MODELO FORMAL PARA ESPECIFICACION E IMPLEMENTACION DE HIPERHISTORIAS

Profesor : Gustavo Rossi

Alumno : Guillermo Capelli

INTRODUCCION

Con el avance de la tecnología en lo que respecta a ambientes multimediales, cambió la forma de procesar y acceder a la información, ahora, la integración de varios medios, con el objetivo de transmitir conocimiento, hace que la forma en que este se adquiera sea diferente a lo que era antes de estos avances.

La nueva tecnología lleva a que la forma de adquirir conocimiento tenga un mayor grado de interactividad, es decir, que el alumno manipule el conocimiento y experimente con las posibilidades de su aplicación en diferentes campos y con esto saque conclusiones y genere en si nuevo conocimiento a partir de la experiencia.

La educación en un ambiente multimedial se hace mas personalizada e individual, por lo que la incidencia de las reacciones del alumno ante el conocimiento que adquiere toman mas importancia en el proceso de aprendizaje.

Este modo de educación adquiere especial importancia, por ejemplo, para los niños mas pequeños que no saben leer, por lo que la manera de representar el conocimiento de una manera visual e interactiva, en la cual el texto escrito no sea lo relevante de la información que se intenta transmitir, hace que la información sea accesible también a ellos.

Las hiperhistorias son una forma de representar la información en un ambiente multimedial en la cual la existencia en ellas de ambientes dentro de los cuales hay objetos y personajes, alguno de los cuales el niño puede manejar, haciendo que realicen acciones y, de esta manera, permitiéndole interactuar con un entorno que le resulte familiar, como ser una casa, un barrio, etc., hace que este pueda adquirir conocimientos derivados de la experiencia de realizar las acciones que debe llevar a cabo.

La manera de implementar estas hiperhistorias lleva a hacer una recorrida por los distintos ambientes de autoría de sistemas de multimedia e hipermedia y se llega a la conclusión de que no hay ninguna herramienta que reúna todas las condiciones necesarias para llevar a cabo estas aplicaciones de una manera relativamente sencilla.

La forma en que en estas aplicaciones de autoría se representa la información, la forma de especificar conceptos como navegación y comportamiento autónomo de entidades, sincronización temporal, concurrencia y otros conceptos que se verán mas adelante, hace que sea difícil implementar en ellas una hiperhistoria que incluyan las características mencionadas anteriormente.

Para ello se definió un formalismo que soporta estas características y otras que veremos mas adelante, lo que permite definir la hiperhistoria de una manera, en principio, textual y luego generar un interface con alguna herramienta para generar aplicaciones de multimedia adecuada al estilo de representación de la interface para la hiperhistoria que se quiere llevar a cabo.

A partir de la descripción textual, en el formalismo, de la hiperhistoria, se generará una aplicación ejecutable, que en adelante llamaremos aplicación, la cual conectada con una interface hecha para esa aplicación, que en adelante llamaremos interface, formarán una hiperhistoria en la cual el comportamiento de los objetos y toda la sincronización temporal estará en la aplicación, generada a partir del modelo y la representación visual e interacción con el alumno estará en la interface.

La interface se implementará en una herramienta para generar aplicaciones multimediales, la elección de esta herramienta se hará de acuerdo al estilo de la interface, su complejidad y el mejor manejo de los medios usados.

La aplicación generada a partir del modelo y la interface implementada para esa aplicación se comunicarán entre sí para pasar, de la aplicación a la interface, la información de cambios de estado de los objetos de la hiperhistoria implementados en la aplicación, que la interface debe representar de alguna manera coherente a su estilo, y de la interface a la aplicación, la información de las acciones que el alumno lleva a cabo, en forma de eventos interpretables por la aplicación. La interface es la encargada de interpretar de alguna manera las acciones del usuario y traducirlas a un formato de comunicación predefinido entre y ésta la aplicación.

De esta manera se separa, en la hiperhistoria que se construye, el contenido de esta, el cual se encapsula en la aplicación, de la representación visual de la misma, lo que hace que una misma hiperhistoria pueda ser representada por distintas interfaces de acuerdo a las necesidades físicas de los alumnos que las usen, como ser niños ciegos, sordos, con problemas motrices, etc. , con lo que no es necesario, de esta manera, reimplementar todo el comportamiento de objetos y entidades de la hiperhistoria al tener la necesidad de cambiar el estilo de interface.

Para la generación de la aplicación se implementará un traductor que a partir de una descripción formal de la hiperhistoria en el modelo que se describirá, generará código en lenguaje Pascal que luego, al ser compilado, dará por resultado un programa ejecutable que es la aplicación mencionada anteriormente.

Más adelante, en una herramienta de autoría adecuada, se generara la interface que corresponda a la aplicación implementada, siguiendo ciertos lineamientos para realizar la comunicación con ésta.

En el presente documento veremos primero una reseña acerca de multimedia en educación, en la cual haremos una revisión de sus ventajas y desventajas en lo que respecta a la transmisión de conocimiento.

Luego veremos una descripción mas extensa y detallada de lo que son las hiperhistorias y su posible utilidad en la educación.

Posteriormente se describirá el modelo diseñado para realizarlas y la manera en que este se implementó, veremos una descripción de las herramientas desarrolladas para escribir y generar la hiperhistoria .

Por último veremos un ejemplo de aplicación de este modelo y su implementación en una hiperhistoria sencilla en la que se ven ejemplos de todas las características definidas para ellas y las posibles extensiones que se le puede hacer al modelo y los posibles trabajos y herramientas a desarrollar para la continuación del proyecto.

CAPITULO I

MULTIMEDIA EN EDUCACION.

1.1 - Introducción.

Un sistema de educación multimedial no sólo debe ser un ambiente en el cual se representa información, la cual es adquirida por el estudiante, o solo ser un medio de expresión de un profesor o maestro, por el contrario, estos ambientes deben brindar la posibilidad de que el estudiante acceda a la información, que interactúe con ella, que la revise y trate de interpretarla y de construir nuevo conocimiento.

Con la tecnología multimedial queda atrás la idea del alumno que adquiere información, que es brindada por un profesor a través de variados medios, durante un periodo y, luego, es examinado por este u otros profesores para conocer el grado de aprendizaje, sino que la evaluación esta mezclada con la adquisición del conocimiento de manera interactiva.

Por medio de un programa de computación, una situación puede ser descripta a través de texto, gráficos, animaciones, figuras, fotos, videos, sonido. La realidad puede ser mostrada con un video o película, mientras los detalles pueden ser explicados a través de gráficos de computadora. Fenómenos que normalmente no pueden ser observados por el ojo humano pueden ser visualizados a través de la manipulación del tiempo, la velocidad, los gráficos y las animaciones generadas por la computadora. [Haugen 92]

Las estrategias educacionales actuales, con un profesor al frente de un grupo de estudiantes, son universales, no importando de que país o cultura se este hablando. Por otra parte, las clases dictadas usando una computadora son, en estrategia educativa, las mismas que las dictadas sin computadoras y con la misma estrategia que cuando no existía la computadora.

Esto resalta que las computadoras son usadas de manera poco adecuada en educación, desaprovechando todo el potencial que ellas brindan en lo que respecta al acceso de la información y la interactividad con el estudiante.

En la manera actual de educar, se espera que cada estudiante acceda a la información y la asimile de la misma forma, o sea, brindándole a todos los alumnos la misma información representada de la misma manera.

Esto hace que ciertos estudiantes adquieran e interpreten el conocimiento brindado y otros no lo hagan de manera óptima. Esto se debe, en gran parte, a que a estos últimos estudiantes no se les brinda una manera de representación de la información adecuada a su manera de razonar y adquirir conocimiento. Si un estudiante no responde como debiera al conocimiento que se le intenta impartir, se debe, en mayor medida, a que este conocimiento no se le esta brindando de la mejor manera posible para ese alumno.

Las estrategias para brindar conocimientos son muchas y pueden variar enormemente, pueden usar diferentes medios para comunicar la información, tener diferentes direcciones pedagógicas o pueden ser llevadas a cabo de diferente manera según las dificultades individuales de cada estudiante. [Bork 92]

Las necesidades especiales se ven mas resaltadas en caso de alumnos discapacitados, los cuales necesitan, como apoyo a educación, herramientas cuidadosamente diseñadas que puedan motivar su desarrollo, ayudando estas herramientas a combatir su discapacidad o a buscar un medio de que el alumno pueda aprender a pesar de ella.

1.2 - Diferentes necesidades de aprendizaje.

No todos los alumnos aprenden de la misma manera, la forma en que un alumno interpreta un concepto o saca conclusiones de algún conocimiento que se le imparte varia según el alumno, por lo tanto la forma en que estos conocimientos se le brindan al alumno debería adaptarse, idealmente, a las necesidades cognitivas de cada uno.

Llevar esto a la práctica con los medios tradicionales de enseñanza es impracticable por el costo que esto significaría, ya que seria necesaria una educación mucho mas personalizada, en donde la relación entre el profesor y el alumno sea mas fluida, con el objeto de que este determine para cada alumno el mejor método posible de aprendizaje que este necesitaría y lo aplique, además de realizar un seguimiento mas cercano del progreso del alumno.

La mejor manera de incorporar conocimiento es diferente para cada alumno, algunos aprenden con mas facilidad si escuchan los conceptos, otros comprenden mejor la información si se les presenta de manera visual, o ambas, etc., no hay una forma de presentar la información que se pueda decir que satisfaga todas las necesidades, por lo tanto, aplicar los medios tradicionales,

como ser libros, videos, grabaciones, etc. en forma personalizada, según las necesidades de cada alumno, sería algo muy engorroso.

La presentación de la información de diferente manera, según las necesidades del alumno sería muy difícil sin la asistencia de un soporte multimedial que ayude a su clasificación, ordenamiento y que permita el manejo de diferentes medios, como ser animaciones, audio, video, etc. en forma coordinada e interactiva. [Bork 92]

En el caso de los estudiantes con ciertas discapacidades mentales, los cuales tienen dificultades especiales para acceder a la información, el uso de aplicaciones multimediales, diseñadas especialmente según la dificultad de cada alumno, para apoyar su educación, puede favorecer al desarrollo en ellos de estructuras mentales que lleven a un progreso cognitivo del alumno, ya que la información se les puede brindar por medios y formas no convencionales, los que hacen que el alumno comprenda y asimile conceptos difíciles de transmitir por medios tradicionales.

Si el ambiente educativo es capaz de adaptar la información y las actividades de enseñanza a las habilidades del alumno, la satisfacción generada por el estudiante puede alterar positivamente el proceso de aprendizaje. Sin embargo, a menos que podamos entender las operaciones mentales que el alumno emplea, es difícil de imaginar las situaciones que pueden afectar la actividad mental del alumno. Algunos estudios dicen que el aprendizaje puede ser positivamente afectado si los estudiantes son conscientes de lo que necesitan aprender. Los mismos estudios revelan que los consejos dados al alumno en una situación de enseñanza individualizada basada en una evaluación permanente del mismo pueden afectar positivamente el proceso de aprendizaje haciendo que mejore la adquisición de conocimiento.

La individualización del aprendizaje en un ambiente interactivo debe lograrse a través del desarrollo de un diseño flexible que sea sensitivo a las varias formas de comunicación e intercambio de información entre hombres y máquinas. Este también debe ser sensible a los cambios que necesiten los alumnos, el sistema deberá adaptarse en si mismo a las necesidades de los alumnos. [Goetzfried and Hannafin, 1985 ; Ross et al., 1984 ; Tennyson and Bultrey, 1980]

1.3 - Motivación.

Otro factor en el aprendizaje es que lo que cada alumno estudia debe resultarle interesante, debe inducirlo a querer estudiar mas, el aprender no tiene que ser una tarea ardua, de tal manera que necesite estímulos externos que lo obliguen a aprender. Este ultimo método es el mas común en la educación actual.

La representación correcta de la información que se quiere transmitir, según el tipo de alumno al cual se oriente, puede hacer que este no piense que es obligado a realizar la instrucción y la haga con una mayor predisposición lo cual lleve a que se mejore el grado de adquisición de conocimiento. [Bork 92]

Esto tiene especial importancia en la educación de niños, a los cuales se los puede inducir a la adquisición de conocimiento mediante la presentación de este en una forma que a ellos le resulte atractiva y entretenida, lo cual puede llevar a que, manipulando este conocimiento en forma adecuada, el niño aprenda a partir de la experiencia, esto toma especial importancia en caso de niños con ciertas discapacidades mentales, a los cuales se les debe enseñar con métodos en los cuales ellos, además de aprender y ejercitar sus capacidades, se entretengan, como para que la enseñanza sea útil.

Los ambientes multimediales claramente permiten introducir estas características, mediante la utilización de los diferentes medios en forma sincronizada, se pueden generar aplicaciones que permitan representar el conocimiento de una manera no usual, lo que permitiría concentrar la atención del alumno, esta también se ve aumentada por la característica de interactividad de la aplicación, lo que hace que este no sea un mero espectador de una película.

El proceso cognitivo del alumno juega un rol fundamental y esta relacionado, sobre todo, a la búsqueda y procesamiento de la información. La habilidad de navegar y explorar un ambiente rico en información, fomenta la búsqueda y clasificación de la información, y la integración y observación desde varios puntos de vista de un concepto mediante la estimulación de la comprensión de un fenómeno a través de representaciones múltiples, gráficas, visuales, aurales o escritas. [Giardina 92]

Para obtener efectos duraderos y positivos de la instrucción, es esencial mantener el interés del alumno, ya sea mediante elementos excitantes en forma de juegos, por eventos inesperados o, en una manera mas relevante, mediante la presentación del contenido en una manera mas atrayente. Los efectos del uso de la multimedia prestan una buena ayuda a este propósito, pero se debe ser consciente que cada elemento debe tener una función clara en el proceso de aprendizaje, no solo debe ser algo que atraiga la atención del alumno. En este ultimo caso lo que tendríamos seria mas una distracción que una ayuda para entender o memorizar el conocimiento que se intenta transmitir. La multimedia puede agregar una nueva dimensión a la programación de estas herramientas pero no debe ser usada de una manera que lleve al extravío del alumno. [Haugen 92]

1.4 - Interacción.

La forma en que el proceso de aprendizaje puede ser llevada a cabo varia según el alumno pueda participar activamente o no en este proceso. La forma en que el estudiante accede a la información puede ser mas o menos interactiva, por ejemplo, un alumno escuchando una clase dada por un profesor es el extremo de menor interacción en el acceso a la información, el alumno es un simple espectador el cual escucha el conocimiento que el profesor le imparte y trata de asimilarlo, para que luego de un tiempo, se lo evalúe para determinar cuanto de ese conocimiento fue aprendido.

Un ejemplo de aprendizaje con alta interacción es el de un aprendiz que realiza una tarea guiado por un maestro, en este caso es el alumno quien realiza la actividad, cualquier situación errónea es puesta de manifiesto y corregida por el maestro, esto lleva a que el proceso de enseñanza y evaluación se haga en forma conjunta y que se pueda determinar en cada momento el grado de asimilación, por parte del alumno, del conocimiento que se le brinda. En este último caso el alumno participa activamente en todos los aspectos del proceso de aprendizaje.

Entre estos dos extremos ejemplificados existe una amplia gama de posibilidades, de cada método de enseñanza o estrategia educativa se puede decir si es o no interactiva, pero lo importante no es esto sino cuánto de interactivo tiene el método, lo cual no significa medir la interactividad en una escala de valores sino considerar aspectos como el grado y la calidad de la interacción.

El grado de interacción en mas alto cuando la intervención del alumno en el proceso de aprendizaje es mas frecuente, pero esto tiene ciertos limites, ya que es necesario cierto tiempo para que este razone acerca del problema. Si la participación del alumno en el proceso hace que este actúe sin tener tiempo suficiente para pensar en lo que esta haciendo, los beneficios de la interacción se diluyen. Por otra parte si el grado de interacción es muy pequeño, el interés del alumno disminuye.

El otro aspecto es mas complicado, determinar si la interacción en determinado método de enseñanza es de buena o mala calidad es algo que solo se puede basar en la intuición de maestros experimentados mas que en estudios experimentales. De todos modos, se pueden seguir ciertas líneas que lleven a una mejor calidad de la interacción, evitando problemas de información excesiva, perdida de orientación, heterogeneidad de la interacción, etc.

Las herramientas basadas en sistemas de multimediales de computación hacen que se pueda perfeccionar este aspecto de la enseñanza, ya que el alumno interactúa con el sistema y este, siempre que este diseñado siguiendo una línea pedagógica clara, responderá de manera adecuada, resaltando los errores, sugiriéndole alternativas, etc. [Bork 92]

1.5 - Desarrollo de sistemas multimediales.

Ultimamente el desarrollo tecnológico ha permitido la construcción de sistemas multimediales, efectuando la combinación de fuentes de información tan variadas como pueden ser audio, video, animaciones, texto, etc. Estos sistemas, organizados de manera similar a los hipertextos, esto es, una base de información organizada en nodos vinculados entre sí por links, son usados como un modo de transmitir conocimiento de una manera no tradicional.

Desarrollos de estas características son utilizados en áreas diversas como pueden ser educación, negocios, publicidad, etc., ya que mejoran aspectos de la comunicación hombre-máquina y del acceso a la información.

Una de las dificultades en estos desarrollos consiste en la gran cantidad de información que estos deben manipular, debido al tamaño de datos como las imágenes, las grabaciones de audio o el video, y el hecho de que esta no es homogénea, es decir el modo de almacenamiento varia según el tipo de información, la forma de codificarla o comprimirla, etc.

También se deben considerar otros problemas en el desarrollo de estos sistemas como ser el "overhead" cognitivo, o sea, que el sistema exponga al lector o usuario mucha mas información de la que este pueda procesar, y el de desorientación, es decir, que la exploración lleve al lector por caminos irrelevantes y no deseados, impidiéndole que este pueda obtener la información necesaria.

Por otra parte, el modelo de hipermedia representa una alternativa muy conveniente para construir sistemas multimediales ya que presenta no sólo una metáfora sencilla y eficiente, sino además porque el modelo matemático subyacente permite formalizar dicha construcción, así como extender sus facilidades, una hipermedia no solo puede accederse por navegación sino además por consultas, al estilo de las bases de datos.

Ultimamente se han desarrollado una serie muy variada de sistemas de soporte para la construcción de aplicaciones multimediales e hipermediales, entre los cuales pueden mencionarse Hypercard y Supercard para ambientes Apple y Toolbook y

Guide sobre Windows en Pc's compatibles. La disponibilidad de este tipo de software ha facilitado la construcción de este tipo de aplicaciones en áreas diversas.

La utilización de esta tecnología como una herramienta de apoyo a la utilización de la computadora para efectos de aprendizaje comienza a ser reportada en el ámbito internacional desde hace poco tiempo. Esta tecnología brinda facilidades de acceso a una diversidad de canales de comunicación a personas con problemas de comunicación con el entorno, en especial a personas discapacitadas.

El desarrollo de sistemas multimediales lleva a la necesidad de resolver problemas informáticos de relativa complejidad, como por ejemplo, la necesidad de separar el contenido abstracto de los sistemas de la interface de los mismos.

A pesar de los signos alentadores, el diseño de los sistemas multimediales para educación es aun algo mas intuitivo que científico. Ahora, ciertos problemas sistemáticos son aparentes, ejemplo, la falta de cohesión que facilite la integración eficiente de la información dentro de una estructura cognitiva coherente. El mensaje educacional interactivo no es diseñado de manera secuencial pero es basado en la exploración llevada a cabo por los intereses del alumno. Así, no es diseñado de acuerdo a una óptima secuencia de aprendizaje, lo cual puede llevar a algún desperdicio de energía cognitiva. [Glardina 92].

CAPITULO II

HIPERHISTORIAS

2.1 - ¿Que es una hiperhistoria?

Tradicionalmente la concepción de historias, involucra la descripción de una sucesión de hechos, en los cuales el alumno se limita a percibirla a través de un texto escrito, la observación de un film, o por ejemplo escuchar una narración grabada. Si bien en estas alternativas tradicionales se puede simular cierto grado de interacción, siempre esta limitado al conjunto de herramientas que provea el ambiente de la historia (por ejemplo: tablas de decisión, anotadores para llevar la cuenta de objetos que posee el protagonista, etc.).

La simulación de eventos de tiempo real es mas difícil todavía, pues este debería manipular algún tipo de reloj, etc. Este tipo de simulaciones pone una complejidad y carga cognitiva en el alumno demasiado alta, haciendo engorroso el uso de un sistema así.

Si nosotros deseamos crear un ambiente donde el alumno sea protagonista real de la historia, con alto grado de interactividad y con representación adecuada de los eventos en el tiempo, deberemos proveer una herramienta que gestione esta funcionalidad de manera automática, de tal manera de liberar al alumno de tareas que no son directamente relevantes con su objetivo.

Una hiperhistoria es un conjunto de ambientes (que llamaremos *contextos*), los cuales por medio de un conjunto de *links*, forman una estructura de navegación, éstos links serán los encargados de conectar entre sí a los ambientes.

El alumno en esta estructura, recorrerá los ambientes por medio de la manipulación de un personaje, que llamaremos el *protagonista* de la historia y que representara la conexión entre el lector y el sistema. Estos ambientes estarán también poblados de *entidades* (por ejemplo, sillas, mesas, luces, animales, automóviles, etc.), los cuales tendrán su propio comportamiento, e inclusive existirán otros personajes.

La suma de estas entidades, navegando por los ambientes, interactuando entre si y el paso del tiempo, creara una sucesión de eventos los cuales desencadenarán hechos y alternativas diferentes en función de la acción del protagonista.

Esta seria la descripción de lo que es una *hiperhistoria*. Esta definición de hiperhistoria permite que la apreciación de hechos por parte del alumno sea distinta y sensible a la actividad del protagonista y otras entidades.

