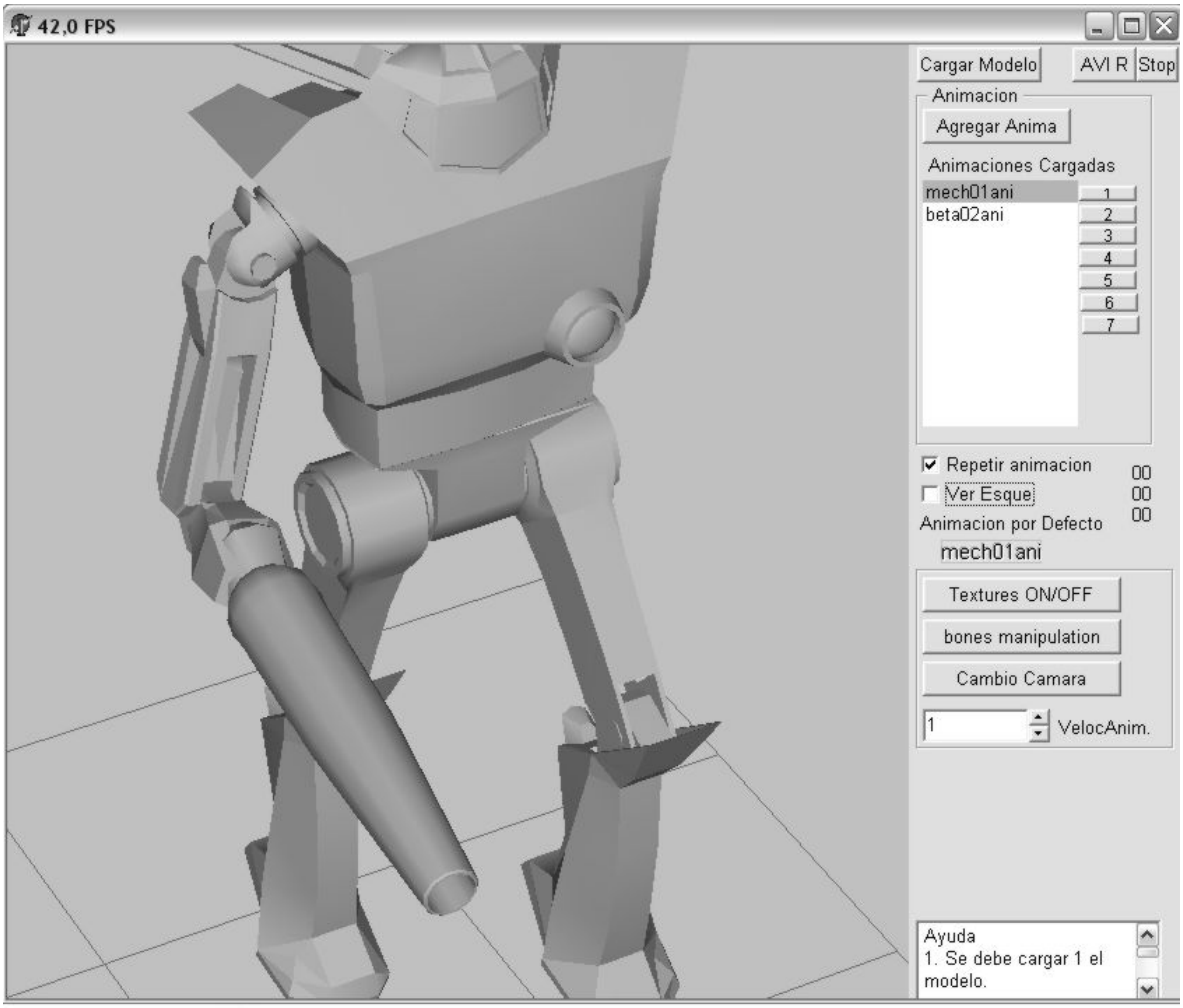


Figura 74. Herramienta para visualizar modelos SMD.