En este ambiente es claro que la idea de guión tal como aparece en un relato escrito es totalmente diferente, pues no existe la descripción monolítica de la historia, sino que ella esta distribuida en cada una de la entidades que pueblan los ambientes, las cuales al comportarse como se lo especifique, mas la acción del lector sobre el o los personajes que controle, los cuales influirán sobre los sucesos en la hiperhistoria, hará que se arme una historia particular, la cual puede ser diferente si diferentes lectores usan la misma hiperhistoria de diferente manera.

El curso de la hiperhistoria no estará fijado de antemano, este variará según las acciones del lector y lo llevará por caminos diferentes dependiendo de estas acciones.

Hay que tener en cuenta que separar los ambientes de las entidades permite que, en un mismo ambiente, se puedan recrear a priori dos historias enteramente diferentes. Es la misma idea que aparece en las series de televisión, donde las cosas ocurren siempre en el mismo lugar, pero siempre son historias diferentes. Debe quedar en claro que esto podría ocurrir al utilizar los mismos ambientes, pero llenándolos en cada caso con objetos diferentes o simplemente haciendo que el personaje ejecute diferentes secuencias de acciones.

2.2 - Hiperhistorias en educación.

La forma de presentar información a través de historias interactivas hace que esta pueda ser mas fácil de transmitir a los niños, ya que a estos les gustan las historias y las recuerdan con facilidad. Cuando un niño "lee" una historia interactiva, puede identificar, tomar y usar los conocimientos relevantes para la resolución de un problema, ya que tiene un acceso a la información rápido y flexible. Las hiperhistorias motivan el aprendizaje por parte de los niños proveyendo un contexto rico en información.

Las historias están ancladas en episodios acerca de incidentes específicos, explotando y utilizando la imaginación de los niños, generándose una situación de gran cohesión, en donde el aprendizaje y motivación están relacionados.

El alumno no sólo se motiva sino que se involucra con la historia, al ocurrir esto, tiene que identificar, recuperar y utilizar datos relevantes para resolver desafíos, teniendo rápidos y flexibles accesos a secuencias de la historia. En su interacción con la historia el alumno puede examinarla en orden cronológico, en cronología inversa o en una sección intrincada de acceso azaroso.

Las hiperhistorias se construirán para entretener, poner a prueba, interactuar y motivar a los alumnos dándole control sobre los personajes y objetos para formar ideas y desarrollar estrategias con la hipótesis implícita de promover el desarrollo intelectual del alumno.

Las hiperhistorias facilitan la navegación, cuando el alumno explora el hipermedio y cuando trata de recordar la información, permiten ubicar la información en un contexto significativo en lugar de examinarlo por separado, lo cual es importante para aprender. El alumno no puede entender el significado de un concepto sólo por su definición, debe ver muchos ejemplos de cómo es utilizado ese concepto en un determinado contexto antes de que lo pueda comprender. El entendimiento contextual y estructural son fundamentales para aprender y comprender.

Mediante el uso de hiperhistorias se tratará de explotar la noción de espacio, posición, secuencia y extensión en tiempo, como también mejorar el desarrollo de estructuras cognitivas de alto orden.

2.3 - Hiperhistorias y discapacidad.

Hemos dicho que las aplicaciones multimediales, diseñadas especialmente, pueden ayudar a la educación de niños con ciertas discapacidades, en especial, concentraremos nuestra atención en el desarrollo en los alumnos de las relaciones espacio-temporales y de lateralidad.

Existe un grupo critico de la población infantil latinoamericana que posee significativas deficiencias en este aspecto, por otra parte las personas con una discapacidad física y sensorial, como pueden ser las personas sordas, ciegas, mudas, con parálisis cerebral, entre otras, presentan alteraciones en sus estructuras mentales que les impiden desarrollar adecuadamente este tipo de relaciones cognitivas.

A continuación veremos una descripción de estos problemas y como las hiperhistorias pueden ayudar, en alguna medida, a paliar estos defectos.

2.3.1 - Relaciones espaciales.

Estas relaciones se refieren al desarrollo cognitivo que permite :

- Ubicarse en el espacio circundante.
- Comprender las nociones de atrás, adelante, arriba, abajo.
- Determinar la posición relativa entre objetos.
- Orientarse a si mismo y a los otros objetos en forma estática y en movimiento.
- Organización del la persona frente al mundo que lo rodea, ubicación de los objetos en posiciones relativas a otros, movimiento, dirección.

Estas relaciones se estructuran de la siguiente manera :

1. Conocimiento de las nociones, el alumno debe aprender a situarse y situar los objetos, tiene que percibir las distintas formas, cantidades, tamaños.
2. Orientación espacial, luego de que se poseen las nociones espaciales mencionadas en el punto anterior, se debe enseñar al alumno a orientarse, esto es, a girar, avanzar, retroceder, ir hacia abajo o arriba. De esta manera aprende a percibir en las discriminaciones visuales lo que se encuentra orientado en la misma dirección y sentido, adquiriendo dirección gráfica.
3. Organización espacial, es la integración de los dos primeros puntos, lo que permite al alumno disponer del espacio. Así, en una primera etapa el alumno puede, por ejemplo, poner una figura humana frente a una casa, luego orientarla hacia un lado y mas tarde crear un pueblo y hacer que la figura haga un recorrido sin pasar dos veces por el mismo lugar.

4. Comprensión de las relaciones espaciales, se basa en el raciocinio a partir de situaciones espaciales bien precisas. Las personas con problemas de percepción espacial no distinguen una "b" de una "d" una "p" de una "q", el numero "26" del "62", no distingue el "arriba" del "abajo", confunde la "b" con la "p" o la "n" con la "u", además de presentar escasa memoria espacial.

Un sistema multimedial que tenga como objetivo el reforzamiento o la construcción de estas estructuras cognitivas deberían tener en cuenta, entre otros, los siguientes puntos :

- Fortalacer la discriminación visual.
- Favorecer el uso de topologías.
- Favorecer la idea de orientación de objetivos.
- Favorecer el conocimiento de los términos, insistiendo en las nociones de "arriba" y "abajo".
- Trabajar con figuras que involucren el recorrido de trayectos.
- Trabajar con situaciones que requieran aplicar la memoria para facilitar la resolución de problemas, por ejemplo, recordar dónde estaba ubicada una cosa para recuperarla rápidamente.

2.3.2 - Relaciones temporales.

Estas relaciones se refieren a la ubicación de la persona con respecto al tiempo e implican :

- Forma como la persona se sitúa en el tiempo, ejemplo, ayer, hoy, mañana.
- Capacidad de situarse en función de una sucesión de acontecimientos.
- Comprensión de la duración de intervalos, nociones de tiempos largos (meses), cortos (horas, minutos), noción de regular e irregular (aceleración, frenada), noción de cadencia rápida y lenta (diferencia entre correr y caminar).
- Interpretación de la renovación cíclica de ciertos periodos, ejemplo, días de la semana, meses, estaciones.
- Comprensión del carácter irreversible del tiempo, del tiempo subjetivo creado por una persona. Este tiempo varia de acuerdo a las personas y a las actividades que desarrollen en ese momento y el tiempo objetivo, es decir, que siempre será el mismo, por ejemplo, una hora siempre tendrá sesenta minutos.

La estructuración de este tipo de relaciones se lleva a cabo siguiendo las siguientes etapas :

1. Orden y sucesión, la persona percibe y memoriza lo que pasa antes, ahora, después..., en que orden fueron realizados las acciones, que fue realizado primero y que ultimo, a clasificar cosas según un orden lógico o cronología.
2. Duración de los intervalos, la persona percibe lo que pasa rápido y lo que tarda mas, la noción de tiempo transcurrido, la diferencia entre una hora y un día, etc.
3. Renovación cíclica de ciertos periodos, asociar acciones o situaciones a ciertas actividades o materiales, una lampara se asocia con la noche, un abrigo con el frío, etc.
4. Ritmo, la persona incorpora la noción de orden, sucesión, duración, alternancia. Comienza con su propio ritmo y, gradualmente, es llevado a acompañar con mas precisión un ritmo establecido.

La persona con problemas en las relaciones temporales tiene confusión en la sucesión de los acontecimientos, en percibir intervalos, en ordenar cosas y hechos. No puede manejar dos elementos de una sílaba o los dígitos de un numero ; no puede percibir las dificultades al reconstruir una frase cuyas palabras estén desordenadas. En matemáticas puede tener dificultades en cálculos, ya que para ello se requiere tener puntos de referencia, colocar los números correctamente, saber lo que viene antes y después, etc.

Algunas de las guías a tener en cuanta para la construcción de aplicaciones que ayuden al desarrollo de las relaciones temporales serian las siguientes :

- Que posean situaciones que refuercen una concretización del espacio, retomando las nociones de "frente", "atrás", "arriba', "abajo".

- Que se contemple la resolución de problemas de orden y que favorezcan la idea de sucesión en el tiempo, por ejemplo, el balde que se llena de agua si el grifo se dejo abierto.
- Aspectos que refuercen las nociones de distancia, "entre", "contra", "al lado"
- Situaciones que contemplen el uso de diagramas de topología.
- Que se incluyan situaciones que utilicen el concepto de duración de intervalos.

2.3.3 - Lateralidad.

Idea que el niño tiene de si mismo, de la formación de su esquema corporal, que contribuye para determinar la estructuración espacial, ya que en la medida que el niño percibe su cuerpo, percibe el de los otros. La lateralidad implica :

- El reconocimiento de dos lados.
- El reconocimiento de su derecha e izquierda.
- El reconocimiento de la derecha e izquierda de los otros y de los objetos de su mundo en relación a él.

Una persona cuya lateralidad no este bien definida, no percibe la diferencia entre su lado dominante y el otro, no distingue su derecha e izquierda y es incapaz de seguir la dirección gráfica (lectura comenzando desde la izquierda).

Las características de un sistema que permita la estimulación del desarrollo de la lateralidad en una persona serian, entre otras, que ponga énfasis en situaciones que estimulen la distinción entre el lado derecho y el izquierdo, por ejemplo, cuando un personaje deba ser movido hacia una ubicación particular.

2.3.4 - Rol de las hiperhistorias.

Los descriptos son algunos problemas que se pueden atacar utilizando hiperhistorias, ya que estas, correctamente diseñadas, pueden estimular :

- La percepción espacial.
- La posibilidad de descentralizarse.
- La representación del tiempo, del movimiento, de la velocidad, del espacio.
- La toma de decisiones.
- La estimulación visual.
- La motivación.
- El reconocimiento del espacio inmediato y global.
- La necesidad de anticipar y prever estrategias mas adecuadas a un problema.
- La Integración social a través de un ambiente rico en información.
- El esquema corporal.

De acuerdo a lo descripto anteriormente, mediante algunos tipos de hiperhistorias se puede contribuir a la formación de estructuras mentales de ciertas personas con deficiencias en su desarrollo.

De todos modos, para atacar estos problemas no alcanza con una hiperhistoria en la cual el funcionamiento es correcto, también la interface de esta debe seguir ciertas pautas que llevarían a una interface mas amigable y en la cual los conceptos, que en ella y en la historia se quieren representar, sean comprensibles y claros para los usuarios a los cuales estas aplicaciones van dirigidas. En la próxima sección veremos ciertos aspectos con respecto a esto último.

2.4 - Guías para la construcción de la interface.

Existen ciertas pautas que se deben seguir para la implementación de las hiperhistorias, en especial, la manera en que esta será presentada, es decir, como será su interface, como será la interacción con el lector, como se indicaran las acciones que este quiera realizar, como se vera, etc. Aquí haremos una descripción de algunas de esas pautas.

2.4.1 - Manipulación del protagonista.

El lector debería poder tomar el control solo de a un personaje por vez, en caso de que se le de la posibilidad de controlar a varios, seria deseable que con cada personaje se identifiquen un conjunto posible de acciones que este pueda llevar a cabo.

Seria deseable que haya una indicación acerca del estado de cada personaje según la circunstancias de la historia, como por ejemplo, si el personaje esta confundido, cansado, etc., de manera de hacer notar las circunstancias por las cuales este deba cambiar su comportamiento.

Se debe crear una sensación de movimiento lo mas real posible, con la finalidad de darle mas realismo a la historia y hacerla mas atractiva, lo que llevaría a una concentración mayor de la atención por parte del lector.

2.4.2 - Tipos de movimiento del protagonista.

Se debería poseer a nivel de interface un modo por el cual sólo se permitiría al personaje que controla el lector navegar libremente a través de los contextos, esto seria como una introducción a la hiperhistoria, también seria deseable hacer una especie de presentación de los personajes, en ella se enumerarían sus habilidades, debilidades, etc., esto seria preferente hacerlo mediante comentarios sonoros acompañados de la imagen de cada personaje referido en ese comentario.

Se podrían tener distintos niveles que lleven al aprendizaje de distintas destrezas por parte del lector, en un primer nivel se tendría un mayor control acerca de los movimientos del personaje, esto se puede hacer mediante la técnica de "drag and drop". En niveles sucesivos y una vez que el lector adquirió la habilidad de desplazarse y el concepto de "moverse en el espacio", se podría utilizar un simple click del mouse en una dirección, permitiendo una abstracción mayor del concepto y un menor control "físico" del protagonista.

2.4.3 - Como mostrar las acciones posibles del protagonista.

Se debe proveer un modo, en la interface, de que el lector le indique al protagonista lo que quiere que haga esto presenta una dificultad que es proporcional a las habilidades que el protagonista tenga, es decir, si este solo puede caminar, bastaría que el lector solo hiciera un click del mouse en el lugar hacia el cual desea ir, pero si el numero de habilidades que posee el protagonista es mayor, se presenta el problema de elegir el modo en el cual el lector podrá especificar que acción desea que el protagonista haga. Esto debe hacerse de una manera que sea consistente con el estilo de interface que se desea implementar y además que sea sencillo, de manera que no resulte una dificultad el mero hecho de decirle al protagonista que haga tal o cual cosa.

La acción principal que debe estar presente entre las habilidades del protagonista es la navegación, el lector debe poder indicarle de una manera sencilla si quiere desplazarse, si quiere cambiar de contexto, etc. Luego, el resto de las acciones se referirán a la interacción del protagonista con los objetos que lo rodean en el contexto en que se encuentre, por ejemplo, la activación de un objeto, asociar un objeto a otro, tomar un objeto, soltarlo, preguntar por un objeto, etc. Estas acciones permiten que el lector realice combinaciones en busca de algún resultado.

Cada uno de los métodos de llevar a cabo esta tarea tendrá sus ventajas y desventajas y de acuerdo a las necesidades que condicionen la implementación de la interface se deberá elegir alguno de estos estilos.

El estilo mas común seria el de tener una barra de comandos, esta tiene la ventaja de que es fácil mostrar en ella acciones y es sencillo seleccionarlas, además, como aparecen visibles en todo momento, no es necesario recordar cuales de esas acciones estarían disponibles. La desventaja es que se pierde espacio útil en pantalla y se deja poca flexibilidad para añadir nuevas acciones.

Otra forma seria realizar un doble click del mouse sobre el protagonista y que apareciese un menú tipo pop-up o algo por el estilo que permitiese seleccionar la acción deseada, esto tiene la ventaja de que se puede aprovechar toda la pantalla para el dibujo de los contextos pero, cada vez que se quiere saber si una acción esta disponible se debe hacer aparecer el menú, el cual taparía parte del dibujo del contexto.

Una manera que seria mas consistente seria la de asociar a cada parte del cuerpo del protagonista una acción, como por ejemplo, a la mano asociar el tomar, al bolsillo el soltar, a la vista, obtener información, etc., este método no introduciría al dibujo de la historia elementos que le son ajenos, como pasa en los casos anteriores, pero los dibujos deben ser del tamaño adecuado y lo suficientemente detallados como para permitir la selección, luego se tendría el problema de la interpretación de lo que cada parte seleccionada significa, ejemplo, la mano se podría asociar con tomar o con activar. Por otra parte, determinar en este caso

que acciones están disponibles debería llevarse a cabo directamente con cambios en algunos detalles del dibujo del personaje o, de manera mas simple aunque menos deseable, que cuando el lector intente llevar a cabo una acción no accesible, se le indique por medio de alguna señal.

Por último, con el mouse de, al menos, dos botones se habilita un modo de activación de acciones interesante. Con uno de los botones del mouse se puede elegir diferentes actividades posibles del protagonista y a medida que el lector presiona ese botón, va cambiando, por ejemplo, el dibujo del cursor de acuerdo a la posible acción a realizar, luego de elegida la acción de este modo, con el otro botón del mouse se lleva a cabo. Este tipo de selección tiene la ventaja de que no son necesarios nuevos elementos en el dibujo de la historia en si para representar las acciones y no se depende de como se dibujara el personaje, como se vio en el método anterior. La desventaja es que no hay un acceso directo a cada opción, para determinar si esta disponible una acción se debe presionar el botón del mouse hasta dar una vuelta completa a la secuencia de cursores y solo ahí se podrá determinar la disponibilidad de la acción. Una interface que use este estilo seria altamente modal.

2.4.4 - Navegación en el ambiente.

Sea cual fuere el estilo adoptado para la implementación de la interface, este debe permitir que un lector lleve al protagonista a navegar de un contexto a otro o dentro del mismo contexto. El lector vería, en principio, un solo contexto por vez, que será el contexto en el cual este en ese momento, a pesar de ello se podrían proveer diferentes herramientas como para facilitar al lector determinar la ubicación del personaje dentro de la estructura de navegación, algunas de estas herramientas podrían ser :

Mapa : es una indicación gráfica de la estructura de navegación, en ella se debería proveer la facilidad de observar la ubicación del protagonista y la de los demás objetos y sus actividades, con lo cual se tendría una visión global que facilitaría la ubicación del lector en el espacio de la historia.

Punto de vista : se puede tener la facilidad de verificar el estado de las cosas en otros contextos, permitir que el personaje pase de un contexto a otro sin pasar por los contextos intermedios, utilizando esta facilidad con cierto cuidado de modo de no violar la metáfora de navegación espacial.

2.5 - Requerimientos del diseño e implementación de una hiperhistoria.

Ahora que definimos lo que son las hiperhistorias y vimos alguna reseña acerca de su utilización en educación, veremos que requerimientos poseen ellas que hagan necesaria la definición de un modelo formal para implementarlas.

2.5.1 - Separar el contenido de la interface.

Conceptualmente una historia puede ser descripta abstrayendo su contenido de su representación final. Por ejemplo un mismo relato puede ser presentado a través de un libro o por medio de un film. Si bien el contenido es el mismo, la forma de su presentación cambio de texto a imagen y sonido.

Si esta premisa es satisfecha, una misma historia puede ser descripta independientemente de la interface. Así esta historia puede ser presentada a través de una interface gráfica o una enteramente acústica. Este seria el caso en el que una misma historia pueda ser presentada en el primer caso a un alumno normal y en el segundo caso a un alumno ciego. Otro caso seria el de alumnos sordos, en donde señales acústicas no tienen sentido. En este caso estos eventos deberían ser llevadas a una representación adecuada. (por ej. el sonido de una alarma deberá ser cambiada a una representación gráfica apropiada)

2.5.2 - Composición, Modularidad y Herencia.

Cada entidad modelada en la historia (contextos, entidades y links) potencialmente puede ser reutilizada para la construcción de nuevas historias. Por ejemplo, es interesante tener una librería de objetos que puede existir en una casa, pues independientemente de la historia, ellos se comportan siempre igual. No solo con objetos ocurre esto, pues una casa completa puede ser utilizada en otra ocasión. Por ejemplo una clase de casa previamente definida puede volverse a usar para crear un barrio de casas.

Es así que el método de descripción debe soportar inherentemente modularidad de objetos y ambientes, permitiendo autocontener en cada unidad descriptiva los atributos y su comportamiento.

Entidades preexistentes pueden componer nuevas entidades. Es el caso del barrio de casas. Por lo tanto la composición debe ser soportada.

Una característica interesante es el de la herencia. Esta característica presente en el paradigma de OOD permite evitar redefiniciones, aplicar especialización a entidades existentes, etc. Por ejemplo, dada la definición de una auto y un avión, puede ser generalizado este concepto en una nueva clase denominada vehículo, donde se definiría el comportamiento y atributos comunes a los autos y aviones.

2.5.3 - Concurrencia de eventos.

En una historia escrita tradicional puede existir simultaneidad de acciones que involucren objetos, por ejemplo, la historia cuenta "...mientras Juan estaba caminando, el automóvil avanzaba hacia donde el se encontraba...". En la hiperhistoria, dada su naturaleza, debería poder presentar escenas como la descripta anteriormente en forma dinámica e interactiva.

Otro caso mas complejo es que algunos objetos pueden estar desarrollando una actividad en otros ambientes que no sean los que en ese momento este percibiendo el alumno. Por ejemplo, una canilla puede estar llenando un recipiente en un ambiente. El recipiente se ira llenando aunque el alumno nunca lo vea, pero en el momento que el alumno arribe al ambiente, el recipiente se deberá ver tan lleno como corresponda en ese momento.

2.5.4 - Especificación independiente del lenguaje.

El lenguaje de especificación de la hiperhistoria debe ser independiente del lenguaje utilizado para su implementación, de manera que si se quiere mudar una hiperhistoria de plataforma, no se necesitara reescribir la hiperhistoria para adaptarla al nuevo ambiente, sino solo el modulo que traduce la hiperhistoria al lenguaje de implementación.

2.5.5 - Implementación de la interface en un lenguaje distinto que para la hiperhistoria.

El contenido de la historia y su dinámica puede ser finalmente implementada en un lenguaje tradicional. Recordemos que en función del alumno, la interface puede cambiar. Es así que en función de ella, la elección del lenguaje de implementación de la interface puede variar. Por ejemplo, una interface gráfica puede ser construida y manejada con herramientas tales como Hypercard o Toolbook, pero si la presentación es enteramente acústica, lo mas probable es que utilicemos un lenguaje que provea facilidades para manejar adecuadamente el sonido (procesamiento, ejecución, modelar earcons).

En adición a lo anterior, ciertas herramientas de software poseen gran expresividad a nivel de programación, pero son débiles a nivel de presentación. Es así que esta idea permite extraer de cada producto, su punto mas fuerte. Es por esta razón que no solo es deseable una separación a nivel de diseño, sino también a nivel de implementación.

Por otra parte, es posible que una misma hiperhistoria sea "leída" simultáneamente por varios alumnos, en una red de computadoras, cada uno de ellos, en su computadora, tendrá funcionando una instancia de la interface correspondiente a la hiperhistoria que estén leyendo y la aplicación de esta funcionará en una sola computadora, la cual proveerá la coordinación y comunicación entre todas la interfaces, esto se entiende como que si, por ejemplo, un lector actúa sobre un objeto en una de las interfaces, como esta acción se traslada al kernel, este mantendrá al resto de las interfaces informadas acerca de dicha acción y cada lector, en cada una de esas interfaces, podrá advertir el cambio, con alguna señal que la interface que este manejando le presente, la naturaleza de esta señal, es decir, si es gráfica, sonora, textual, etc., dependerá en cada caso del estilo de interface que use cada lector.

Al separar la implementación del contenido de la hiperhistoria de su interface seria posible, de manera mas sencilla, que alumnos con diferentes discapacidades interactúen entre si, cada uno con una interface que se adaptaría a su tipo de discapacidad, por medio de una única hiperhistoria, la cual coordinaría el funcionamiento de todas las interfaces conectadas a ella.

2.5.6 - Objetos con comportamiento dinámico y autónomo.

Algunos de los objetos que intervengan en la hiperhistoria van a tener una actividad propia que no será motivada por acciones del alumno sobre ellos, sino que será un comportamiento propio de cada objeto. Por lo tanto es necesario definir una forma de especificar este comportamiento. Sería por ejemplo el caso de un semáforo.

Por cada objeto que se comporte de esta manera se deberá poder definir cual será su temporización, es decir, cada cuanto tiempo realizara acciones, cuando se detendrá, etc.

2.5.7 - Comunicación sincrónica y asincrónica.

En otros casos la actividad puede ser mas compleja e involucrar interacción de varios objetos autónomos, por ejemplo, en el caso de un ambiente que involucre una simulación de trafico automotor, peatones y autos interactuando de tal manera que no ocurran accidentes. Es por ello que debe existir algún tipo de comunicación entre objetos.

La forma de comunicación tradicional entre objetos es la sincrónica, en ella, un objeto invoca un método en otro y hasta que la ejecución del segundo no termine, el control no retorna al primero. Esto no alcanza para nuestro modelo, veamos el siguiente ejemplo: tenemos fichas de domino paradas formando una línea, supongamos que al mismo tiempo se tiran las dos fichas de las puntas, en la realidad, se generarían dos efectos que harán que las piezas vayan cayendo, desde ambas puntas a la vez.

Si queremos representar esto y decimos que el comportamiento de cada ficha cuando se cae es tirar a la que tiene a su lado, no alcanza el método de mensajes sincrónicos tradicional, pues cada ficha envía un mensaje a la ficha que esta hacia el lado que debe caer, y así sucesivamente. El efecto resultante sería que todas las fichas caerían hacia el mismo lado, y este lado dependería de la punta en la que se tiro la primer ficha.

En cambio usando mensajes asincrónicos y una cola de despacho adecuada, cada mensaje enviado se encolaría en ella, permitiendo que se vayan alternado los mensajes de las fichas que van cayendo de cada punta de la fila.

Es así, que la existencia de mensajes asincrónicos es justificada por este tipo de efecto deseado.

2.6 - Trabajos previos.

Por las características presentes en una hiperhistoria, este tipo de aplicación es similar de algún modo, a una aplicación de hipermedia. Pero las aplicaciones hipermediales consisten en contextos que contienen objetos que son estáticos (texto, imágenes estáticas, etc.) o bien tienen el dinamismo propio del medio que los representa (video, animación, sonido, etc.). La idea de objetos dinámicos que potencialmente pueden navegar el ambiente en forma automática y concurrente con el alumno, escapa a los modelos tradicionales de diseño de hipermedia.

Los modelos tradicionales especifican siempre entidades que no poseen comportamiento "inteligente" y dinámico. Es por ello que se hace necesaria la definición de un modelo para la especificación e implementación de hiperhistorias que satisfaga todos los requerimientos vistos anteriormente.

2.7 - Ejemplo de hiperhistoria.

Ahora vamos a explicar un pequeño ejemplo de hiperhistoria, el cual se desarrollará mas detalladamente, explicando detalles de implementación, al final del informe; aquí veremos como sería la interface y como será ser la interacción con el usuario del ejemplo.

La hiperhistoria del ejemplo se desarrolla en una casa y tiene como protagonista un personaje, el cual tiene la habilidad de pasar de una habitación a otra de la casa, actuar con los objetos que lo circundan, tomar y soltar objetos, transportarlos y asociarlos para producir ciertas acciones.

El usuario interactúa con la historia dándole órdenes a los personajes para que actúen con su ambiente, estas ordenes se indican mediante el uso del mouse, con estas órdenes, se busca que el lector lleve al personaje a efectuar una serie de acciones tendientes a resolver el problema, que es sabido por este de antemano.

En el ejemplo que vamos a describir las acciones que puede realizar el lector son sencillas y se limitan a actuar sobre los objetos con la finalidad de hacer que cambie su estado, por ejemplo, si actuamos sobre una puerta esta se abrirá, si estaba cerrada o se cerrará si estaba abierta.

También existen otros tipos de acciones a realizar, como la navegación, es decir, llevar al personaje de un contexto a otro, combinar dos objetos con la finalidad de cambiar el estado de uno de ellos, por ejemplo, al combinar una llave con una puerta, esta puede abrirse si estaba cerrada o cerrarse si no estaba cerrada con la llave.

Los efectos que se generen con estas acciones y su utilización van a depender de como esta implementada la hiperhistoria y de las acciones que en esta implementación acepte realizar cada personaje y cada objeto, es decir, si se quiere tomar un objeto cuyo comportamiento no le permita ser tomado, no ocurrirá acción alguna y el intento de tomar el objeto será ignorado.

Con esta hiperhistoria el usuario para utilizarla debe darle órdenes a los personajes para llevar a cabo una secuencia de acciones tendientes a llevar a la historia a un determinado punto, este punto será el final de la historia o el objetivo a cumplir.

Esto no es estrictamente necesario para todas las hiperhistorias, ya que se puede diseñar la historia de modo que no tenga un final preciso sino que el objetivo de la historia sea la simple interacción, este sería el caso mas común para las historias diseñadas para la estimulación de niños discapacitados en los que se busca el desarrollo de un determinado aspecto de su conocimiento.

La hiperhistoria del ejemplo ambientó en una casa en la cual se dibujaron los ambientes mas comunes en ella y ademas la vereda y una plaza, con ello se busca situar al niño en un contexto que le sea familiar.

La historia del ejemplo consiste en llevar al lector a realizar las acciones necesarias con el objetivo de extinguir el fuego que se esta propagando en el living de la casa, para eso debe hacer que el personaje navegue los contextos y recoja un vaso de agua ubicado en uno de ellos para luego arrojarlo sobre el fuego y asi apagarlo.

La finalidad de esta hiperhistoria es llevar al niño a que, urgido por el hecho de que el fuego se esta extendiendo, busque el modo de apagarlo y con ello aprenda a interactuar con el medio y utilizar los objetos que en el se hallan y asi desarrolle ciertas habilidades que lo ayuden a superar sus problemas cognitivos.

La forma de interacción que se usó se denomina "drag and drop", por ejemplo, para tomar un objeto debe arrastrarlo con el mouse desde donde se encuentra hacia el personaje, para activar un objeto debe hacer drag desde el cuerpo del peronaje hacia el objeto que se quiere activar, etc.

Con esta historia en particular se busca que el niño desarrollar los siguientes aspectos : mediante la navegación del personaje se intenta que el niño desarrolle su lateralidad, es decir, como debe mover al personaje de un lado a otro, se espera que el niño aprenda a distinguir un lado del otro, se oriente en la pantalla y pueda decidir cuando y hacia donde moverse.

Al tener, el niño, que realizar una acción en un tiempo limitado, deberá adquirir una noción de este, es decir, se percatará que el tiempo pasa y que debe realizar las acciones en un lapso limitado de tiempo para llegar a tiempo a resolver el problema, con esto se estimulan las relaciones temporales.

Mediante la navegacion a traves de contextos, se espera que el niño refuerce las relaciones espaciales, ya que debe aprender a situarse en el entorno, aprender donde se esta ubicado y hacia donde ir para alcanzar un determinado lugar u objeto.

Este es un ejemplo sencillo pero contiene los aspectos a considerar y los ejemplos del modo en que se pueden realizar historias mas grandes y con otras orientaciones o fines mas especificos.

De aquí en adelante veremos la descripción del lenguaje, la implementación del traductor, del núcleo de la historia, que llamaremos kernel, el cual se genera a partir del lenguaje, las herramientas para generar ese kernel y el modo de conectarlo a una interface, ademas de algún detalle de la implementación del ejemplo.

CAPITULO III

MODELO FORMAL PARA HIPERHISTORIAS.

3.1 - Introducción.

Vistos los requerimientos que tenemos, es necesario un modelo para el diseño e implementación de hiperhistorias que abarque todos los aspectos anteriormente descriptos.

Dicho modelo debe proveer mecanismos para soportar la especificación de la estructura de navegación, la definición de objetos autónomos con comportamiento, que soporte la concurrencia de eventos, etc.

Definiremos a estos efectos un modelo orientado a objetos, ya que este paradigma brinda la potencia necesaria para el diseño de los diferentes componentes de la hiperhistoria brindando el necesario encapsulamiento e independencia entre objetos.

La forma de interacción entre los objetos será a través de eventos los cuales permitirán la implementación de entidades cuyo funcionamiento sea concurrente con el de otras, mediante el manejo asincrónico de los mismos.

Se proveen tres clases básicas, CONTEXTO, LINK y ENTIDAD, el diseño de la hiperhistoria deberá hacerse a través de la implementación de subclases de estas. Existe otra clase, que se denomina CHANNEL, que se usará para especificar interacciones entre objetos, pero no se podrán definir subclases de esta.

Los objetos componentes de la hiperhistoria, que no sean contextos ni links, serán declarados dentro de CONJUNTOS, luego, estos últimos serán los que se incluyan dentro de los contextos correspondientes para indicar que objetos habitarán que contextos.

El modelo posee la propiedad de poder especificar la hiperhistoria abstrayéndose de cual será la representación final de la interface, esta es para los objetos que forman el kernel de la hleprhistoria un objeto más al cual enviarle eventos cuando su comportamiento asi lo indique, estos mensajes se enviarán cuando se desee que una determinada entidad de la historia genere alguna acción explicita la cual debe ser percibida por el usuario.

El comportamiento que el programador puede especificar para los objetos que programe se basa en la especificación de los eventos a los cuales cada objeto responderá, los cuales, si no son aceptados por la clase del objeto que recibe el evento se propagaran hacia arriba en la jerarquía de clases definida, si el evento, en la propagación, llega a la clase raíz y no es aceptado por esta, entonces es rechazado.

3.2 - Componentes básicos.

3.2.1 - Contextos.

Representan ambientes navegables, los cuales contarán con un conjunto de entidades que lo habitan, comunicaciones con otros contextos, lo que llamamos links, a través de las cuales los objetos incluidos dentro del el pueden cambiar de contexto, y un comportamiento.

Los contextos tienen un comportamiento predefinido y el usuario puede, en las subclases de este, escribir su propio comportamiento, el cual puede agregarse al ya existente o sobreescribirlo, dependiendo de las reglas que se reimplementen.

En lo que respecta a la propagación de eventos, si un contexto recibe un evento y no lo tiene como cabecera de ninguna regla, es retransmitido a todos los objetos que habitan ese contexto, de este modo el contexto provee un mecanismo de interacción entre un objeto y sus cohabitantes en el contexto, permitiendo a un objeto, que quiera enviar un mensaje a los demás objetos que estén en su mismo contexto, hacerlo simplemente enviándole un mensaje al contexto que lo contiene.

3.2.2 Links.

Es la clase que representa la comunicación entre contextos. Se caracteriza por tener referencias a los contextos que comunica, que pueden ser dos o más, dependiendo de como se lo ligue en la definición del contexto que lo contiene.

El comportamiento del link determinará la naturaleza de la conexión (unidireccional, bidireccional, que par de contextos, entre los que están ligados a él, comunica, etc.). Parte de este comportamiento ya esta predefinido en los links, pero si se le quiere dar

un comportamiento especial, el programador puede definirlo.

El conjunto de contextos y links forman la estructura de navegación de la hiperhistoria, luego esta puede ser recorrida por las entidades que la pueblan, siempre y cuando el comportamiento de estas así lo indique o lo permita.

3.2.3 - Entidades

Es la clase básica de la cual derivará cualquier objeto de la hiperhistoria. Personajes y objetos, no importando su dinámica, serán alguna instancia de una clase que es subclase de la clase entidad.

Las entidades son los únicos objetos que pueden moverse a través de la estructura de navegación. Esta clase de objeto, como cualquiera de los objetos precedentes, tienen un comportamiento predefinido y, en subclases de esta, se le puede implementar un comportamiento que indique como se actuarán a lo largo del desarrollo de la historia.

3.2.4 - Canales.

Representa el medio de conexión entre dos o mas componentes entre si en forma explícita. Representa una ligadura de causa/efecto entre entidades, links o contextos entre si.

Los canales no pueden tener subclases, y el único comportamiento que poseen es el de transmitir un evento que reciban desde uno de los objetos a los cuales este ligado hacia todos los demás objetos conectados a el excepto el generador del evento.

3.2.5 - Conjuntos.

Los conjuntos son el medio por el cual se crean e inicializan las entidades y canales que intervendrán en la hiperhistoria.

En la definición de estos se indica que objetos lo compondrán y como serán inicializados, luego, para hacer que los objetos especificados sean incluidos en un contexto se debe incluir el conjunto completo en el contexto.

Se puede incluir mas de un conjunto por contexto e incluso el mismo conjunto se puede incluir en mas de un contexto, esto generara instancias distintas de los objetos incluidos en ese conjunto en cada uno de los contextos en los cuales se incluya.

3.2.6 - La Interface.

La interface no es un objeto que este dentro del kernel de la hiperhistoria, pero los objetos que estan en el pueden referirse a ella para indicar que estan haciendo una accion que puede ser percibida.

Dentro del comportamiento de cada objeto se puede indicar que se le envía un evento a la representación del objeto en la interface, de esta manera se tiene un modo de mantener una comunicación explícita entre esta y la hiperhistoria.

Existe también un modo implícito para comunicarse con la interface a través de la modificación del valor de atributos comunes del objeto declarado, esto se explicará mas adelante.

3.2.7 - El Timer.

Existe un objeto particular en la hiperhistoria que se denomina timer, aquellos objetos que deban tener un comportamiento autónomo, es decir, que cada cierto tiempo realicen ciertas acciones, deben programarse de manera que envien eventos al timer para programarlo, a partir de ese momento, a intervalos de tiempo regulares, el timer le enviará eventos al objeto que desde el cual se realizó la programación, en la implementación de estos eventos se debe programar el comportamiento autónomo.

El timer no es un objeto exclusivo de la entidad de la hiperhistoria que lo programe, este llevará el control de todas las entidades que requieran su temporización en forma simultánea.

3.3 - Sintaxis para la especificación.

Nuestro formalismo establece una plantilla de especificación para cada una de las clases descriptas anteriormente. Una

plantilla de especificación que establece una sintaxis especifica para cada una de las subclases de ella. De esta manera explícitamente no se incluyen constructores en clases en las que no tiene sentido semántico, por ejemplo, no tiene sentido incorporar un constructor para definir contextos dentro de una subclase de link.

La ventaja de particionar así la definición, es que para cada tipo de plantilla, la sintaxis es propia y adaptada a la funcionalidad deseada de la clase. Pasaremos a definir cada una de las plantillas.

Significado del método usado para describir la sintaxis.

- **Mayúsculas**: símbolos reservados del lenguaje.
- **Minúsculas**: construcciones definidas por el programador.
- **[...]**: construcción opcional.
- **[..|..]**: una o ninguna de las opciones indicadas.
- **<..|..>**: una de las opciones indicadas.
- **+**: la construcción se puede repetir 1 o mas veces.
- *****: la construcción se puede repetir 0 o mas veces.

3.3.1 - Plantillas de definición.

3.3.1.1 - Contextos.

```
CLASS nClase IS A <CONTEXT | claseContextoPredefinida>
    [ ENTRYS
        declaración de entradas ]
    [ CONNECTIONS
        declaración de conexiones ];
    [ CONTEXTS
        declaración de contextos + ]
    [ LINKS
        instanciación de links + ]
    [ ATTRIBUTES
        declaración de atributos + ]
    [ INTERNAL ATTRIBUTES
        declaración de atributos + ]
    [ INITIALIZATION
        [ inicialización de atributos ]*
        [ inicialización de links ]*
        [ inclusión de entidades ]*
        [ conexionado de objetos ]* ]
    [ BEHAVIOR
        especificación de comportamiento]
END
```

3.3.1.2 - Links.

```
CLASS nClaseLink IS A <LINK | claseLinkPredefinida>
    [ CONNECTIONS
        declaración de conexiones ]
    [ ATTRIBUTES
        [ declaración de atributos ]+ ]
```

[INTERNAL ATTRIBUTES
[declaración de atributos]+]
[INITIALIZATION
[inicialización de atributos]+]
[BEHAVIOR
especificación de comportamiento]
END

3.3.1.3 - Entidades.

CLASS nClase IS A <ENTITY | claseEntidadPredefinida>
[CONNECTIONS
declaración de conexiones]
[ATTRIBUTES
[declaración de atributos]+]
[INTERNAL ATTRIBUTES
[declaración de atributos]+]
[INITIALIZATION
[inicialización de atributos]+]
[BEHAVIOR
especificación de comportamiento]
END

3.3.2 - Componentes de las plantillas.

3.3.2.1 - Entradas.

Las entradas a los contextos son referencias a contextos internos que se definen en un contexto, deben especificarse cada vez que teniendo, un contexto, contextos internos, al entrar un objeto al contexto externo, en realidad entra uno de los contextos incluidos en este.

La sintaxis para escribir que entradas tendrá cada contexto es la siguiente;

ENTRYS nomEntrada [, nomEntrada]*;

Las palabras usadas como nombres de entradas no tienen que ser palabras reservadas, nombres de clases ya definidas, nombres de tipos definidos, etc.

3.3.2.2 - Contextos internos.

Esta declaración debe utilizarse para indicar que el contexto que esta siendo definido tiene en su interior otros contextos que son parte de la estructura de navegación.

La sintaxis para definirlos es la siguiente:

CONTEXTS
nCont1[,nCont2]* [IS A <CONTEXT| clasePredefinida >];

Las palabras usadas como nombres de contextos no deben ser palabras reservadas, nombres de clases ya definidas, nombres de tipos definidos, etc.

La clase predefinida debe ser alguna clase ya implementada que sea descendiente de CONTEXT.

3.3.2.3 - Links internos.

Los links que se definen en este punto son aquellos que comunicarán contextos internos al contexto que esta siendo definido. Esta sección solo se limita a la declaración de los mismos, la definición de cuales son los contextos que comunica se hace mas adelante en la sección de inicialización del contexto. La sintaxis para definir los links es la siguiente:

LINKS
nomLink1[,nomLink2]* IS A <LINK | clasePredefinida >;

Las palabras usadas como nombres de links no deben ser palabras reservadas, nombres de clases ya definidas, nombres de tipos definidos, etc. La clase predefinida debe ser alguna clase ya implementada que sea descendiente de LINK.

3.3.2.4 - Conexiones.

Las conexiones son referencias a otros objetos que se dejan pendientes para que luego sean conectadas a otros objetos o canales, los cuales recibirán los eventos que a través de ellas se envíen.

La sintaxis para especificar las conexiones que tendrán los objetos de la clase que esta siendo definida es la siguiente::

CONNECTIONS nConexion1 [, nConexion2]*;

Las palabras usadas como conexiones no deben ser palabras reservadas, nombres de clases ya definidas, nombres de tipos definidos, etc.

3.3.2.5 - Atributos.

Los atributos son los que especifican el estado interno de cada objeto a lo largo de la hiperhistoria.

La sintaxis para la declaración de los atributos es la siguiente:

ATTRIBUTES nomAtributo1[,nomAtributo2]* : nomTipo;

Las palabras usadas como conexiones no deben ser palabras reservadas, nombres de clases ya definidas, nombres de tipos definidos, etc. El tipo de los atributos puede ser alguno de los siguientes:

INTEGER | WORD | BYTE | LONGINT|
STRING | BOOLEAN | HOBJECT

Los primeros cuatro son tipos numéricos comunes, el tipo string es una cadena de caracteres, boolean es true o false y el ultimo tipo, hObject, representa un objeto cualquiera de la hiperhistoria al cual se le pueden enviar eventos.

Los atributos comunes tienen la particularidad de que al cambiar su valor, automáticamente se notifica a la interface del cambio.

3.3.2.6 - Atributos internos.

Los atributos internos son similares en cuanto a su sintaxis para la declaración y utilización, la única diferencia es que no se notifica a la interface cuando estos cambian de valor.

3.3.2.7 - Inicialización.

Esta sección define como se inicializará cada instancia de la clase que se esta creando.

Dentro de la inicialización se pueden realizar varias acciones:

3.3.2.7.1 - Inicialización de atributos.

Aquí se inicializan los atributos comunes e internos de la clase que se esta definiendo, la sintaxis es:

atributo := expresión constante;

El tipo del resultado de la expresión debe ser el mismo que el del atributo que se quiere inicializar.

3.3.2.7.2 - Inicialización de los links.

Indica que links unirán determinados contextos, esta declaración es exclusiva de la clase CONTEXT.

Los contextos que se utilicen deben ser los definidos en la sección CONTEXTS y los links, los definidos en la sección LINKS del mismo contexto que se esta creando.

Esta es la sintaxis para unir contextos por medio de links en la definición de otro contexto:

nomContexto LINKED WITH nomLink[,nomLink];*

Donde dice *nomLink*, en lugar de uno de los links definidos, también puede ir el nombre de una entrada definida en el contexto en la sección ENTRYS.