Después de que se terminaron las pruebas del mundo tridimensional, se incluyo este dentro del sistema y se probo de nuevo el sistema completo, esta vez no se utilizo ninguna partida prediseñada, solo se corrió la aplicación en diferentes partidas para comprobar que se había acoplado bien al

sistema las clases que se necesitan para lograr ver el ambiente tridimensional que simula el campo de batalla.

5.3 PRUEBAS TERCER PROTOTIPO.

El tercer prototipo se centro en la posibilidad que brinda el sistema de guardar los estados de la partida después al comienzo de cada turno. Esta fase de pruebas fue mas sencilla que las demás ya que una gran parte de objetos que se debían guardar para poder permitir guardar el estado de la partida ya eran persistentes, así que el trabajo que se realizo en esta fase fue el de probar que todo se estuviera guardando de manera adecuada.

La metodología que se utilizo para estas pruebas fue sencilla, se guardo el estado de la partida durante todos los turnos de una partida para comprobar que esta opción estaba funcionando correctamente. Cada vez que se guardaba se recuperaba el juego desde el archivo guardado.

De nuevo esto metodología de prueba puede parecer exagerada, pero era la única forma de asegurar que se estaba guardando la información correctamente.

Este tipo de pruebas consumieron tiempo adicional al del desarrollo de la herramienta, pero el objetivo de estas, era asegurar que el sistema estuviera libre de fallas. Obviamente el sistema no esta 100% libre de fallas, es posible y probable encontrar algún punto de ruptura ya que para lograr una la excelencia de calidad en el desarrollo del software es necesario combinar varias metodologías dentro del desarrollo de software, no es suficiente el haber utilizado una metodología es necesario asegurar la calidad utilizando procedimientos adicionales de aseguramiento de calidad.

Lo que si se puede asegurar es que el sistema producido es lo estable y permite recrear una partida de battletech en la cual pueden participar jugadores conectados desde lugares remotos, y esta estabilidad y confiabilidad esta garantizada siempre y cuando se mantenga el sistema dentro de los parámetros entre los que se realizaron las pruebas. Es decir si se mantienen uno requerimientos mínimos de hardware, se mantienen los requerimientos de software y se mantiene la carga de usuarios con la que se estima que el sistema es estable y fiable.

Durante el desarrollo de estas pruebas se descubrió que los requisitos mínimos de software y hardware para poder correr el sistema, tanto cliente como servidor son:

- Procesador: Pentium III / Atlhon de 750Mhz.
- Memoria RAM: 256MB
- Tarjeta de Video: 32MB de memoria
- Sistema Operativo: Windows 98SE/ Windows XP
- Tarjeta de Red
- 100MB de espoacio en el disco duro.

Es necesario hacer algunas aclaraciones sobre las características de la lista anterior.

Lo ideal seria que se utilizara una tarjeta de video con 32MB de memoria con aceleración por hardware de OpenGL.

Es posible que se pueda jugar solo teniendo una dirección IP, es decir podría ser posible jugar estando conectado por línea telefónica, sin embarga no se realizo ningún tipo de prueba con este tipo de conexión.

El sistema debería funcionar en Windows ME y en Windows 2000 pero no en estos sistemas no se realizaron pruebas exhaustivas. La aplicación se utilizo en Windows ME sin observar fallas, pero no se realizaron las pruebas suficientes.

El sistema funciona en sistemas que utilizan OpenGL acelerado por software o con menor cantidad de memoria de video, pero el sistema se vuelve inestable con el transcurrir del tiempo de juego, así que no es aconsejable. En este aspecto también es importante destacar que se observaron algunos problemas menores con tarjetas de video incorporadas en la tarjeta madre que comparten la memoria de video con la memoria RAM.