3.3.2.7.3 - Inclusión de entidades dentro de los contextos.

Esta declaración indica que conjuntos de objetos se incluirán en cada contexto, ya sea el que esta siendo definido o cualquiera de los contextos internos.

La sintaxis para la declaración es:

< nomCont[.nomContInterno]*] | SELF > INCLUDES espConjunto

La sintaxis de la especificación de conjunto es la siguiente:

nomConjunto [+ nomConjunto]*

Cuando a la cláusula INCLUDES la antecede la especificación SELF, los objetos serán incluidos dentro del contexto que esta siendo definido.

3.3.2.7.4 - Conexionado de los objetos dentro del contexto.

Esta declaración se debe utilizar para ligar dos o mas objetos entre si a través de canales o directamente entre si.

La sintaxis para la declaración es;

objeto1 CONNECT [nomConexion] WITH objeto2;

Esto significa que el *objeto1* se conectara al *objeto2* a través de la conexión *nomConexion* si esta se especifica, en este caso el *objeto1* podrá enviar eventos hacia el *objeto2* a través de la conexión *nomConexion*, en caso contrario el *objeto2* deberá ser un canal para que la conexión tenga sentido y solo podrá recibir eventos a través de ese canal.

SI el *objeto2* es un canal, entonces se podrá conectar este a mas de un objeto, en caso contrario, la conexión es uno a uno.

Objeto1 y *objeto2* son las especificaciones de los objetos que deben conectarse, la sintaxis para escribirla es:

[nomContexto1[.nomContexto2]*].nomObjeto

Esta es una notación puntual para decir a que objeto me estoy refiriendo según el contexto en el que este se encuentre.

Si solo se pone el nombre del objeto, estoy haciendo referencia a un objeto incluido dentro del contexto en el que hago la definición.

3.3.2.8 - Comportamiento.

El comportamiento de los objetos que intervienen en la hiperhistoria se especifica a través de reglas agrupadas en bloques anidados.

La sintaxis para escribir los bloques es la siguiente:

BEHAVIOR
BLOCK <nombreBloque>
    [reglas]*
    [bloques]*
END;

La sintaxis para escribir las reglas con las es la siguiente:

(<nomEvento>[(parámetros)],
precondición,
sentencias,
postcondición)

El nombre de evento debe ser una cadena de caracteres que debe comenzar con una letra y que puede seguir con números, letras o carácter subrayado, los diez primeros caracteres del nombre son significativos, el resto es ignorado.

La sintaxis para escribir los parámetros es la siguiente:

[VAR] nombre [, nombre]* : tipo;

El prefijo var indica que, en los parámetros siguientes en la declaración hasta la especificación del tipo, si el que envía el evento coloca una variable, esta modificara su valor según la como la implementación de la regla modifique el valor del parámetro.

La precondición consiste en una expresión que debe retornar true o false, en la expresión pueden intervenir atributos del objetos, comunes o internos, y parámetros pasados a la regla.

Las sentencias especificarán que acciones llevará a cabo el objeto cuando acepte el evento que encabece la regla.

Hay cinco tipos de sentencia: conjunto de sentencias, declaración de variables locales a la regla, asignación, envío de evento y ejecución condicional.

El conjunto de sentencias es simplemente la agrupación de estas entre paréntesis, la sintaxis es:

(sentencias);

Esta construcción determina bloques de sentencias que serán útiles cuando se use la ejecución condicional.

La sintaxis para la declaración de variables locales es la siguiente:

LOCAL <nombre> [, nombre]* : tipo;

Las reglas para los nombres de las variables locales son las mismas que para los parámetros, estas variables tiene vida únicamente es la regla que las declara, desapareciendo una vez que esta termina su ejecución.

La sintaxis de la asignación es la siguiente:

<variable> := <expresión>;

La variable puede ser un atributo de la clase, un parámetro pasado a la regla o una variable local declarada en la misma regla.

La expresión puede ser cualquiera que involucre estos componentes y cuyo tipo resultante sea el mismo que la variable a asignar.

Otra de las sentencias es el envío de eventos a otro objeto de la hiperhistoria, la sintaxis es la siguiente:

objeto< <--|<== >evento[(parámetro [,parámetro]*)];

El objeto puede ser un atributo común o interno de la clase definida, un parámetro de la regla o una variable local a la lista de acciones, del tipo hObject o una conexión definida en la misma clase en la sección CONNECTIONS.

El símbolo <-- indica el envío de un evento sincrónico y el símbolo <== indica el envío de un evento asincrónico.

Los parámetros pueden ser atributos comunes o internos de la clase, parámetros de la regla, variables locales a la lista de acciones de la regla corriente o expresiones de variables o constantes.

Por último se tiene la ejecución condicional de sentencias, la sintaxis es la siguiente:

```
IF (expresión booleana) THEN
    sentencia ;
[ ELSE
    sentencia; ]
```

La expresión booleana debe ser cualquiera que retorne true o false y las sentencias deben ser cualquiera de las mencionadas anteriormente, incluso otra condicional.

3.3.3 - Conjuntos.

Los conjuntos son construcciones especiales que permiten agrupar objetos para luego incluirlos en alguno de los contextos de la estructura de navegación de la hiperhistoria. La sintaxis para si definición es la siguiente:

```
SET <nombre del conjunto>
    nObjeto [ , nObjeto ]* IS A claseObjeto; ...
    INITIALIZATION
    nObjeto.atributo := <expresión constante>; ...
END;
```

Los nombres de objetos deben ser cualquier cadena de caracteres que comience con una letra y siga solo con letras, números o carácter subrayado.

La clase del objeto puede ser ENTITY, CHANNEL o cualquier subclase de entity definida por el usuario.

La expresión debe ser una constante o una expresión valida formada por constantes cuyo tipo resultante sea el mismo que el del atributo a inicializar.

Los objetos especificados en el conjunto se instanciarán cada vez que el conjunto declarado sea incluido en algún contexto.

De la misma manera, la inicialización se ejecutará cada vez que el conjunto declarado sea incluido en algún contexto.

3.4 - Semántica y funcionamiento de las componentes.

3.4.1 - Conjuntos.

3.4.1.1 - Definición.

Los conjuntos no son entidades que intervendrán en la hiperhistoria, sino que son construcciones que permiten definir las entidades concretas que intervendrán en ella para luego incluirlas en los contextos correspondientes.

Esta construcción tiene la finalidad de agrupar la declaración de objetos que se deberán declarar juntos en la hiperhistoria y que van a ser colocados, siempre juntos, en el mismo contexto.

También dentro de los conjuntos es posible inicializar los atributos comunes e internos de los objetos en el declarados antes de que sea incluidos en los contextos.

3.4.1.2 - Declaración.

Los conjuntos se pueden declarar en cualquier momento, fuera de cualquier clase y antes de la declaración del contexto en el cual se usen.

La forma en que estos se deben usar en los contextos se vera cuando se vea el uso de la cláusula INCLUDES de estos.

Ahora veremos ejemplos de como se deben declarar los conjuntos, supongamos que en un contexto deseamos incluir una llave de luz y una lámpara, para ello definiremos un conjunto como sigue:

```
SET llaveLampara
    llave IS A clase_llave;
    lampara IS A clase_lampara;
INITIALIZATION
    lampara.encendida := false;
END;
```

En este caso, se declaran dentro del conjunto dos entidades (cuyas clases ya deben estar definidas) y se inicializa el atributo *encendida* de la entidad *lampara*.

Si luego nosotros incluimos este set en dos contextos distintos, en cada uno tendremos una llave y una lámpara, con el mismo nombre, por lo tanto no se debe incluir el mismo conjunto mas de una vez en el mismo contexto ya que solo se tendrá en cuenta la primera inclusión.

3.4.2 - Entidades.

3.4.2.1 - Definición.

Las entidades representan a todos los objetos que intervienen en la hiperhistoria exceptuando los canales.

En realidad, los contextos y los links son subclases de una entidad básica, por lo tanto heredan su comportamiento, excepto, claro está, aquel comportamiento que es redefinido en cada subclase.

Cualquier objeto definido en la hiperhistoria que no sea un contexto o un link debe ser una entidad, la cual tendrá como comportamiento especifico, además del común que se explicará mas adelante, el de poder navegar de un contexto a otro a través de los links y ser tomado o soltado por otra entidad, el resto del comportamiento es también aplicable a los contextos y a las entidades.

3.4.2.2 - Instanciación.

Las entidades se instanciarán cada vez que el conjunto en el cual se declaren sea incluido en algún contexto, esto implica que

si un mismo conjunto es incluido en dos contextos distintos, en la hiperhistoria se instanciarán dos veces, con el mismo nombre, los objetos declarados dentro del conjunto.

La identificación de los objetos, en caso de querer conectarlos con otros objetos o con canales en la sección de inicialización del contexto que se esta declarando o de un contexto mas externo, se hace considerando, además del nombre de los objetos, el o los contextos en los cuales el objeto esta incluido, por lo tanto, es posible individualizar objetos con el mismo nombre siempre que se incluyan en contextos distintos.

El siguiente es un ejemplo que ilustra esta semántica:

```
SET objetos
    lampara, silla IS A ENTITY;
    INITIALIZATION
        lampara.color := 12;
        silla.color := 6;
END;

CLASS casa IS A CONTEXT
    CONTEXTS
        living, cocina IS A CONTEXT;
    INITIALIZATION
        living INCLUDES objetos;
END;
```

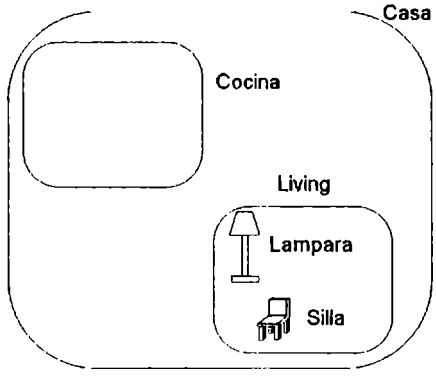
En el ejemplo, se crea una casa conformada por dos contextos internos, uno de ellos llamado *cocina* y otro, *living* y con dos entidades dentro de este último, una llamada *lampara* y otra, *silla*.

La identificación de los objetos que están dentro del contexto *living* será *living.lampara* o *living.silla,* esto siempre y cuando la casa sea el contexto mas externo o la referencia a estos objetos se haga de esta manera dentro de la inicialización de la clase que esta siendo declarada.

En el siguiente gráfico se muestra como quedaría la estructura de contextos y entidades incluidas de este último ejemplo.

La identificación de los objetos desde fuera del kernel sera el nombre de cada objeto precedido por todos los contextos en los que esta incluido, los objetos en el ejemplo se denominarían entonces *casa.living.lampara* y *casa.living.silla.* En caso de querer hacer referencia a un contexto se sigue con la misma sintaxis, o sea, *casa.cocina*, *casa.living* para referirse a los contextos dentro del contexto *casa* y *casa* simplemente, para referirse al contexto mas externo.

Este ultimo nivel de especificación solo será usado cuando se ligue efectivamente el kernel a la interface para determinar la correspondencia de objetos entre uno y otro.



3.4.2.3 - Atributos predefinidos.

Las entidades tienen atributos predefinidos los cuales pueden ser utilizados luego en la implementación del comportamiento de las entidades, contextos y links, ya que estos últimos los heredan.

Los atributos predefinidos para las entidades son los siguientes:

Id : string

Nombre con que se declara el objeto en el conjunto, no incluye los contextos en los que esta incluido, no debe ser modificado.

Icl : string

Nombre de la clase del objeto. No debe ser modificada.

Sender : hObject

Cuando el objeto recibe un evento, se instancia con el objeto que lo envía.

Ok : boolean

Se instancia con true si el evento inmediato anterior enviado en el comportamiento fue aceptado. Hay que tener en cuenta que cuando se envía un evento de manera asincrónica esta variable siempre será true.

MyContext : hObject

Se instancia con el contexto en el cual el objeto esta, este atributo cambia de valor cuando el objeto pasa de un contexto al otro a través de un link. Cuando el objeto esta tomado por otro, el valor es NULL.

Owner : hObject

Se instancia con el objeto que toma al objeto definido, cuando no esta tomado por nadie, el valor es NULL.

Self : hObject

Se instancia con el mismo objeto.

Los atributos de tipo hObject se deben tratar como a cualquier otra referencia a un objeto, es decir, se le pueden enviar eventos y pasarlos como parámetros.

Los de otros tipos pueden ser usados en cualquier expresión pero no asignados.

3.4.2.4 - Atributos definidos por el usuario.

El programador puede definir, para las clases que desarrolle, atributos, los cuales determinarán el estado interno de cada entidad en cada momento de la hiperhistoria.

Los nombres válidos para los atributos son cualquier cadena de caracteres que comience con una letra y que contenga, luego, solo letras, números y el carácter subrayado.

Los atributos no pueden tener el mismo nombre que las palabras reservadas, que los nombres de atributos predefinidos o ya definidos por el usuario, etc.

Solo los diez primeros caracteres del atributo son significativos.

Veamos un ejemplo de declaración en una subclase de entidad definida por el usuario:

```
CLASS clase_lampara IS A ENTITY
    ATTRIBUTES
        encendida : BOOLEAN;
    INITIALIZATION
        encendida:=false;
    BEHAVIOR
    BLOCK main
        (pulso,true,encendida:=not encendida;,NULL)
END;
```

En este caso se define la entidad *clase_lampara*, la cual tiene un atributo booleano que es *encendida*.

En la sección de inicialización se le asigna el valor false, esto quiere decir que cuando una entidad de esta clase, que llamaremos *lampara*, sea incluida en un contexto, al arrancar la hiperhistoria, el atributo *encendida* tendrá este valor.

Por último, en el comportamiento, cada vez que la *lampara* recibe el evento *pulso*, el atributo encendida cambia su valor.

En este caso como el atributo es común, cada vez que cambie el valor, la implementación de la hiperhistoria generara, para la representación de la lampara en la interface, el evento *encendida*, con el valor correspondiente al atributo como parámetro.

El comportamiento de la lámpara en la implementación de la interface deberá responder, a este evento, generando una representación del cambio de valor del atributo que se corresponda con el estilo de la Interface.

En el ejemplo, y para una interface visual, se cambiará la representación de la lámpara apagada por la de la misma lampara encendida, en caso de que esta sea visible al usuario de la interface.

En caso de una interface acústica se podría emitir una señal sonora que indique que la lámpara se encendió, si es que este cambio de estado es relevante para este tipo de interfaces.

Para evitar que el cambio de valor de un atributo genere un evento hacia la interface, este de debe declarar como interno.

3.4.2.5 - Comportamiento básico predefinido.

El comportamiento de una entidad se define mediante la implementación de reglas, estas tienen como cabecera un evento, una precondición, parámetros, una lista de acciones internas y una postcondición.

Existen eventos a los que las entidades responden por el hecho de ser tal, estos eventos se denominan predefinidos y los parámetros, la precondición, las acciones y la postcondición ya están definidos.

Estos eventos, para los objetos cuyas clases son entidad o alguna subclase de ella, incluyendo a los contextos y los links son los siguientes:

GetName(var name : string)

Retorna en el parámetro el nombre de la entidad receptora del evento, este nombre es el que se indica cuando se declara la entidad en un conjunto sin incluir los nombres de los contextos externos.

GetClass(var class : string)

Retorna en el parámetro en nombre de la clase de la entidad receptora del evento.

GetContext(var cont : hObject)

Retorna el contexto en el cual esta incluida la entidad receptora del evento, si esta no esta tomada por ninguna otra.

GetOwner(var own : hObject)

Retorna la entidad que tiene tomada a la receptora del evento, si esta tomada por alguna, si no retorna NULL.

TakeBy(obj : hObject)

Hace que la entidad receptora del evento sea tomada por la entidad pasada como parámetro.

LeaveBy(obj : hObject)

Hace que la entidad receptora sea soltada por la entidad pasada como parámetro, siempre, claro esta, que esta ultima entidad sea la que en realidad la tenga tomada, en caso contrario, el evento no tiene efecto.

Los eventos *TakeBy* y *LeaveBy* generan sendos eventos hacia el contexto que contiene al receptor, estos eventos son la notificación de que un objeto fue tomado o dejado en el contexto; el evento que genera *TakeBy* es *oTaked(obj : hObject)*, donde en *obj* se envía el objeto tomado y el evento que genera *LeaveBy* es *oLeaved(obj : hObject)* que indica que el objeto *obj* fue soltado en el contexto receptor. Los contextos no responden de manera predeterminada a estos eventos.

Los siguientes eventos son para acceder a la colección de objetos que tiene cada entidad :

ObjFirst

Hace que la entidad corriente de la colección sea el primero.

ObjNext

Hace que la entidad corriente sea el siguiente en la colección.

ObjCurr(var obj : hObject)

Retorna en el parámetro la entidad corriente en la colección.

ObjIsLast(var last : boolean)

Determina si la entidad corriente en la colección es el ultimo elemento de la misma, si es así, asigna true al parámetro enviado, el caso contrario, el valor que le asigna es false.

La precondición para los eventos antes mencionados es siempre *true*, la lista de acciones es la que permite ejecutar el comportamiento descripto mas arriba y la postcondición es siempre *NULL*.

Todos los eventos del comportamiento predefinido anteriormente citados deben ser enviados de manera sincrónica para que tengan el efecto mencionado.

En caso de que sean enviados de manera sincrónica, no se podrá asegurar que tengan el funcionamiento especificado ya que no se puede determinar en el momento que se despacharán, además, el valor del parámetro de salida, aquellos que lo posean,

será indeterminado.

Esta última consideración es similar para los eventos definidos por el usuario en las subclases que desarrolle.

Mas adelante veremos ejemplos del uso de estos eventos en la implementación de una clase definida por el usuario.

3.4.2.6 - Comportamiento definido por el usuario.

3.4.2.6.1 - Bloques.

Las reglas que se deben definir para especificar el comportamiento de las entidades desarrolladas por el programador deben agruparse en bloques anidados, éstos estarán, según el comportamiento implementado, algunos activos y otros no, solo las reglas incluidas en bloques activos serán las que se podrán ejecutar ante la llegada de un evento, el resto permanecen inhabilitadas hasta que se active el bloque que las contiene.

3.4.2.6.1.1 - Anidamiento.

Los bloque se pueden incluir unos dentro de otros según lo requiera el comportamiento que se quiere definir para la entidad.

Siempre debe existir un bloque mas externo, llamémosle, de ahora en adelante, de nivel 0 (cero), luego cada bloque debe contener primero, dentro de él, las reglas y luego los bloques internos, que llamaremos, de ahora en adelante, de nivel 1 (uno) para los que estén dentro del bloque de nivel 0, de nivel 2 (dos) para los que estén dentro de bloques de nivel 1 y así sucesivamente.

3.4.2.6.1.2 - Activación.

Que bloques estén activados y cuales no, al inicio de la ejecución del comportamiento de la entidad, depende del anidamiento de estos, luego, usando las postcondiciones de las reglas se puede modificar la condición de activado o no de un bloque.

El activar un bloque significa que todas las reglas que estén contenidas en el, cuando llegue un evento, podrán ser ejecutadas, esto en caso de que se cumpla su precondición,

Las reglas que se encuentren en bloques no activos, no podrán ser activadas aunque se llegue el evento y se cumpla la precondición.

Las reglas que determinan la activación de los bloques son las siguientes:

I - Al iniciarse la hiperhistoria, en todas las entidades que la componen, incluso contextos y links, que tengan definidas un comportamiento, se activa el bloque de nivel 0.

II - Al activar un bloque de nivel N se activa el primer bloque de nivel N+1 contenido en el si existe.

Ejemplo:

```
BLOCK main
    <reglas A>
    BLOCK uno
        <reglas B>
    END;
    BLOCK dos
        <reglas C>
    END;
END;
```

Aquí al activarse el bloque *main* se activara el bloque *uno,* por lo tanto, las reglas identificadas como A o B están disponibles para ser ejecutadas, no así las que están identificadas como C.

III - Una regla A incluida en un bloque de nivel N, puede activar bloques según el siguiente esquema:

- Un bloque de nivel N+1, para bloques incluidos en el mismo bloque que contiene a la regla A.
- Un bloque de nivel N, para bloques no incluidos en el mismo bloque que contiene a la regla A.

Ejemplo:

```
BLOCK main
    BLOCK uno
        < regla A >..
        BLOCK tres
        END;
        BLOCK cuatro
        END;
    END;
    BLOCK dos
        BLOCK cinco
        END;
        BLOCK seis
        END;
    END;
END;
```

Observemos este ejemplo, la regla identificada como A es de nivel 1, por lo tanto, por la primer condición podrá activar los bloques *tres* y *cuatro*, es decir los bloques de un nivel mas que el que contiene a la regla incluidos en el mismo bloque.

Por la segunda condición podría activar los bloques *uno* y *dos* los cuales están al mismo nivel del bloque que contiene a la regla. La regla A no podría activar los bloques *cinco* y *seis*, ya que son internos a el bloque dos que tiene el mismo nivel del bloque que los contiene; pero podría activar en forma indirecta el bloque *cinco* activando el bloque *dos*, ya que por (II), al activar un bloque se activa el primer bloque interno contenido en este.

IV - Cuando un bloque se desactiva, también se desactivan los bloques contenidos en el.

Ejemplo:

```
BLOCK main
    BLOCK uno
        BLOCK dos
        END;
    END;
END;
```

En este ejemplo, cuando el bloque *uno* se desactiva, se desactiva también el bloque *dos*.

V - Cuando una regla que esta dentro de un bloque X activa un nuevo bloque de igual o menor nivel que X, origina que el bloque que contiene a X de nivel igual al nuevo se desactive.

Ejemplo:

```
BLOCK main
    BLOCK uno
        BLOCK tres
        END;
        BLOCK cuatro
            <regla A>
        END;
    END;
    BLOCK dos
    END;
END;
```

Observemos que, en el ejemplo, cuando la regla A activa el bloque *dos*, el bloque *uno* y todos los bloques en el incluidos se desactivarán.

VI - Cuando una regla A activa un bloque X de nivel mayor al bloque que contiene a A, se desactiva el bloque activo de igual nivel que el bloque X.

Ejemplo:

```
BLOCK main
    <regla A>
    BLOCK uno
    END;
    BLOCK dos
    END;
END;
```

En este ejemplo, supongamos que el bloque *uno* esta activo, si la regla A activa al bloque *dos*, se desactiva el bloque *uno*.

3.4.2.6.1.3 - Herencia de la estructura de bloques.

Dada una clase definida por el usuario en la cual se haya definido un comportamiento, las subclases de esta heredarán la estructura de bloques especificada para la clase mencionada.

Luego, si el nuevo objeto necesita, por alguna razón, modificar el comportamiento agregando bloques a la estructura, puede hacerlo respetando las siguientes reglas:

I - Para agregar un bloque al principio de los bloques internos de otro bloque, se debe colocar anidado en primer lugar en la estructura de la nueva clase.

Ejemplo :

Supongamos tener definida en una clase la siguiente estructura de comportamiento.

```
BLOCK main
    BLOCK uno
    END;
    BLOCK dos
    END;
END
```

Y supongamos que queremos primero definir una subclase de esta, pero queremos que el primer bloque anidado en *main* sea *nuevo* y que luego siga el bloque *uno* y luego el *dos*. Para especificar esto se haría así:

```
BLOCK main
    BLOCK nuevo
    END;
END;
```

Con esta declaración, la nueva clase tendría la siguiente estructura de bloques:

```
BLOCK main
    BLOCK nuevo
    END;
    BLOCK uno
    END;
    BLOCK dos
    END;
END;
```

II - Para agregar un nuevo bloque entre dos bloques definidos en la superclase o como el último bloque anidado, se debe especificar, vacío, el bloque anterior después del cual se colocara el nuevo bloque.

Ejemplo :

Supongamos tener definida la misma superclase del ejemplo anterior pero ahora queremos agregar un bloque anidado en *main* entre los bloques *uno* y *dos*. En este caso lo que tendríamos que especificar es lo siguiente:

```
BLOCK main
    BLOCK uno
    END;
    BLOCK medio
    END;
END
```

Con esta declaración, la subclase quedaría con la siguiente estructura de bloques:

```
BLOCK main
    BLOCK uno
    END;
    BLOCK medio
    END;
    BLOCK dos
    END;
END
```

III - Para agregar reglas dentro de bloques definidos en la superclase, se deben especificar todos los bloques externos al bloque en que se quieren agregar reglas y el mismo bloque.

Ejemplo :

Supongamos tener la superclase de los ejemplos anteriores y que queremos agregar la regla A en el bloque main y la regla B en el bloque dos, entonces tenemos que escribir:

```
BLOCK main
    < regla A >
    BLOCK dos
        < regla B >
    END;
END;
```

Con esta definición, el bloque main de la nueva clase tiene, además de las reglas que tenia, la regla A, y el bloque dos de la subclase, tiene la regla B además de las que tenia.

3.4.2.6.2 - Reglas.

3.4.2.6.2.1 - Orden de ejecución.

Cuando un objeto recibe un evento, se busca, entre las reglas que están en los bloques activos, una cuyo encabezado coincida con el nombre del evento que recibe el objeto, luego, si la precondición de la regla es verdadera, se ejecutara la lista de acciones internas de la regla y la postcondición.

El orden en que la búsqueda de la regla que coincida con el evento recibido se realiza como sigue:

1. Se busca manejar el evento en el bloque de mayor nivel activo, llamémosle de nivel N.
2. Si no hay reglas con el mismo encabezado o la precondición de las reglas que hay es falsa, buscar en el bloque de nivel N -1 que engloba al de nivel N.
3. Esto se hace hasta que se llegue al nivel 0.
4. Si el bloque de nivel 0 no hay ninguna regla que cumpla las condiciones para ser ejecutada, se repite el proceso descripto en 1, 2 y 3 desde el bloque de nivel N, siempre que exista, en la superclase del objeto que recibió el evento.
5. Esto se repita para todas las superclases hasta que se active una regla o se llegue a la clase mas alta en la jerarquía.
6. Si no lo maneja ningún bloque el evento se rechaza.

Si el bloque activo en la subclase no existe en la superclase, se busca manejar el evento en el bloque de la superclase que engloba al bloque agregado en la subclase.

Ejemplo:

Supongamos la siguiente estructura de comportamiento en la clase X:

```
BLOCK main
        ( A, <precondición>, ... )
        BLOCK uno
            ( A, <precondición>, ... )
        END;
        BLOCK dos
            ( A, <precondición>, ... )
        END;
END;
```

Luego definamos una subclase de X, llamada Y, en la que agregamos nuevos eventos y bloques como sigue:

```
BLOCK main
    ( A, <precondición>,.... )
    BLOCK uno
    END;
    BLOCK nuevo
        ( A, <precondición>, ... )
        BLOCK nuevoUno
            ( A, <precondición>,.... )
        END
        BLOCK nuevoDos
        END
    END
END
```

Supongamos que el bloque *nuevoUno* sea el bloque activo de mayor nivel en este momento, y que el objeto recibe el evento A, la búsqueda, en este caso, se inicia con las reglas contenidas en el bloque nuevoUno, en el caso que no se encuentre o que la precondición de la regla que se encuentre sea falsa, la búsqueda continua en el bloque *nuevo* y si tampoco se acepta el evento en este bloque, la búsqueda sigue en el bloque *main*.

Si hasta allí no se manejo el evento, entonces se buscará responder al evento con el comportamiento definido en la superclase, como esta no tiene el bloque *nuevoUno*, ni el bloque *nuevo*, la búsqueda comienza por el primer bloque que engloba a los dos últimos bloques mencionados, o sea el bloque 'main' de la superclase.

Si en este último bloque no se atiende el evento, se seguirá un esquema similar en las superclases de la clase X, hasta llegar al tope de la jerarquía, si esto ocurre y no se activó ninguna regla, el evento se rechazará.

Esto último quiere decir que, si en el código del objeto que envía el evento, se consulta la variable *ok* luego de enviado el evento, esta tendrá el valor falso, esto siempre y cuando el evento haya sido enviado en forma sincrónica.

Siempre, la llegada de un evento a un objeto, genera la activación de una regla a lo sumo, una vez que la regla se ejecutó, se interrumpe la búsqueda.

Si en el comportamiento definido por el usuario se implementa una regla cuyo evento es el mismo que uno de los predefinidos y la precondición, para esta regla, cuando llega este evento es true, se ejecutará la lista de acciones del evento programado y no se ejecutará el comportamiento anteriormente descripto de estos eventos.

3.4.2.6.2.2 - Encabezado

El encabezado de una regla esta conformado por el nombre y los parámetros que la regla puede recibir.

El nombre representa un evento ante el cual la entidad deberá responder en caso de que la precondición de la regla sea verdadera.

Luego, en la lista de parámetros se reciben los valores de variables o expresiones que el objeto que genera el evento envía al receptor.

Dos reglas tienen el mismo encabezado siempre y cuando el nombre del evento a que responden sea el mismo, no importando que las dos reglas tengan distinto numero de parámetros.

Si el objeto que envía el evento coloca en la llamada menos parámetros de los que la regla en el objeto receptor tiene, entonces, los parámetros restantes en el receptor tomaran el valor nulo correspondiente al tipo de estos.

Ejemplo:

```
( mover, puede, ...
( mover( d : integer ), not puede, ....
```

Cuando el objeto que implemente estas dos reglas reciba el evento mover, y la variable puede sea true, ejecutara la primer regla, la cual no espera parámetros, en caso de que la variable puede sea false, se ejecutara la segunda, la cual espera un parámetro entero, que en caso de que el objeto llamante no lo haya especificado, tomara el valor 0 (cero).

Cuando en el encabezado se especifica un parámetro como *var*, significa que si se modifica en la lista de acciones internas el parámetro, este cambio se verá en el llamante, siempre y cuando este último haya enviado como parámetro una variable local, un parámetro de la regla o un atributo interno o común.

Ejemplo :

Supongamos tener la siguiente regla :

(getCount(var c : integer), true, c := count; , NULL)

Y que la lista de acciones de otro objeto haga lo siguiente:

LOCAL cuenta : integer;
objeto<–getCount(cuenta);

La variable *cuenta* del objeto llamador se instanciará con el valor del atributo *count* del objeto que recibe el mensaje.

Observar que el envío del evento se hace de manera sincrónica, ya que en caso contrario, el valor de cuenta sería indefinido luego del envío del evento.

3.4.2.6.2.3 - Precondiciones.

La precondición consiste en una expresión cuyo tipo resultante debe ser boolean.

La precondición indica si, ante la llegada del evento correspondiente al encabezado de la regla, esta puede ejecutar la lista de acciones internas.

Cuando la precondición es falsa, no quiere decir que la regla se rechaza sino que se sigue buscando otra regla con el mismo encabezado en los bloques que engloban al actual o en las superclases.

Cuando la precondición es true, se acepta el evento y se interrumpe la búsqueda de las reglas, ejecutándose entonces la lista de acciones internas y la precondición de la regla encontrada.

Ejemplos :

(mover(d : integer), d > 10, ...)

(encender, not encendida, encendida := true; , NULL)

En este segundo ejemplo suponemos que *encendida* es un atributo de la clase que implementa la regla.

3.4.2.6.2.4 - Acciones internas.

La lista de acciones internas define el comportamiento del objeto ante la llegada de un evento destinado a este.

Si, por razones de implementación, no se necesita hacer en la regla acción alguna, entonces se debe especificar esto en esta sección de la regla, esto se hace mediante la cláusula NULL.

Ejemplo:

(mover, true, NULL, NULL)

En este caso lo que se quiere hacer es que, cuando la búsqueda del evento igual al que llego pase por esta regla, no haga ninguna accion. Esto es útil para inhabilitar comportamientos definidos en alguna superclase que no se quiere que herede la subclase.

Ahora veremos las diferentes clases de sentencias con que se pueden especificar las acciones internas en las reglas.

3.4.2.6.2.5 - Variables locales

En esta sección de la regla se especifica como va a responder el objeto cuando se acepte el evento que coincida con el encabezado de la regla. Como se vió mas arriba, la lista de acciones internas consiste en una declaración de variables locales y luego una lista de sentencias ejecutables.

Los nombres de las variables no pueden ser iguales a nombres de atributos, conexiones y otras declaraciones de la entidad, no deben ser nombres de clases ya definidas ni palabras reservadas del lenguaje, etc. Las variables locales declaradas son creadas y tiene vida dentro de la lista de acciones internas y dejan de existir cuando esta termina, por lo que no conservan su valor. Hasta que no son asignadas el valor que poseen es indefinido, aunque es uno de los posibles según su tipo, excepto si el tipo es hObject.

Estas variables no son visibles desde otro lugar que no sea la lista de acciones internas en la que se declaran.

Para declarar variables locales de distinto tipo en la misma lista de acciones internas, repetir la cláusula LOCAL.

Ejemplo de declaración de variables locales :

```
( mover, not fixed,
        LOCAL contador, n : Integer;
        LOCAL unObjeto : hObject; ...
```

3.4.2.6.2.6 - Conjunto de sentencias.

Un conjunto de sentencias es una construcción que consisten en una o mas sentencias entre paréntesis, el conjunto puede considerarse como una sentencia atómica.

Cada una de las sentencias internas al conjunto debe terminar con punto y coma incluso la última.

Ejemplo:

```
( robot<--inicializar; a := a + 1; robot<--activar( a ); );
```

En un conjunto de sentencias no debe incluirse una declaración de variables locales.

3.4.2.6.2.7 - Envío de eventos.

Esta es la sentencia que permite la interacción entre objetos de la hiperhistoria.

Con esta sentencia, desde la lista de acciones internas se puede enviar un evento desde un objeto a otro o hacia la representación del objeto en la interface.

Existen dos modos de enviar un evento:

- sincrónico : el evento es procesado por el objeto receptor en el mismo momento en que este es enviado y al terminar, se continúa la ejecución de la lista de acciones del objeto que envía el evento.
- asincrónico : el evento es enviado a una cola de espera y la ejecución de la lista de acciones internas del objeto continúa. El evento enviado será recibido por el objeto destino luego, según la posición de este en la cola de eventos.

Para simular concurrencia, se debería implementar aquellos eventos que no requieren respuesta inmediata en forma

asincrónica, ya que al hacerlo de otra manera, se determina en el momento de la implementación el orden de ejecución.

Ejemplos :

Supongamos querer enviar un evento al contexto en el que estoy incluido, por ejemplo, *iluminar*, se haría de la siguiente manera:

```
myContext<==iluminar;
```

Este modo de mandarlo (símbolo <==) es el asincrónico, por lo tanto el contexto recibirá el evento, al menos, después de terminada la ejecución de la regla actual.

Si necesito especificar parámetros, puedo hacer lo siguiente, supongamos que le envío el evento a un robot:

```
robot<==mover( 10, direccion );
```

En este caso, el objeto que recibe el evento, si tiene especificados los parámetros, los podrá usar dentro de la lista de acciones internas de la regla que corresponda al evento enviado.

En este ultimo caso, el 10, como es una constante, se supone que es de tipo *integer*, si sabemos que el destino espera como primer parámetro un *byte* u otro tipo, debemos indicar que el parámetro se interprete como tal, para el ejemplo haremos lo siguiente:

```
robot<==mover( asByte( 10 ), direccion );
```

AsByte es una construcción que permite indicar con que tipo se quiere interpretar un parámetro en el envío de un evento.

Los constructores para hacer este tipo de interpretaciones se usan para tipos numéricos y son:

- asByte : interpretar la expresión como *byte*.
- asWord : interpretar la expresión como *word*.
- AsInteger : interpretar la expresión como *integer*.
- AsLongint : interpretar la expresión como *longint*

El objeto receptor del evento debe ser una especificación de conexión, un atributo (común o interno), un parámetro o una variable local de tipo hObject, el objeto TIMER o el objeto INTERFACE.

Ejemplos:

```
INTERFACE<--luz( asWord( uno + dos ) );
```

Esta sentencia envía el evento *luz* hacia la representación en la interface del objeto que implemente esta sentencia, con un parámetro de tipo *word* que es la suma de los atributos *uno* y *dos* que suponemos definidos en la clase.

Por ejemplo para decirle al timer que cada diez unidades de tiempo envíe al objeto corriente el evento *mover* debo hacer lo siguiente:

```
TIMER<--setEvent( 'mover', 10, true );
```

Mas adelante explicaremos de manera detallada a que mensajes responde el timer y como funciona.

Luego de ejecutada la sentencia del envío de evento, el atributo *ok* toma el valor true en caso de ser un envío asincrónico o un envío sincrónico en el que el objeto destino acepta el evento.

En caso de que el destino rechace el evento, si el envío es sincrónico, *ok* toma el valor false.
Los eventos enviados al timer o a la interface son siempre aceptados, por lo tanto *ok* siempre reporta el valor true.

Ejemplo:

```
robot<--activar;
if ok then ( ... );
```

Los eventos a la interface se pueden generar explícitamente mediante el envío al objeto INTERFACE o, de manera implícita, cuando se cambia el valor de un atributo común de un objeto.
En este último caso el objeto que representa, en la interface, al que envía el evento recibe como nombre de evento, el nombre del atributo común que cambio de valor y, como único parámetro, el valor del atributo.

Ejemplo:

Supongamos que tenemos la declaración siguiente:

```
ATTRIBUTES abierta : boolean;
INITIALIZATION abierta := false;
```

Y luego en la implementación de una regla se hace lo siguiente:

```
abierta := not abierta;
```

Entonces, el objeto que representa en la interface al que implementa esto, recibe el evento *abierta* con un parámetro booleano que es el nuevo valor del atributo luego de ejecutada la asignación.

3.4.2.6.2.8 - Asignación

Esta sentencia es usada para dar valor a atributos, parámetros y variables locales.
Al asignar tipos simples como integer, string, etc. el operador actúa por copia, es decir el valor de la variable asignada puede ser alterado sin afectar al valor original, suponiendo que a una variable se le haya asignado el valor de otra.
En caso de asignar una variable del tipo hObject a otra, en realidad las dos quedan como referencias al mismo objeto, por lo tanto es indistinto enviarle un evento a través de una referencia o de otra, ya que el objeto que recibe el evento es el mismo.

Ejemplos :

Supongamos que el atributo *cuenta* tiene el valor 10 y se hace lo siguiente:

```
LOCAL aux : integer;
aux := cuenta;
aux := 5;
```

Luego de estas sentencias el valor de *aux* es 5 pero *cuenta* conserva el valor 10.

En caso de tener, por ejemplo, un atributo *objeto* que sea del tipo hObject y que los contextos reciban un evento *setCount* con el que asignan un contador interno, y un evento *getCount*, el cual permite extraer el valor de este, y hacemos:

```
LOCAL aux : integer;
objeto := myContext;
```

```
objeto<--setCount( 10 );
myContext<--getCount( aux );
```

En este caso, aux tendrá el valor 10, ya que al asignar *myContext* a *objeto*, esta variable hace referencia al mismo contexto que *myContext*.

Observar que si el envío del evento *setCount* se hiciera de manera asincrónica, este no se ejecutaría, al menos, hasta que no terminara el procesamiento de la lista de acciones actual, por lo tanto el valor que se obtuviera con *getCount* no el asignado por *setCount* en la línea anterior, sino el que tuviera asignado el contexto en ese momento.

3.4.2.6.2.9 - Ejecución condicional.

Esta sentencia permite que, dada una condición booleana, se pueda decidir ejecutar o no una o más sentencias o elegir entre dos sentencias o conjuntos de sentencias.

Ejemplo:

Supongamos que enviamos un evento de manera sincrónica y queremos, en caso que sea aceptado incrementar el contador *count*, entonces debemos hacer como sigue:

```
robot<--mover;
if ok then count := count + 1;
```

En caso de querer hacer mas de una acción cuando se cumpla la condición usamos el constructor de conjunto de sentencias:

```
if ok then
    ( count := count + 1;
    INTERFACE<--acepto; );
```

En caso de querer tomar una acción cuando la condición es verdadera, se tiene la cláusula *else*, entonces se hace como sigue:

```
if ok then
    ( count := count + 1;
    INTERFACE<--acepto; );
    else
    ( count := count - 1;
    INTERFACE<--rechazo; );
```

Los If's se pueden anidar uno dentro del otro, pero el else siempre corresponderá al if mas cercano declarado.

3.4.2.6.2.10 - Postcondiciones.

Esta sección de la regla se utiliza para activar bloques que estaban desactivados, o sea, que las reglas en ellos no podían ser ejecutadas, y al mismo tiempo desactivar otros. Las reglas de activación de bloques se explicaron mas arriba, en esta sección de la regla se pueden especificar dos cosas:

- **nombre de un bloque** : con lo cual se activa el bloque con ese nombre y se desactiva el bloque que contiene a la regla.
- **END** : con lo que se desactiva el bloque corriente y se desactiva el siguiente, si es que existe, si no existe, se activa el siguiente al bloque que contiene a este ultimo y así sucesivamente.

Los bloques deben agrupar reglas que definen un comportamiento para el objeto definido, el cual se cambiara, parcial o

totalmente, cuando se active otro bloque.

Ejemplos:

En primer lugar veremos un ejemplo de activación de bloques invocándolos por su nombre.

```
BEGIN main
    BLOCK active
        ( swichActive, true, NULL, inactive )
        ...
    END;
    BLOCK Inactive
        ( swichActive, true, NULL, active )
        ...
    END;
END;
```

En el comportamiento aquí definido se tienen dos bloques, uno para cuando la entidad esta activa y otro par cuando el estado es inactivo, cada vez que la entidad recibe el evento *swichActive*, cambia de estado de activo a inactivo o viceversa.

Cada bloque puede tener mas reglas que las especificadas en el ejemplo, pero solo las de un bloque por vez estarán activas en un momento dado, de modo que un evento pueda dispararlas.

Ahora veremos un ejemplo de activación de bloques mediante el uso de la cláusula END.

```
BEGIN main
    BLOCK Initial
        ( startRun, true, NULL, END )
        ...
    END;
    BLOCK running
        ...
    END;
END;
```

En este caso, el comportamiento de la entidad se inicia con la activación del bloque *initial* segun lo visto en las reglas de activación de bloques, con esto, se le buscara respuesta a los eventos que reciba en este bloque y en el bloque *main*, como así también en los bloques correspondientes de la superclase, si es que la hubiera; esto hasta que reciba el evento *startRun*, en ese momento se desactiva el bloque *initial* para activarse el bloque *running* con lo cual, los eventos que reciba a partir de ese momento serán buscados en las reglas incluidas dentro de este ultimo bloque (y de los bloques que lo contienen y en la superclase, por supuesto).

Nótese que si alguna subclase, de la entidad que implemente este comportamiento, agrega un bloque entre el bloque *initial* y el bloque *running*, será este el que se active en lugar de *running*.

3.4.2.6.3 Uso de las conexiones.

Las conexiones son un modo de dejar pendiente una referencia a otro objeto en la declaración de una entidad, esta referencia será hecha luego, cuando en la inicialización de algún contexto que incluya una instancia del objeto que esta siendo definido se use esta conexión para comunicar al objeto con otro o con un canal.

El motivo de esto es permitir definir un comportamiento tal que envíe eventos, a través de la conexión, a un objeto todavía inexistente y, luego, ligarlo con el objeto o canal que se desea que reciba los eventos que el objeto genere.

Un ejemplo sencillo es el de una llave de luz, esta debe enviar una indicación, a los objetos que tiene conectados, de que se

enciendan o se apaguen, pero estos objetos no tienen por que estar definidos al momento de construir la llave, sino que esta se hace y luego se le conectaran los objetos que la llave deba encender.