6. CONCLUSIONES

- El desarrollo de video juegos dejó de ser un juego. En el desarrollo actual de este tipo de aplicaciones están involucrados profesionales de diversas áreas, como ingenieros, diseñadores, publicistas, físicos, sicólogos, entre otros, utilizando diversas metodologías para desarrollo y aseguramiento de calidad de software.

  El desarrollo de juegos dejo de ser un área dominada por "hackers" adolescentes y pasó a ser una industria muy competitiva, y lucrativa si se logra crear un buen producto. El desarrollo de video juegos se convirtió en un problema de ingeniería de software de unas características muy especiales.

- Se logró desarrollar el sistema de entretenimiento en línea basado en el juego de estrategia battletech propuesto, con un diseño en dos capas que permite escalar el sistema a un segundo nivel de reglas o cambiar su interfaz grafica sin necesidad de realizar ningún cambio al motor de reglas, haciendo uso de la programación orientada a objetos y de una metodología iterativa e incremental.

- Se logro implementar un sistema de comunicación para el sistema propuesto utilizando sockets sin necesidad de recodificar su funcionalidad dentro de una estructura de objetos gracias al uso de una componente que ya cumplía con estas características y que había sido previamente probado por una comunidad de usuarios en Internet. Esto muestra una de las grandes ventajas de la filosofía de la orientación a objetos, la reutilización de código y la encapsulación de la información.

- Fue posible desarrollar una interfaz tridimensional para el sistema de entretenimiento gracias a la colaboración de un diseñador industrial y al uso de OpenGL.

- El desarrollo de juegos se puede considerar como un desarrollo de una herramienta atípica debido a que incorpora factores que no se encuentran comúnmente en el desarrollo de herramientas empresariales, como por ejemplo computación grafica, desarrollo de componentes específicos, desarrollo de métodos de comunicación, entre otros.

- Se evidenció la necesidad de realizar buenos diseños, ya que con las herramientas actuales es posible generar lo que se podría llamar código estático, estructuras de clases, pero no es posible generar el código de los métodos. Con el tiempo estas herramientas mejoraran y el siguiente paso es la generación de código para métodos a partir de diagramas de actividades o quizás de interacción. Por esta razón saber programar no es suficiente, de hecho podría llegar a ser innecesario en el futuro.

- Resulta importante probar varias metodologías de desarrollo de software para poder establecer pros y contras de cada una de ellas.

- Se percibió la necesidad de conocer y usar herramientas CASE como mejora de la productividad de los desarrolladores de software.

- Se aprendió a usar herramientas CASE como ayuda para documentar y especificar software de una manera eficiente, permitiendo de esta forma mejorar la calidad del producto final.

- Se evidenció que el uso de UML como lenguaje de modelado no solo aporta al equipo de programadores sino que también es de gran ayuda para comunicarse con profesionales de otras áreas (diseñadores Industriales) involucrados en el desarrollo o con usuarios finales del sistema (jugadores).

- Gracias al uso de componentes para Delphi de código abierto gratuito y librerías gratuitas como OpenGL, se podría pensar en hacer desarrollos de software con menos presupuesto e inversión en herramientas, quizás sin inversión en herramientas de desarrollo ya que actualmente existen proyectos como FreePascal, de licenciamiento gratuito, que permitirían adquirir estos productos sin hacer ninguna inversión. Sin embargo, un inconveniente que trae el uso de estas herramientas es que muchas de ellas poseen una documentación pobre y esto genera un aumento en el tiempo de desarrollo, ya que la única forma de descubrir su funcionamiento es a

través de las pruebas o estudiando código generado por terceros que utilizaron la herramienta. Otro aspecto que hay que tener en cuenta con estas herramientas es que no todas son estables y confiables, se deben realizar pruebas preliminares y hacer investigaciones para saber quiénes han usados estas herramientas con éxito. Sin embargo, algunos de estos problemas también pueden presentarse en herramientas comerciales.

- Debido a la versatilidad, importancia y gusto de los consumidores por interfaces graficas novedosas (en 2D o 3D), se hace necesario que los desarrolladores de software actuales posean conocimientos básicos sobre diseño grafico y uso de herramientas de diseño en 2D y 3D.