Veamos la declaración de una llave de luz usando conexiones:

```
CLASS clase_llave IS A ENTITY
    CONNECTIONS output;
    ATTRIBUTES encendida : BOOLEAN;
    BEHAVIOR
    BLOCK main
        ( enciende, not encendida, encendida := true; output<==encender, NULL )
        ( apaga, encendida, encendida := false; output<==apagar, NULL )
END;
```

En este ejemplo, cada instancia de *clase_llave* cuando reciba el evento *enciende,* enviará a la salida el evento *encender* y cuando reciba el evento *apaga*, hará lo mismo con el evento *apagar.* Luego, cuando se conecte otro objeto o un canal a esta conexión, este recibirá estos eventos.

3.4.2.6.4 - Ejemplo de implementación de una entidad.

Ahora veremos como se utilizan los conceptos anteriormente vertidos para la definición de entidades que intervendrán en la hiperhistoria, para esto, veremos varios ejemplos de construcción de clases de entidades.

Los objetos de la hiperhistoria forman una jerarquía, el objeto tope de esta jerarquía es la entidad, para definir un objeto en la hiperhistoria que no sea ni contexto ni link, deberá declararse como subclase de entidad o de una subclase de entidad.

Puede que en cada hiperhistoria todas las entidades definidas en ella tengan propiedades comunes, además de las predefinidas, que son necesarias, para ello, se debe definir una clase, que siendo subclase de entidad, sea la superclase de todas las entidades de la hiperhistoria, por ejemplo, como sigue:

```
CLASS ENTIDADBASE is an ENTITY
    BEHAVIOR
    BLOCK main
        ( queSoy, true, INTERFACE<--declr( 'Soy un/a ' + icl ); ,NULL )
END;
```

En este ejemplo, la entidad básica definida responde, además de los predefinidos, a un evento definido por el usuario, *queSoy*, cuya precondición es true, por lo tanto, siempre que una entidad lo reciba, será aceptado.

En la lista de acciones se implementa el envío, a la interface, del evento *decir* con el string *'Soy un/a '* y el nombre de la clase, como parámetro.

La postcondición de esta regla, por ser *NULL*, no genera cambio en la estructura de bloques activos de la entidad.

Con esta implementación, cualquier subclase de esta clase que reciba el evento *queSoy* enviará, si no se redefinió el evento en esa subclase o en otras en el camino de búsqueda, a la representación del objeto definido en la interface, el evento *decir* con la cadena mencionada como parámetro, luego, la interface debe reaccionar a la recepción de este evento con un comportamiento adecuado, el cual deberá mostrar al usuario de la hiperhistoria, a través de una representación adecuada al estilo de la interface, el valor parámetro pasado o alguna acción que represente el evento.

Supongamos ahora que en la hiperhistoria tenemos dos clases de objetos, aquellos que tienen un comportamiento autónomo y aquellos que pueden ser tomados por el personaje o algún otro objeto.

Para esto debemos definir dos subclases de la *entidadBase*, en caso, claro está, que las nuevas clases quieran conservar el funcionamiento de la primera.

Veremos primero un ejemplo de definición de una clase que será la tope para todas aquellas clases que representen objetos que tengan comportamiento autónomo:

```
CLASS ENTAUTONOMA is an ENTIDADBASE
    BEHAVIOR
    BLOCK main
        ( mover( T : word ), true, TIMER <== setEvent( 'oMover', T, true );,NULL )
        ( detener, true, TIMER <== killEvent( 'oMover' );, NULL )
        ( dormir( T : word ), true, TIMER <== setEvent( 'oDespertar', T, false);, NULL )
END;
```

En este ejemplo, se define una clase cuyas entidades responderán, además de los heredados, a tres eventos, *mover, detener y dormir.*

El primero de los eventos, *mover*, toma un word como parámetro, como la precondición es true, siempre aceptará este evento, y al hacerlo, indicará al timer que, a partir de ahora, envíe al objeto el evento *oMover* cada T unidades de tiempo, este evento puede ser invocado desde dentro de una subclase del objeto de la siguiente manera:

```
self <== mover( 10 );
```

Desde otra clase que pueda enviarle eventos a esta (supongamos que la variable *objeto* sea la referencia), se haría de la siguiente manera :

```
objeto <== mover( 10 );
```

Desde la interface se pueden enviar eventos directamente a un objeto de la hiperhistoria, pero esto será transparente a la implementación en este nivel, los objetos que reciban eventos desde la interface se determinarán en la implementación de esta y en su unión con la hiperhistoria.

El envío de estos eventos, como cualquiera definido por el usuario, puede ser asincrónico, como en el ejemplo, o sincrónico, esto es porque los parámetros de los eventos no retornan el valor al objeto que envía el evento y no es necesaria, a priori, la ejecución de estos antes de que termine la lista de acciones de la regla ejecutada en el objeto que envía el evento.

El comportamiento del timer y la cantidad y tipo de los parámetros que recibe se explicarán detalladamente mas adelante, aquí sólo interesa su efecto de modo ilustrativo.

Con la recepción del evento *detener*, se hace que el timer deje de enviar al objeto el evento *oMover*, la manera de enviar este evento es la misma que para mover.

Por último, el evento *dormir* hace que el timer envíe, luego de un tiempo T, el evento *oDespertar* al objeto, esto lo hace solo una vez.

Veamos ahora una implementación para una superclase de objetos que podrán ser tomados por otras entidades:

```
CLASS OBJTOMABLE is an ENTIDADBASE
    BEHAVIOR
    BLOCK main
        ( tomar, true,    self <--takeBy( sender ); INTERFACE <-- meToman;, NULL )
        ( soltar, true,   self <-- leaveBy( sender ); INTERFACE<--meSueltan;, NULL )
END;
```

3.4.3 - Links.

3.4.3.1 - Definición.

Los links son aquellos objetos de la hiperhistoria que ligan dos o mas contextos, permitiendo que una entidad, cuyo comportamiento así lo permita, pase, a través de ellos de un contexto a otro.

Los links, como subclases de las entidades heredan todas las propiedades de estas, mas el comportamiento que se agrega, propio de los links.

3.4.3.2 - Instanciación.

La instanciación de los links se produce cuando estos son declarados como internos dentro del contexto en el que serán usados.

Ejemplo:

```
CLASS casa IS A CONTEXT
    CONTEXTS
        hall, pasillo IS A ambiente;
    LINKS
        puertaPasillo IS A puerta;
    INITIALIZATION
        hall LINKED WITH puertaPasillo;
        pasillo LINKED WITH puertaPasillo;
END;
```

En este ejemplo, se crea un link llamado *puertaPasillo*, el cual, como se determina en la sección de inicialización, une los contextos *hall* y *pasillo*. ; también se puso lo básico de la declaración de un contexto para ejemplificar la inicialización de un link, una descripción completa del uso de los contextos será hallada mas adelante.

3.4.3.3 - Atributos predefinidos.

Los link tiene atributos predefinidos, heredados algunos de las entidades, con el mismo significado, otros, cuyo significado cambia o no son validos y algunos nuevos, exclusivos de los links. Los siguientes son atributos que hereda de las entidades con el mismo significado que en las entidades:

Id : string
Nombre con que se declara el link en el contexto que lo usa.

Icl : string
Nombre de la clase del link.

Sender : hObject
Objeto que envía el mensaje.

Ok : boolean
True si el último evento enviado fue aceptado.

MyContext : hObject
Contexto en el cual esta incluido el objeto.

Self : hObject
Referencia a si mismo.

El atributo *owner* no tiene sentido para los links ya que estos no pueden ser tomados por las entidades, por lo tanto su valor es siempre *NULL*.. Los siguientes atributos son específicos de los links:

FstContext : integer
Indice, en la colección interna, del primer contexto que comunica el link.

SndContext : integer
Es el Indice, en la colección interna, del segundo contexto comunicado por el link.

Estos dos números se utilizan para determinar cuales contextos comunica cada link.

El numero correspondiente a cada contexto depende del orden con que se fue ligando el link a los contextos, el primer contexto ligado tendrá el numero cero, el segundo el uno, etc.

Ejemplo:

```
CLASS edificio IS A CONTEXT
    CONTEXTS
        PB, piso1, piso2, piso3 IS A ambiente;
        cajaAsc IS A ambiente;
    LINKS
        puertaAsc IS A clasePuertaAsc;
    INITIALIZATION
        cajaAsc LINKED WITH puertaAsc;
        PB LINKED WITH puertaAsc;
        piso1 LINKED WITH puertaAsc;
        piso2 LINKED WITH puertaAsc;
        piso3 LINKED WITH puertaAsc;
END;
```

En el ejemplo, a *cajaAscensor* le corresponde el Indice cero en la colección interna del link *puertaAscensor*, a *plantaBaja* el uno, a *piso1* el dos y así sucesivamente.

En el momento que se quiera que un link cambie los contextos que comunica, simplemente se deberá cambiar, en el comportamiento del link, los valores los atributos *fstContext* y *sndContext*, los cuales comienzan, al iniciarse la hiperhistoria, inicializados con los valores 0 y 1 respectivamente.

En caso de que los dos atributos tomen el mismo valor, al cruzar, una entidad, el link no afectaría su ubicación dentro de la estructura de navegación.

3.4.3.4 - Comportamiento básico predefinido.

Los links, como las entidades, tienen un comportamiento predefinido además del que hereda de la superclase *ENTITY*, el cual permite hacer que un objeto pase, a través de ellos, de un contexto a otro.

El comportamiento heredado de la clase *ENTITY* es el siguiente:

GetName(var name : string)
Retorna en el parámetro el nombre del link receptor del evento, este nombre es el que se indica cuando se declara el link dentro de un contexto, sin incluir los nombres de los contextos externos.

GetClass(var class : string)
Retorna en el parámetro en nombre de la clase del link receptor del evento.

GetContext(var cont : hObject)
Retorna el contexto en el cual esta incluido el link receptor del evento, este contexto no es ninguno de los que comunica el link sino el que engloba a todos ellos.

Los siguientes eventos son para acceder a la colección interna que tiene cada link, la cual hereda de las entidades :

ObjFirst
Hace que el contexto corriente de la colección sea el primero.

ObjNext
Hace que el contexto corriente sea el siguiente en la colección.

ObjCurr(var obj : hObject)
Retorna en el parámetro el contexto corriente en la colección.

ObjIsLast(var last : boolean)
Determina si el contexto corriente en la colección es el último elemento de la misma.

En el caso de los links, los objetos a los que se acceden a través de estas funciones son los contextos que comunica el link receptor de los eventos.

Los eventos getOwner, takeBy y LeaveBy, no tienen sentido para los links ya que estos no pueden ser tomados por ninguna entidad, por lo tanto, el comportamiento en caso de que se los envíe es indefinido.

Los siguientes eventos se agregan a los heredados, estos son exclusivos de los links:

CrossBy(obj : hObject)

Cuando un link recibe este evento, cruza al objeto especificado en obj del contexto en el cual está hacia el otro con el que está comunicado en ese momento.

FContext(var obj : hObject)

Obtiene el contexto, en obj, que es apuntado por el atributo FstContext.

SContext(var obj : hObject)

Obtiene el contexto, en obj, que es apuntado por el atributo SndContext.

El evento *crossBy* genera dos nuevos eventos, uno hacia el contexto del cual sale el objeto que cruza: *entityOut(link : hObject)*, el cual informa al contexto origen que un objeto salió de el a través del link indicado en el parámetro.

El otro evento es *entityIn(link : hObject)*, el cual informa al contexto destino que un objeto ingresó en él a través del link indicado en el parámetro. En ambos eventos en la variable *sender* se encuentra una referencia al objeto que cruzó.

Estos dos eventos son asincrónicos y no son manejados, de manera predefinida, en los contextos ni en ninguna de sus superclases, solo están como notificación en caso de que la aplicación lo requiera.

La precondición para los eventos antes mencionados es siempre *true*, la lista de acciones es la que permite ejecutar el comportamiento descripto mas arriba y la postcondición es siempre *NULL*.

Todos los eventos del comportamiento predefinido anteriormente citados deben ser enviados de manera sincrónica para que tengan el efecto mencionado.

En caso de que sean enviados de manera asincrónica, no se podrá asegurar que tengan el funcionamiento especificado ya que no se puede determinar en el momento que se despacharán, además, el valor del parámetro de salida, aquellos que lo posean, será indeterminado.

3.4.3.5 - Comportamiento definido por el usuario.

La manera en la cual el usuario puede definir comportamiento especifico para cada clase de link que desarrolle es la misma que en las entidades, es decir, a través de bloques y reglas.

3.4.3.6 - Ejemplo de implementación de un link.

El siguiente es un ejemplo de como se debe implementar un link que permita, simplemente, que una entidad lo atraviese para pasar de un contexto a otro, sin restricciones.

```
CLASS LinkBase IS A LINK
    BEHAVIOR
    BLOCK main
        ( cruzar, true, self<--crossBy( sender );, NULL )
END;
```

En este caso, al recibir una instancia del link el evento *cruzar*, cruza a la entidad que se lo envía desde el contexto en el cual se encontraba al otro contexto comunicado por el link.

Notar que es este caso, el link aceptara siempre el evento *cruzar*, en caso de no querer esto de debe especificar una condición, por ejemplo para implementar una puerta:

```
CLASS puerta IS A LINK
    ATTRIBUTTES abierta : boolean;
    BEHAVIOR
    BLOCK main
        ( abrir, not abierta, abierta := true;, NULL )
        ( cerrar, abierta, abierta := false;, NULL )
        ( cruzar, abierta, self<--crossBy( sender );, NULL )
END;
```

En este caso, una entidad, al enviarle el evento cruzar al link, lo cruzará efectivamente siempre y cuando el atributo *abierta* sea verdadero, en caso contrario, el cruce no se efectuará.

Notar que como el atributo *abierta* no es interno, la interface será notificada en caso de que este cambie de valor, por lo tanto la representación, en la interface, de la puerta debe variar según reciba el evento*abierta* con parámetro *true o false*.

Mas tarde veremos como se notificará a la interface del cambio de contexto de la entidad que cruzó el link.

3.4.4 - Contextos.

3.4.4.1 - Definición.

Los contextos son los objetos de la hiperhistoria que pueden ser habitados por las entidades, y los que, a través de los links, se comunican para formar la estructura de navegación de la hiperhistoria.

La clase CONTEXT, que es la clase base para la definición de contextos en la hiperhistoria, es subclase de ENTITY, por lo que hereda todos sus atributos y maneja todos los eventos predefinidos a los que responde esta última, con interpretaciones especiales que hacen a las características especiales de los contextos.

La declaración de una clase de contexto crea una "plantilla" a partir del cual se crearán instancias de ambientes simples o de estructuras de navegación mas complejas junto con el comportamiento de cada link y las entidades que estén incluidas en cada uno de los contextos que forman la estructura. También en esta declaración se especifican las conexiones entre entidades internas al contexto y los atributos y comportamiento del contexto.

3.4.4.2 - Contextos internos.

Cada contexto es instanciado cada vez que se declara como interno a otro contexto, excepto el contexto mas externo, el cual tiene una sola instancia y debe ser declarado en último lugar.

Veamos el siguiente ejemplo que define una casa simple, para lo cual definimos primero una clase de contexto que llamamos *cuarto* que tiene el atributo *iluminado* y el comportamiento suficiente como para modificarlo a través del evento *cambialluminar*. A esta clase la usaremos como un ambiente standard para la definición de los contextos internos de la casa.

```
CLASS cuarto IS A CONTEXT
    ATTRIBUTES
        iluminado:BOOLEAN;
    INITIALIZATION
        iluminado:=false;
    BEHAVIOR
    BLOCK main
        (cambialluminar,true,iluminado:=not iluminado;,NULL)
END;
```

Hasta aquí no se produjo la instanciación del ningún cuarto, solo definimos la clase que nos servirá de molde para la definición de la casa, que lo hacemos como sigue.

```
CLASS casa IS A CONTEXT
    CONTEXTS
        living, pieza, cocina IS A cuarto;
    LINKS
        puertaPieza, puertaCocina IS A puerta;
    INITIALIZATION
        living LINKED WITH puertaPieza,puertaCocina;
        pieza LINKED WITH puertaPieza;
        cocina LINKED WITH puertaCocina;
END;
```

En esta segunda declaración definimos la casa, la cual tiene tres ambientes internos, cuya clase es *cuarto,* en este momento se instanciaron los contextos *living*, *pieza* y *cocina* y luego, en la parte de instanciación, se los liga en forma adecuada con los links para construir la estructura de navegación de la casa.

Los nombres de los contextos internos serán, a partir de ahora, los indicados aquí en la declaración, mientras que el contexto mas externo, la casa, conservara este nombre, para este ultimo contexto, no será necesario hacer una declaración como para los contextos internos.

3.4.4.3 - Links internos.

Los links definidos son internos al contexto que los contiene a ellos y a todos los contextos que el link comunica, en el ejemplo anterior, el link *puertaPieza* y el link *puertaCocina* son internos al contexto *casa*, esta consideración se hace en referencia a la determinación de como se nombrará a estos con una especificación de objeto, por ejemplo, suponiendo que la casa fuera el contexto mas externo, la especificación completa del link *puertaPieza* seria *casa.puertaPieza.*

Los contextos, como las entidades que son su superclase, tienen una colección de objetos internos; en el caso de la casa, en esta colección estarán incluidos los contextos *living, pieza* y *cocina,* en cambio, la colección de algún contexto que no tenga contextos internos y este comunicado a través de links, contendrá, además de las entidades que lo habiten, los links a través de los que se comunica con otros contextos; por ejemplo, el la colección del contexto *living* contendrá referencias a los links *puertaPieza* y *puertaCocina*.

3.4.4.4 - Atributos de los contextos.

Los contextos, como subclase de la clase ENTITY, heredan de esta todos sus atributos, algunos de los cuales tienen el mismo significado que en esta y otros no tienen sentido en estos.

Los atributos heredados y su interpretación en los contextos son los siguientes:

Id : string
Nombre con que se declara el contexto en el contexto que lo usa.

Icl : string
Nombre de la clase del contexto.

Sender : hObject
Objeto que envía el mensaje.

Ok : boolean
Resultado del envio del ultimo evento.

MyContext : hObject
Contexto en el cual esta incluido.

Self : hObject
Referencia a si mismo.

El atributo *owner* no tiene sentido para los contextos ya que estos no pueden ser tomados por las entidades, por lo tanto su

valor es siempre *NULL*.

Los contextos no tienen otros atributos específicos además de los heredados.

3.4.4.5 - Comportamiento básico predefinido.

Los contextos, como las entidades, tienen un comportamiento predefinido además del que hereda de la superclase *ENTITY*, el cual es el siguiente:

GetName(var name : string)

Retorna en el parámetro el nombre del contexto receptor del evento, este nombre es el que se indica cuando se declara un contexto dentro de otro o el nombre de la clase del contexto mas externo, sin incluir los nombres de los contextos externos.

GetClass(var class : string)

Retorna en el parámetro en nombre de la clase del contexto receptor del evento.

GetContext(var cont : hObject)

Retorna el contexto en el cual esta incluido el contexto receptor del evento, si es el mas externo, retorna NULL.

Los siguientes eventos son para acceder a la colección interna que tiene cada contexto, la cual hereda de las entidades:

ObjFirst

Hace que el objeto corriente de la colección sea el primero.

ObjNext

Hace que el objeto corriente sea el siguiente en la colección.

ObjCurr(var obj : hObject)

Retorna en el parámetro el objeto corriente en la colección.

ObjIsLast(var last : boolean)

Determina si el objeto corriente en la colección es el ultimo elemento de la misma.

En el caso de los contexto, los objetos a los que se acceden a través de estas funciones son las entidades que habitan el contexto en cada momento y los links con que cada contexto se comunica con otros, y en el caso de contextos que solo tienen contextos y links en su interior, la colección contiene a estos.

Los eventos getOwner, takeBy y LeaveBy, no tienen sentido para los contextos ya que estos no pueden ser tomados por ninguna entidad, por lo tanto, el comportamiento en caso de que se los envíe es indefinido.

La precondición para los eventos antes mencionados es siempre *true*, la lista de acciones es la que permite ejecutar el comportamiento descripto mas arriba y la postcondición es siempre *NULL*.

Todos los eventos del comportamiento predefinido anteriormente citados deben ser enviados de manera sincrónica para que tengan el efecto mencionado.

En caso de que sean enviados de manera sincrónica, no se podrá asegurar que tengan el funcionamiento especificado ya que no se puede determinar en el momento que se despacharán, además, el valor del parámetro de salida, aquellos que lo posean, será indeterminado.

Los contextos no manejan otros eventos predefinidos que no sean los ya descriptos, pero tienen un comportamiento especial si reciben eventos que no sean estos predefinidos y no sean manejados por eventos de usuario, este comportamiento consiste en reenviar el evento recibido, de manera asincrónica, hacia todos los objetos habitantes del contexto.

3.4.4.6 - Comunicación entre contextos internos.

Para determinar que contextos se comunican con que otros, se debe utilizar la cláusula *LINKED WITH*, la cual permite especificar la ligazón entre contextos y links.

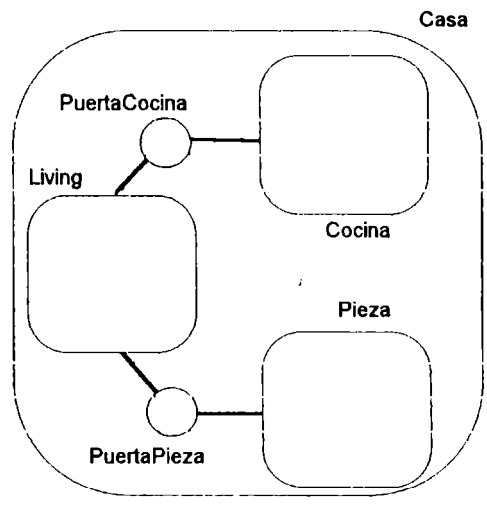
Veamos el ejemplo de la casa anteriormente descripta, en la cual tenemos la siguiente declaración en primer lugar:

living LINKED WITH puertaPieza,puertaCocina;

Con esta declaración indicamos que el contexto *living* estará comunicado con otros contextos (todavía no especificamos cuales), a través de los links *puertaPieza* y *puertaCocina*, luego completamos la declaración ligando estos dos links con los otros contextos con los cuales se comunican:

pieza LINKED WITH puertaPieza;
cocina LINKED WITH puertaCocina;

Al terminar esta declaración tenemos una estructura que la podemos ver gráficamente a la derecha:



3.4.4.7 - Entradas a un contexto.

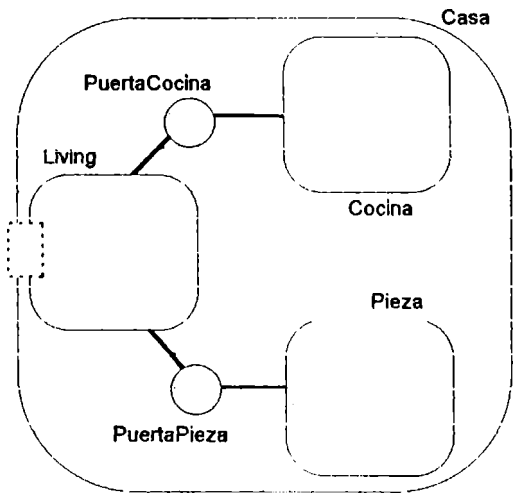
Si vemos el caso anterior, una entidad podrá navegar los contextos definidos de la casa solamente, ya que esta no tiene ninguna comunicación con el exterior, por lo tanto quedaría aislada en caso de que se incluyera la casa, como contexto interno, en una estructura de navegación mas complicada.

Para solucionar esto existe el concepto de entrada a un contexto, es decir, cuando se tiene una estructura de navegación compleja (con contextos internos) y es necesario que a una entidad se le permita pasar desde un contexto externo al definido a uno de sus contextos internos y viceversa.

Agreguemos, entonces, a la casa previamente definida una entrada, para especificar a cual de los objetos internos debe ingresar a la casa o desde cual contexto interno una entidad puede salir de ella, para esto reescribimos la definición del contexto *casa* previamente declarada:

```
CLASS casa IS A CONTEXT
    ENTRYS entrada;
    CONTEXTS
        living, pieza, cocina IS A cuarto;
    LINKS
        puertaPieza, puertaCocina IS A puerta;
    INITIALIZATION
        living LINKED WITH entrada,puertaPieza,puertaCocina;
        pieza LINKED WITH puertaPieza;
        cocina LINKED WITH puertaCocina;
END;
```

En el ejemplo se agrega la cláusula ENTRYS con la que se especifica una ligadura entre la casa y el contexto living, que es interno a ella, luego, cuando la casa sea incluida dentro de otro contexto y se la comunique, a través de un link, con otro contexto, en realidad, dicho link comunicará este último contexto con el contexto living, por lo tanto, una entidad que atraviese el link mencionado, en realidad ingresará o saldrá del living hacia o desde el contexto externo a la casa con el cual esta este comunicada. Veamos, a la derecha, un gráfico de como quedaría representada la casa.



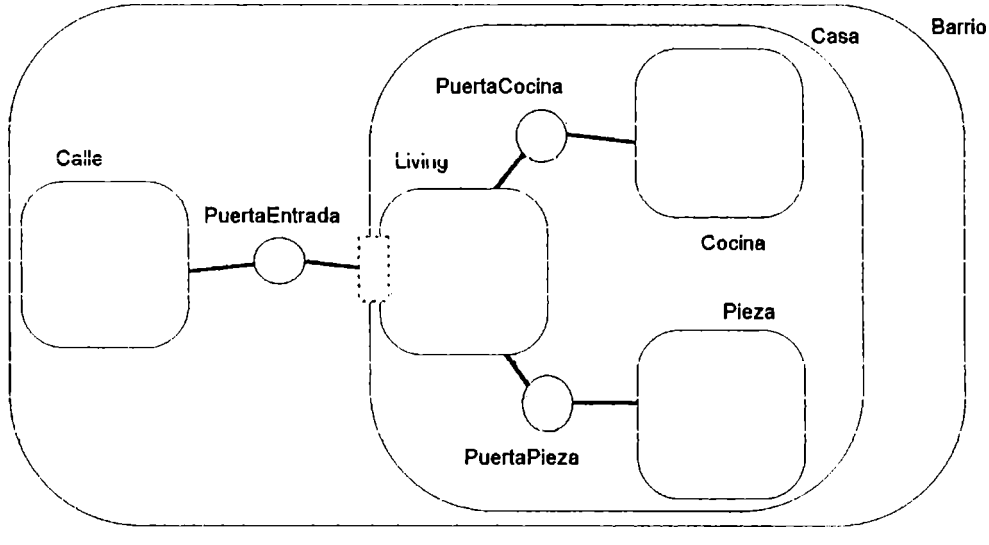
El rectángulo de líneas punteadas representa la entrada de la casa, esta comunica directamente del desde el exterior de esta al contexto interno *living* de la casa.

Ahora veamos un ejemplo del uso de la casa como contexto interno de otro que llamaremos barrio y que incluye, además del contexto *casa*, otro que denominamos *calle*.

```
CLASS barrio IS A CONTEXT
    CONTEXTS
        calle IS A CONTEXT;
        unaCasa IS A casa;
    LINKS
        puertaCasa IS A puerta;
    INITIALIZATION
        calle LINKED WITH puertaCasa;
        unaCasa LINKED WITH puertaCasa;
END;
```

Aquí, la comunicación del link *puertaCasa* con el contexto *calle* se hace como se describió anteriormente, en cambio, al comunicar el link con el contexto unaCasa, en realidad se esta ligando este al contexto *living*, esto porque la entrada que le definimos a la casa hace que en el momento de realizar la comunicación, se haga referencia al contexto interno mencionado en lugar de la casa.

El siguiente gráfico representa como queda la estructura de navegación para esta última declaración:



Esto funciona así aun si se agregan mas entradas a la cláusula ENTRYS, cada vez que se efectúe una ligazón con el contexto que especifique las entradas, se ligará a un contexto interno, si se liga un contexto más veces que la cantidad de

entradas que posee, las comunicaciones que resten se comportaran de la manera común, es decir, ingresaran al contexto pero no lo harán a ningún contexto interno a él.

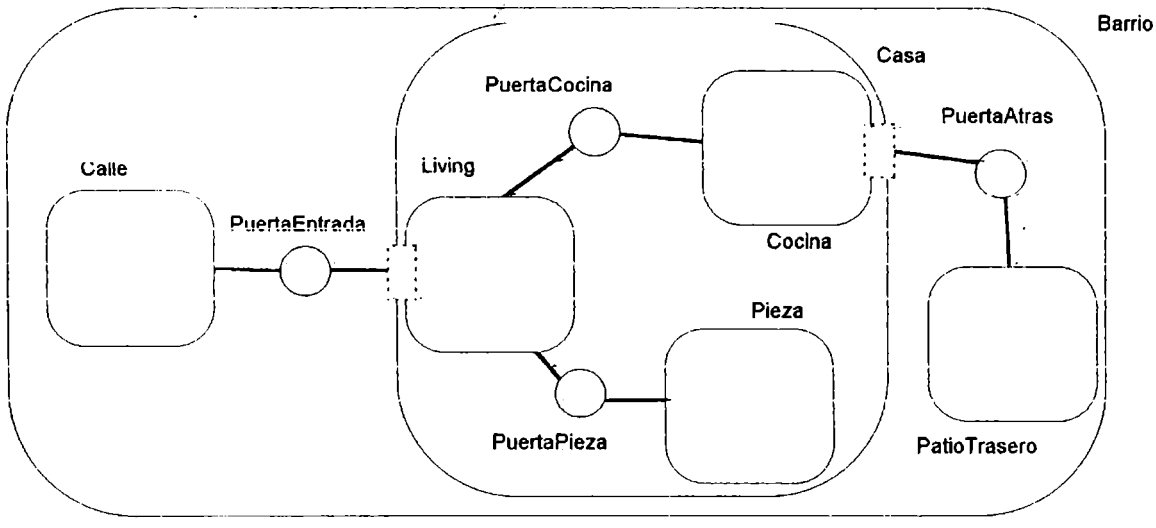
Veremos un ejemplo de esto, para lo cual redefiniremos la casa agregando mas entradas:


```
CLASS casa IS A CONTEXT
    ENTRYS entradapri, entradapos;
    CONTEXTS
        living, pieza, cocina IS A cuarto;
    LINKS
        puertaPieza, puertaCocina IS A puerta;
    INITIALIZATION
        living LINKED WITH entradapri,puertaPieza,puertaCocina;
        pieza LINKED WITH puertaPieza;
        cocina LINKED WITH puertaCocina,entradapos;
END;
```

Y supongamos que hacemos la inclusión de la casa en el barrio de la siguiente manera:

```
CLASS barrio  IS A CONTEXT
    CONTEXTS
        calle, patioTrasero IS A CONTEXT;
        unaCasa IS A casa;
    LINKS
        puertaCasa, puertaAtras IS A puerta;
    INITIALIZATION
        calle LINKED WITH puertaCasa;
        unaCasa LINKED WITH puertaCasa, puertaAtras;
        patioTrasero LINKED WITH puertaAtras;
END;
```

En este caso la estructura de navegación quedará como indica el siguiente gráfico:



3.4.4.8 - Herencia de la estructura de navegación.

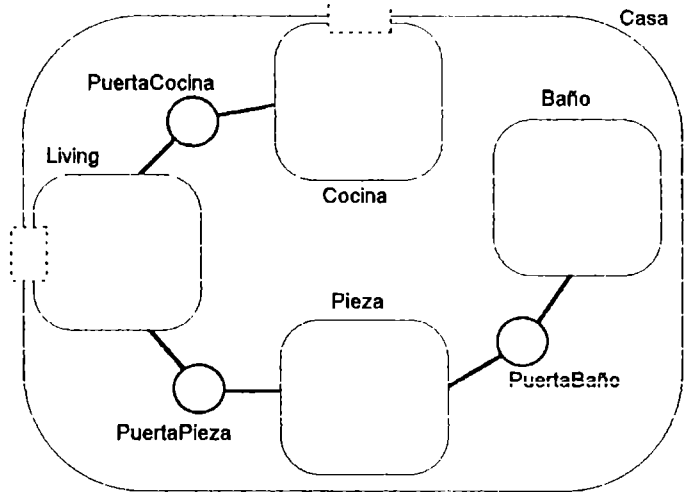
Cuando se define un contexto que es subclase de otra ya definida, la cual tiene declarados contextos o links internos, el primero hereda toda la estructura de navegación de la superclase, permitiéndose agregar a esta nuevos contextos y links internos (además de los atributos y el comportamiento), los cuales extiendan la estructura de navegación.

El siguiente es un ejemplo de una subclase de la casa anteriormente definida, esta subclase agrega un contexto interno mas con su respectivo link para conectarlo a los contextos existentes.

```
CLASS casaBaño IS A casa
    CONTEXTS
        baño IS A cuarto;
    LINKS
        puertaBaño IS A puerta;
    INITIALIZATION
        pieza LINKED WITH puertaBaño;
        baño LINKED WITH puertaBaño;
END;
```

Esta última declaración crea una nueva casa que es una subclase del contexto *casa*, se extiende en este caso la estructura de la misma agregándole un contexto denominado *baño*.

El gráfico de la derecha muestra como queda representada la nueva subclase.



No es necesario es todas las subclases agregar ambos, contextos y links, se puede agregara solo un link y unir dos contextos ya definidos o agregar solo un contexto y unirlo, a través de algún link ya definido, con otro contexto ya definido, este seria el caso de definir un edificio con un número determinado de pisos y luego agregar un piso mas en una subclase de este contexto.

3.4.4.9 - Inclusión de objetos.

La cláusula de inclusión de objetos es utilizada para indicar que objetos serán incluidos en que contextos, los objetos deberán estar definidos en conjuntos previamente declarados, tomemos por ejemplo los siguientes conjuntos:

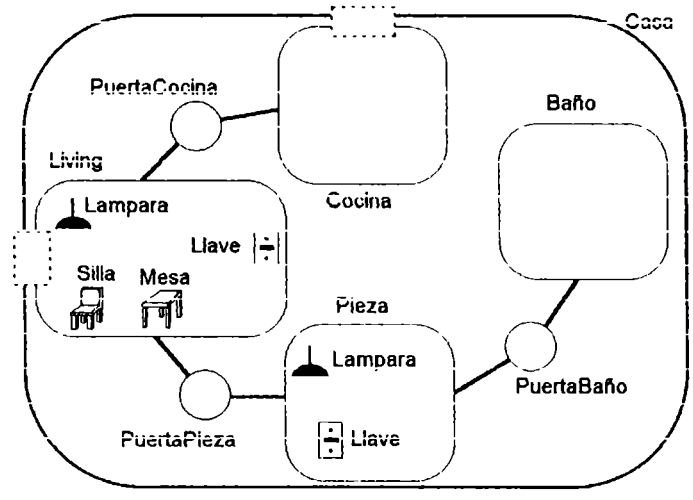
```
SET muebles
    silla IS A clase_silla;
    mesa IS A clase_mesa;
END;

SET luces
        lampara IS A clase_lampara;
        llave IS A clase_llave;
    INITIALIZATION  lampara.encendida := false;
END;
```

Luego incluiremos estos objetos en la casa precedente de manera que en el contexto *living* haya una lámpara con la llave que la encienda y una mesa y una silla y que en el contexto *pieza* haya otra lámpara con su correspondiente llave, para esto, en la sección de inicialización del contexto *casa* debemos incluir las siguientes declaraciones:

living INCLUDES muebles + luces;
pieza INCLUDES luces;

Con la primer cláusula, se le incluyen al contexto *living* los muebles y las luces, con lo cual quedan en su interior cuatro entidades que son las mencionadas arriba, en tanto que en el contexto *pieza*, se incluyen sólo las luces, las cuales, a pesar de que se utilice el mismo conjunto para incluirlas son objetos independientes de los del contexto *living*.

El gráfico a la derecha muestra como queda la casa con los contextos internos y las entidades incluídas en ellos.



3.4.4.10 - Conexionado de objetos.

Dentro de los contextos se indican que objetos estarán conectados con que otros, usando las conexiones definidas en estos, directamente o a través de canales, los cuales se deben definir en los sets e incluirse en contextos como cualquier otra entidad.

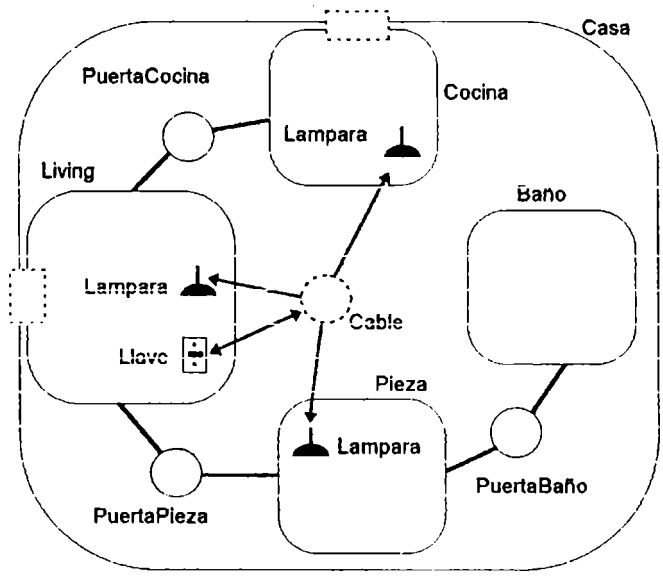
La cláusula usada para efectuar esta inicialización es CONNECT WITH, en la cual se deben indicar los objetos involucrados y, si existiera, la especificación de la conexión del objeto que será usada para la operación.

Veremos un ejemplo de esto, supongamos tener la estructura de navegación de la casa anterior sin entidades en su interior y querer que, con una sola llave ubicada en el living, se enciendan lámparas, una en el mismo living, otra en la cocina, y otra en la pieza, para esto debemos definir como primer lugar los siguientes conjuntos :

```
SET cLlave
    llave IS A clase_llave;
END;

SET cLamp
    lampara IS A clase_lampara;
    INITIALIZATION
        lampara.encendida := false;
END;

SET elCable
    cable IS A CHANNEL;
END;
```

Con estos conjuntos definimos los objetos necesarios para la estructura que queremos construir, ahora debemos utilizarlos dentro de la declaración de la casa como sigue:

```
CLASS casaBaño IS A casa
    CONTEXTS
        baño IS A cuarto;
    LINKS
        puertaBaño IS A puerta,
    INITIALIZATION
        pieza LINKED WITH puertaBaño;
        baño LINKED WITH puertaBaño;

        SELF INCLUDES unCable;
        living INCLUDES cLlave+cLamp;
        cocina INCLUDES cLampara;
        pieza INCLUDES cLampara;

        living.llave CONNECT salida WITH cable;
        living.lampara CONNECT WITH cable;
        cocina.lampara  CONNECT WITH cable;
        pieza.lampara CONNECT WITH cable;
END;
```



Con la cláusula *SELF INCLUDES unCable* hacemos que el cable que unirá a las tres lámparas y a la llave se instancia dentro de la casa pero fuera de cualquiera de los contextos internos a ella, ya que no pertenece a ninguno de ellos exclusivamente. Luego en el living instanciamos la llave y una lampara y instanciamos dos lámparas mas una en la cocina y otra en la pieza y luego unimos todo a través del canal definido.

Al lado de las declaraciones vemos un gráfico en el que se representa las conexiones establecida en estas, el circulo en el medio representa el canal y las flechas que salen, que a los objetos a los que le llegan esas flechas recibirán los eventos que reciba el conector.

Notar que la llave esta unida al canal por una flecha doble, esto significa que, como la llave tiene una conexión, además de recibir eventos, puede enviarlos hacia el canal para que los reciban los demás objetos conectados a el.

En el ejemplo, cuando se acciona la llave, esta envía los eventos *encender* o *apagar*, según el evento que esta haya recibido, y el canal los reenvía a todos los objetos conectados a el excepto al generador del evento, en este caso la *llave*. Por lo tanto las tres lámparas conectadas recibirán estos eventos.

Con este ejemplo finalizamos la definición y descripción del lenguaje o formalismo que se construyó para desarrollar las hiperhistorias, en el próximo capítulo veremos como será la implementación que se genera a partir de este lenguaje.

CAPITULO IV

IMPLEMENTACION

En el presente capítulo se describirá como será la implementación de las hiperhistorias que se generen a partir del modelo descripto en el capítulo anterior, esta implementación se refiere a la aplicación que se generará y que será lo que se ejecute para que funcione la hiperhistoria, mas adelante, en capítulos posteriores se verán detalles acerca del proceso de traducción del modelo a la implementación y de la interface.

4.1 - Arquitectura del sistema.

El sistema se dividirá en varios componentes funcionales, enumeraremos seguidamente cada uno de ellos:

Kernel.

Es el componente principal de la hiperhistoria, el que recibe mensajes de la interface y los despacha hacia los objetos de la hiperhistoria, el que rutea los eventos entre objetos de la hiperhistoria y que envía los eventos que los objetos generan hacia la interface.

Cola de eventos.

Es una estructura de datos en la cual se encolarán los eventos que lleguen de la interface, los eventos del timer y los eventos que los objetos envíen de manera asincrónica, esta cola será de política FIFO y asegurará la concurrencia de eventos en el tiempo.

Timer.

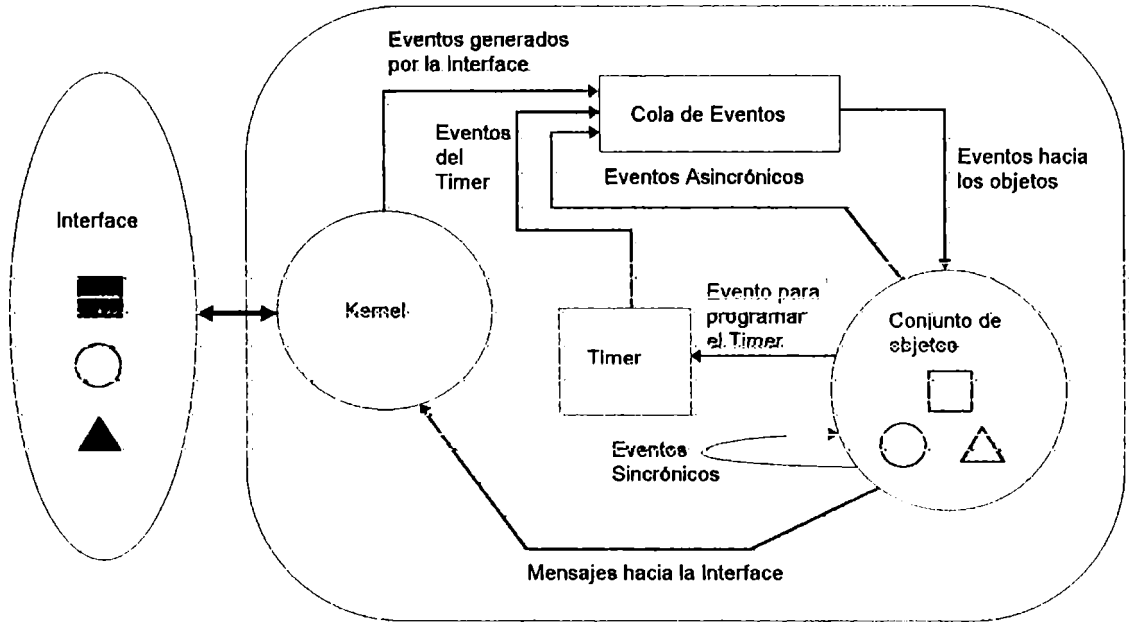
A este componente se lo invocará desde la sección de comportamiento de los objetos para hacer que estos reciban eventos a ciertos intervalos regulares de tiempo, el objeto le deberá indicar al timer que evento quiere que le envíe y cada cuanto tiempo, etc.

Conjunto de objetos.

Son los que el programador define e interconecta para implementar la hiperhistoria, este conjunto esta formado por los contextos, las entidades, los links y los canales, la cantidad de objetos dentro de este conjunto es variable y dependerá de la hiperhistoria que se implemente.