- Fue posible la creación de un grupo multidisciplinario durante el desarrollo de la parte gráfica en 3D del proyecto al trabajar con un diseñador gráfico. Esto fue posible gracias al uso de UML y al intercambio de información específica de diseño y de programación entre las partes sin entrar en detalles innecesarios y dejando que cada una de ellas hiciera lo que le correspondía.

- Fue posible usar tecnologías que son la base de nuevas opciones para el manejo de la información, como fue el caso del componente para la creación de objetos persistentes, que es la base de una base de datos orientada a objetos.

- Resultó interesante y enriquecedor fusionar dos metodologías de desarrollo de software, ya que esto permitió profundizar en cada una de ellas y de esta forma identificar falencias y fortalezas existentes dentro de cada una de estas metodologías.

- Se evidenció una falta de funcionalidad y estética al momento de crear las primeras interfaces de la herramienta que fueron descubiertas por el diseñador gráfico participante en el desarrollo del sistema. Gracias a esto fue posible solucionar este problema en prototipos siguientes a la identificación de este.

7. RECOMENDACIONES

- Seria interesante desarrollar un proyecto con tres objetivos, que daría continuidad al presente. El primero de ellos se centraría en la interfaz 3D aumentando la cantidad de animaciones para dar realismo y aumentando la cantidad de mechs disponibles; el segundo agregaría los niveles 2 y 3 de reglas de battletech y el tercero se centraría en crear la inteligencia computacional que permitiera realzar partidas entre jugadores y máquina.

- El presente proyecto podría constituir una base para desarrollar nuevos proyectos centrados en el entretenimiento, especialmente los utilizados en educación.

- Seria interesante desarrollar más proyectos que incluyan el uso de librerías como GLScene, que son gratuitas y con código fuente, para de esta forma apoyar el desarrollo de este tipo de iniciativas. El uso de estas librerías permitiría mejorar la producción de software educativo, juegos, documentación en línea, instructivos para capacitación empresarial, apoyo a pacientes con enfermedades incurables entre otros.

- Debido a la variedad de metodologías que se están utilizando en el mundo actualmente, como proceso unificado, programación extrema, prototipado, entre otras, seria de gran importancia fomentar grupos de estudio de estas nuevas metodologías.

- Gracias al auge de las herramientas CASE, sería interesante que se hicieran estudios sobre el tipo de herramientas que existen en el mercado, las ventajas y desventajas del uso de ellas, las capacidades actuales de estas herramientas y hacia dónde se dirige el desarrollo de las mismas, para incentivar su uso en la EISI, además de muchas disciplinas adicionales que se están utilizando durante el desarrollo del software, por ejemplo, metodologías para pruebas, uso de patrones, reconstrucción de software (Refactoring Software), metodologías de aseguramiento de calidad de software, para mencionar algunas.

- Es necesario estimular la realización de proyectos multidisciplinarios de software durante la carrera, ya que la complejidad de los desarrollos actuales de software requiere de equipos que reúnen profesionales de diversas áreas, y esas habilidades deben desarrollarse antes de enfrentarnos a la vida profesional.

BIBLIOGRAFÍA

PROGRAMACIÓN ORIENTADA A OBJETOS. Segunda Edición. Luis Joyanes Aguilar. McGraw Hill. España. 1998.

PROGRAMING IN AN OBJECT-ORIENTED ENVIROMENT. Raimund K Ege. Academic Press Inc. United Kingdom. 1992. 300p.

OBJECT-ORIENTED SYSTEM ANALYSIS. Modeling the World in Data. Shaller Sally, Stephen J. Mellor. Yourdon Press. United States of America. New Jersey. 1998. 144p.

OBJECT-ORIENTED MODELING AND DESIGN. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. Prentice Hall International Edition. New Jersey. 1991.

DISEÑO ORIENTADO A OBJETOS. Grady Booch. Traducción: Jaime O. Albarracin. Universidad Industrial de Santander. Bucaramanga. 1998. 176p.

ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS CON APLICACIONES. Grady Booch. Addison-Wesley/Diaz de Santos. Willimntong, Delaware. E.U.A. 1996.

OBJECT-ORIENTED SOFTWARE CONSTRUCTION. Bertran Meyer. Prentice may International. United Kindom. 1998.

FUNDAMENTALS OF OBJECT-ORIENTED DESIGN IN UML. Meilir Page-Jones. Addison Wesley. New York 2000.

OBJECT-ORIENTED MODELING AND DESIGN FOR DATABASE APLICATIONS. Michael Blaha, William Premerlani. Prentice-Hall Inc. New Jersey 1998.

REFACTORING: IMPROVING THE DESIGN OF EXISTING CODE. First Edition. Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. Addison Wesley Pub Co. 1999. 464 pag.

THE UNIFIED SOFTWARE DEVELOPMENT PROCESS. Ivar Jacobson, Grady Booch, James Rumbaugh. Addison Wesley Object Technology Series. 1999.

UML DISTILLED : A BRIEF GUIDE TO THE STANDARD OBJECT MODELING LANGUAGE. Second edition. Martin Fowler, Kendall Scott. Addison Wesley Longman inc. 2000.

UML 2 FOR DUMMIES. Michael Jesse Chonoles, James A. Schardt. Hungry Minds. 2003.

THE UNIFIED MODELING LANGUAGE USER GUIDE. Grady Booch, James Rumbaugh, Ivar Jacobson. Addison Wesley Longman Inc. 1999.

THE RATIONAL UNIFIED PROCESS AN INTRODUCTION. Second Edition. Krutchten Philippe. Addison Wesley. 2000.

EL LENGUAJE UNIFICADO DE MODELADO. MANUAL DE REFERENCIA. Grady Booch, James Rumbaugh, Ivar Jacobson. Pearson Educación S.A. Madrid 2000.

GUIA DE DESARROLLO DELPHI 5. Steve Teixeira, Xavier Pacheco. Pearson Education. Madrid. 2000.

MASTERING DELPHI 6. Marco Cantu. Sybex inc. Alameda, CA. 2001.

OPENGL SUPERBIBLE. Second Edition. Richard S. Wright Jr, Michael Sweet. Wait Group. United States of America. 2000.

DELPHI DEVELOPER'S GUIDE TO OPENGL. Jon Jacobs. Wordware publishing Inc. 1999.

REDES DE COMPUTADORAS. Tercera Edición. Andrew S Tanenbaum. Prentice Hall. 1998.

TCP / IP ARQUITECTURA, PROTOCOLO E IMPLEMENTACION CON Ipv6 Y SEGURIDAD IP. Sidnie Feit. McGraw Hill. 1998.

COMUNICACION Y REDES DE COMPUTADORAS. Quinta edición. William Stallings. Prentice Hall. 1997.

THE ART OF COMPUTER GAME DESIGN. Chris Crawford. Electronic version. Sue Peabody. Washington University Vancouver. 1997.

TRICKS OF WINDOWS GAME PROGRAMMING GURUS. André Lamothe. SAMS United States of America. 1999.

SOFTWARE ENGINEERING AND COMPUTER GAMES. Rudy Rucker. Addison Wesley. 2002.

3DS MAX 4. CURSO PRACTICO. Castell Cebolla. Alfaomega Grupo Editor SA. Ciudad de Mexico. 2002.

3D STUDIO MAX 3 FUNDAMENTALS. Michael Todd Peterson. New Riders Publishing. 1999.

3DS MAX 4 BIBLE. Kelly L. Murdock. Hungry Minds, inc. 2001.

MATRIX COMPUTATIONS. Gene H. Golub, Charles F. Van Loan. The Johns Hopkins University Press. United States of America. 1993.