Todos estos componentes forman la aplicación central de la hiperhistoria, esta se deberá comunicar luego con una o mas interfaces con el objeto de captar estímulos externos que hagan que se modifique el curso de la historia y para informar al usuario de la misma acerca de su estado interno.

En el siguiente gráfico mostramos esta arquitectura del Kernel de la hiperhistoria ejecutándose y los posibles caminos que pueden tomar los eventos que se produzcan entre las diferentes componentes de la aplicación:



La comunicación con la interface se realizará mediante el envío de eventos, los cuales tendrán una identificación, conocida por la interface y por la implementación de la hiperhistoria, del objeto al cual se dirige el mensaje y uno o mas valores que serán el mensaje que se envía y los posibles parámetros que el mensaje lleve.

Ahora veremos con mas detalle la manera en que cada uno de los componentes antes mencionados son implementados.

4.2 - Kernel.

Es el componente encargado de comunicarse con la interface, pasándole a esta los cambios que se produzcan en la historia y recibiendo y ruteando a los objetos correspondientes, a través de la cola, los eventos que la interface mande a la historia y los eventos que se envíen entre objetos dentro de la hiperhistoria.

4.2.1 - Atributos.

A continuación veremos una descripción de los atributos internos del kernel.

elMensaje : word

En este atributo se almacena la identificación del mensaje que será utilizado para el intercambio de eventos entre la interface y la aplicación, esta identificación se determina cuando se inicia la primera de las componentes de la hiperhistoria (ya sea la aplicación en sí o alguna de sus interfaces) y permanece igual durante toda la ejecución de la hiperhistoria.

allObjects : pCollection

Colección indexada en la cual almacenan referencias a los objetos (entidades, links, contextos y canales) que compondrán la hiperhistoria, el numero de índice dentro de la colección se corresponde con la identificación interna que cada objeto de la hiperhistoria posee.

4.2.2 - Métodos.

Los métodos a los cuales responde el Kernel son los siguientes:

_init

Función de creación e inicialización del Kernel

addObject(anObject : hObject): word

Parámetros:

anObject : hObject Objeto de la hiperhistoria a agregar en la colección interna del Kernel

Acción:

Agrega un objeto de la hiperhistoria a la colección interna del Kernel.

Retorna:

La identificación interna dada por el Kernel, que es el índice en de la colección interna del nuevo objeto.

Este método se ejecuta por cada objeto de la hiperhistoria definido por el programador en el momento en que la hiperhistoria se inicializa.

La Identificación que retorna este método es usada para referirse al objeto, a lo largo de toda la hiperhistoria, en la comunicación con las interfaces.

atObject(intId : word):hObject

Parámetros:

intId : word Identificación interna del objeto.

Acción:

Nada

Retorna:

Referencia al objeto correspondiente a la identificación *intId* pasada como parámetro o NIL en caso de que sea una identificación invalida.

Esta función la usan la cola de eventos y el Kernel cuando deben despachar eventos a objetos de la hiperhistoria.

userMessage(hmem : tHandle)

Parámetros:

hmem : tHandle Handle del área de memoria donde se pasan los datos del evento.

Acción:

Recepciona un mensaje enviado desde alguna de las interfaces.

Retorna:

Este método no retorna nada.

Este método lo ejecuta el mismo Kernel desde el loop principal de procesamiento cuando recibe el mensaje de Windows que le dice que una de las interfaces mando un evento.

El procedimiento userMessage lo que hace es leer la información de la memoria y pasarla a la cola de despacho de eventos para su posterior procesamiento.

sendToInter(iid:word; ev,par:string; cpar:word): boolean

Parámetros:

iid : word Identificación interna del objeto de la hiperhistoria cuya representación en la interface debe recibir el mensaje.

ev : string Nombre del evento.

par : string Cadena con los parámetros enviados con el evento.

cpar : word Cantidad de parámetros enviados.

Acción

Envía un evento hacia las interfaces activas que haya en ese momento

Retorna:

True en caso de que el mensaje sea enviado a la interface de manera exitosa.

Este mensaje almacena en un área de memoria los datos recibidos y envía a la o las interfaces un mensaje Windows con el handler de dicha área como parámetro word.

La interface debe hacer el paso inverso, o sea, al recibir el mensaje de Windows con el que se comunica con el Kernel, debe extraer del área de memoria la información del evento y enviársela al objeto de interface que se corresponda con la identificación interna del objeto de la hiperhistoria que genero este mensaje.

conecta: boolean

Parámetros:

Este método no recibe parámetros.

Acción:

Detecta si hay alguna interface activa o espera por ello.

Retorna:

True en caso de que se haya efectuado la conexión con una o mas interfaces, False en caso de que la aplicación Kernel termine sin que se haya podido conectar.

Este mensaje lo invoca el mismo Kernel, cuando se le indica que se inicie la hiperhistoria, para efectuar la conexión con las interfaces que pueda haber activas en ese momento, en caso de que haya alguna o, en caso de no ser así, quedar a la espera de que alguna interface arranque.

start

Parámetros:

Este método no recibe parámetros.

Acción:

Inicia la ejecución de la hiperhistoria.

Retorna:

Este método no retorna valor alguno, y termina cuando el usuario hace que termine el task del Kernel de la hiperhistoria.

Método con el cual se inicia la hiperhistoria, en este, mediante el método *conecta* intenta iniciar la comunicación con una o más interfaces y si esto sucede, inicia el ciclo principal de la hiperhistoria.

Este ciclo consiste en obtener los mensajes que le llegan a la aplicación Kernel y procesarlos según su naturaleza: si llega un mensaje de la interface, ejecuta el método *userMessage* del mismo Kernel., si llega un mensaje del timer, ejecuta el método *tick* de este y para los demás mensajes que lleguen, los traslada para que los procese Windows.

4.3 - Cola de Eventos.

Este componente esta para asegurar que todos los hilos de control simultáneos que haya en la hiperhistoria se ejecuten de manera intercalada y no uno detrás de otro, ya que cuando se envían mensajes asincrónicos, estos pasan por la cola antes de ser despachados hacia el objeto que los debe recibir.

Veremos un ejemplo de esto, supongamos que tenemos dos secuencias de eventos iniciadas al mismo tiempo por el timer.

Supongamos que la primer secuencia de eventos es la siguiente

A<==mover

B<==inicIar

C<==cerrar

Esto significa que dentro del comportamiento para la regla mover del objeto A se envía el evento iniciar al objeto B y así sucesivamente.

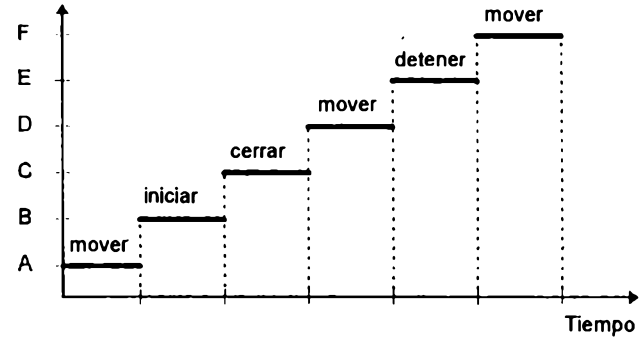
Luego supongamos tener la segunda secuencia de control como sigue:

D<==mover

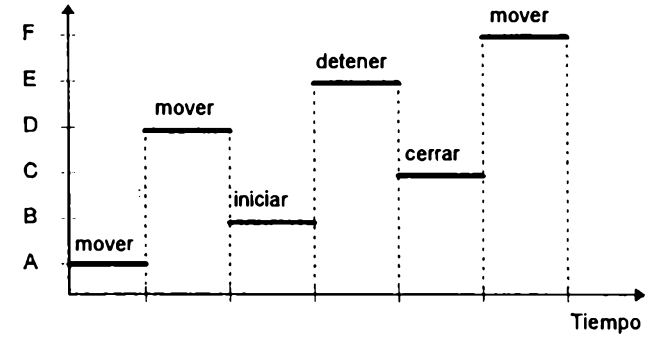
E<==detener

F<==mover

Luego supongamos que el timer envía a los objetos A y D el evento mover en el mismo instante de tiempo, sin la cola de envío de eventos, la ejecución de las dos secuencias de mensajes en el tiempo sería la el que se ve a la derecha, en el que se ve que la primera secuencia de eventos se ejecuta sola y luego se ejecuta la segunda.



En cambio, con la cola de envío de eventos, los mensajes del timer no van directamente a los objetos que deben recibirlos sino que son encolados, supongamos que primero encole el evento *mover* dirigido al objeto A y luego el evento *mover* dirigido al objeto D con lo cual el objeto A recibirá el evento *mover* y al ejecutar el comportamiento encolará el evento *iniciar* dirigido al objeto B, pero este último evento será encolado después del evento *mover* dirigido al objeto D, por lo tanto el próximo evento a ejecutarse será este, lo que hará que, al efectuar el comportamiento definido encole el evento *detener* en la cola, y así sucesivamente.



El gráfico de como sería la ejecución de la segunda secuencia, en la cual envían los eventos a través de la cola, sería el que aparece a la derecha, en este se pone de manifiesto que los eventos se irán ejecutando de manera intercalada, y en primer lugar los que genero el timer y luego siguiendo por cada una de los eventos de las secuencias en forma solapada.

Los mensajes se envían a través de la cola de eventos si en el comportamiento de los objetos se los envía de manera asincrónica (símbolo <==), en caso de enviarlos de manera sincrónica (símbolo <--) la secuencia de ejecución sería como la indicada en el primer gráfico.

A la cola de eventos llegan todos los eventos generados por el timer, por el usuario a través de la interface y aquellos que son enviados entre objetos de la hiperhistoria de manera asincrónica.

La cola de eventos despacha aquellos eventos que tiene encolados solo en las siguientes circunstancias:

- Al iniciar la hiperhistoria y luego de que el Kernel encole los eventos *sys_init* para los objetos que los necesiten.
- Cuando llega un mensaje desde alguna de las interfaces a las cuales esta conectada la hiperhistoria, luego de que este mensaje es encolado.
- Por cada ciclo de reloj del timer y luego de que este encole los eventos que corresponda enviar en ese ciclo.

Con esto se ve que cuando un objeto de la hiperhistoria envíe un evento, en forma asincrónica, a otro objeto, este último no lo recibirá inmediatamente sino que este será encolado y luego será despachado a su turno cuando llegue un tick del reloj o cuando llegue un mensaje desde la interface, lo que ocurra primero.

4.3.1 - Atributos.

La cola tiene un atributo interno que es una colección de eventos pendientes y dos métodos, uno para encolar los eventos,

uno por uno, y otro para vaciar la cola, despachando los eventos que se encuentren encolados hacia los objetos destinatarios de ellos.

4.3.2 - Métodos.

pushEvent(destination:hObject; _event:_tEvent; sender:hObject; _params:string): boolean;

Parámetros:

destination : hObject	Referencia al objeto destinatario del evento.
_event : _tEvent	Nombre del evento enviado al objeto.
sender : hObject	Referencia al objeto que envía el evento.
_params : string	Cadena con los parámetros del evento.

Acción

Este método encola un evento en el último lugar, a esta función la invoca el Kernel cuando recibe un mensaje de la interface, los objetos de la hiperhistoria cuando envían un evento de manera asincrónica y el timer al encolar eventos cuando corresponda.

Retorna:

True si la operación se efectuó con éxito.

flush

Parámetros

Este método no recibe parámetros.

Acción:

Este método vacía la cola de eventos, enviando a los objetos correspondientes los eventos encolados, esto no lo hace con los nuevos eventos que se pudieran encolar como consecuencia de la ejecución del comportamiento de alguno de los eventos enviados, estos últimos serán enviados la próxima vez que la cola se vacíe

Retorna

Este método no retorna valor alguno.

4.4 - Timer.

El timer es el componente de la hiperhistoria que provee temporización al sistema, cada objeto de la hiperhistoria puede invocarlo para que este, luego, le envíe mensajes a intervalos regulares, con los cuales el objeto en cuestión realizara cierta actividad autónoma.

La programación del timer se hace de la manera ya descripta en la semántica del lenguaje, aquí veremos la estructura interna de este.

La actualización del estado del timer se lleva a cabo a intervalos regulares, que llamaremos pulsos, con estos el timer actualiza sus tablas internas y envía los eventos a los objetos que corresponda.

La duración de un pulso es mayor que el tiempo mínimo entre ticks que puede proporcionar el timer de Windows.

4.4.1 - Atributos.

El timer tiene los siguientes atributos:

Items : pCollection

Es una tabla que tiene una entrada por cada objeto y por cada evento con que se programe al timer, cada ítem de la tabla tendrá la siguente estructura.

aObject : hObject

Referencia al objeto destino del evento.

reqEvent : string

Evento que se le enviará a ese objeto.

delay : longint

Cantidad de pulsos que el timer espera para enviar el evento.

count : longint

Cantidad de pulsos que faltan para enviar el evento, este se envía cuando count llega a cero.

forEver : boolean

Si este atributo es falso, cuando count llegue a cero, se eliminará la entrada de la tabla del timer, si es verdadero, lo que se hace es asignar a count la cantidad indicada en delay.

hopeTick : longint

Valor en ticks que debería tener el tiempo del sistema al invocarse el procedimiento tick, si es menor que el real entonces el timer acelerará la marcha hasta que los dos valores se emparejen.

vHigh : boolean

True si el timer está funcionando a velocidad alta, es decir, si está recuperando el tiempo o false si funciona a velocidad normal.

timerId : word

handler del timer de Windows usado.

4.4.2 - Métodos.

El objeto timer tiene los siguientes métodos:

tick

Parámetros

Este método no recibe parámetros.

Acción

Decrementa la cuenta de todos los eventos registrados en el timer y a los que le llegue a cero los envía a los objetos correspondientes.

Retorna

Este método no retorna valor alguno.

Cuando le llega este mensaje, a intervalos regulares, que llamaremos pulsos, el timer resta uno a la cuenta de todos los eventos programados en su tabla interna y, en aquellos en los cuales la cuenta sea cero, son enviados a los objetos correspondientes.

Luego si el atributo *forEver* de esa entrada es false, elimina la entrada, si no asigna a count lo indicado por el atributo *delay* de esa entrada.

Este procedimiento también mantiene al timer funcionando en tiempo real, es decir, cada vez que se lo invoca almacena el tiempo del sistema, y si, entre dos invocaciones sucesivas a esta función, pasa mas tiempo del que tendría que haber pasado (tiempo Indicado por el atributo *hopeTick*), se reprograma el timer de Windows para que este envíe su mensaje mas rápido, con lo cual la función *tick* será invocada con mas frecuencia, esto hasta que alcance al tiempo esperado, en ese momento restaura la frecuencia del timer de Windows a la normal.

setEvent(anObject : hObject; anEvent : _tEvent; aDelay : longint; isForever : boolean): boolean

Parámetros:

anObject : hObject	Referencia al objeto que será destinatario de los eventos.
anEvent : _tEvent	Nombre del evento a enviar
aDelay : longint	Intervalo de tiempo que debe esperar entre cada evento que envíe.
isForever : boolean	True si se quiere que el evento sea enviado en forma continua, false si el evento debe enviarse una sola vez y detenerse.

Acción

Agrega una nueva entrada a la tabla del timer.

Retorna

True si la operación de agregar la entrada a la tabla se realizó con éxito.

Este método se invoca cuando, en la lista de acciones internas de una regla en la implementación del comportamiento de un objeto de la hiperhistoria se envía al timer el evento setEvent, ejemplos:

TIMER<--setEvent(despertar, 20, false);

Aquí se le enviará al objeto el evento despertar a 20 pulsos a partir del momento del envío del evento y luego no se enviará más.

TIMER<--setEvent(cambio, 10, true);

Aquí se le enviara al objeto el evento cambio cada 10 pulsos, a partir del momento del envío del evento al timer, esto se hará siempre hasta que acabe la hiperhistoria o explícitamente se lo interrumpa como veremos en la función siguiente.

El evento comenzará a enviarse al objeto especificado a las *aDelay* unidades de tiempo desde el momento en que se programe el timer con ese evento y luego, si el parámetros isForever es true, será enviado cada aDelay unidades de tiempo al objeto especificado.

killEvent(anObject : hObject; anEvent : _tEvent): boolean

Parámetros:

anObject : hObject Objeto para el cual quiere que se detenga el envío de un evento.

anEvent : _tEvent Nombre del evento que no debe seguir enviándose.

Acción

Hace que se detenga el envío del evento especificado al objeto especificado.

Retorna:

True si la operación se concreto con éxito.

Con esta función se elimina explícitamente una entrada de la tabla interna del timer, con esto se logra detener el envío del evento al objeto indicado por la entrada de la tabla que se corresponda con los parámetros recibidos por el método.

Esta función elimina la entrada cuya referencia al objeto destino coincida con anObject y cuyo evento coincida con anEvent.

Este método se invoca cuando, en la lista de acciones internas de una regla en la implementación del comportamiento de un objeto de la hiperhistoria se envía al timer el evento killEvent, ejemplo:

TIMER<--killEvent(cambio);

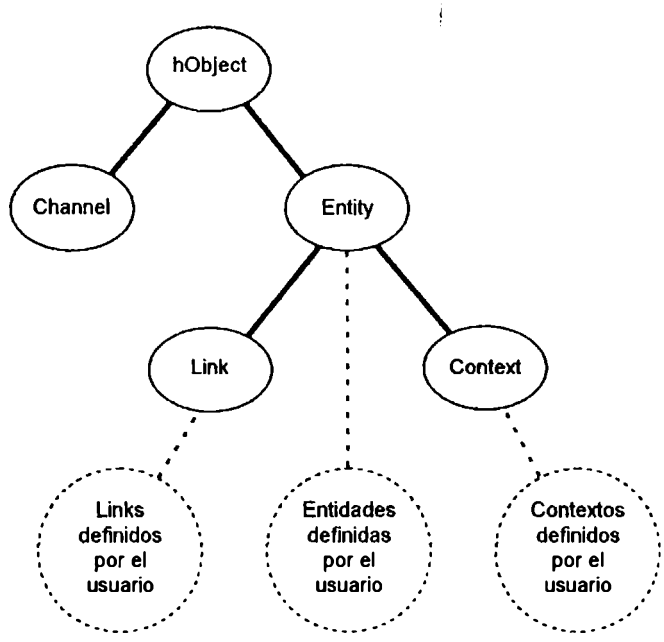
Con esta sentencia se interrumpe, a partir del momento de su ejecución el envío del evento cambio al objeto en cuyo comportamiento de ejecuta.

Esta interrupción no quiere decir que el objeto en cuestión no reciba ningún evento cambio a partir de este momento, ya que alguno de estos puede haber sido encolado antes de la ejecución de la sentencia y el objeto lo recibirá luego, cuando le llegue el turno de ser despachado por la cola.

4.5 - Conjunto de objetos de la hiperhistoria.

Esta formado por un conjunto de objetos los cuales serán instancias de alguna subclase de ENTITY, LINK o CONTEXT o sino instancia de la clase CHANNEL.

La jerarquía básica de clases de los objetos de la hiperhistoria es la siguiente :



Ahora, de aquí en adelante veremos la estructura interna de cada una de estas clases, los métodos que están implementados para cada una de ellas y los principales mecanismos que posee la Implementación de la hiperhistoria para hacer que esta funcione.

4.6 - Clase hObject

Esta es la clase tope en la jerarquía de clases de la hiperhistoria, en ella se implementa el comportamiento básico común a todos los objetos de la hiperhistoria.

4.6.1 - Atributos.

La clase *hObject* tiene los siguientes atributos:

id : string
Es el nombre del objeto dado por el programador de la hiperhistoria.

icl : string
Nombre de la clase a la que pertenece el objeto.

_internalId : word
Identificación interna dada por el Kernel de la hiperhistoria al crearlo

myContext : hObject
Referencia al contexto en el que esta incluído el objeto

_myObjects : pCollection
Colección de los objetos internos del objeto, esta colección se interpretará de diferente manera en cada una de las subclases de hObject.

_indiceCurr : word
Indice dentro de la colección de objetos internos, se usa internamente para la implementación de las rutinas que manejan los objetos internos.

_sendInit : boolean

Este atributo se setea en true cuando alguna instancia de alguna subclase implementa en la parte de comportamiento una regla con el evento *sys_init* como encabezado.

Los atributos que comienzan con el carácter subrayado no son visibles para los objetos de la hiperhistoria dentro del modelo.

4.6.2 - Métodos.

Luego se implementan para esta clase los siguientes métodos:

_init(oid, ocl : string; enContext : hObject);

Parámetros:

oid : string	Nombre del objeto a crearse
ocl : string	Nombre de la clase del objeto
enContext : hObject	Referencia al contexto en el cual estará incluido el objeto a crearse.

Acción:

Crea el objeto, inicializa los atributos *id, icl* y *myContext* con los valores pasados como parámetro, se agrega a la lista de objetos del Kernel y asigna el atributo *_internalId* con la identificación interna que este le asigna, por último asigna los atributos *_needInit* con False, *_myObjects* con una colección vacía e *_indiceCurr* con cero.

Retorna:

Este método no retorna ningún valor.

_needInit(value : boolean);

Parámetros:

value : boolean	True si el objeto acepta el evento *sys_init*, false en caso contrario.

Acción:

Asigna el parámetro al atributo *_sendInit*.

Retorna:

Este método no retorna ningún valor.

_getId: string;

Parámetros:

Este método no tiene parámetros.

Acción:

Nada

Retorna:

El nombre del objeto (el valor del atributo *_id*).

_getMyObjects: pCollection;

Parámetros:

Este método no recibe parámetros

Acción:

Nada

Retorna:

La colección de objetos internos al objeto.

_getExternalContext: hObject; virtual;

Parámetros:

Este método no recibe parámetros.

Acción:

Nada, este método es reimplementado por la subclase ENTITY.

Retorna:

NIL

_getContext: hObject; virtual;

Parámetros:

Este método no recibe parámetros.

Acción:

Nada, este método es reimplementado por las subclase ENTITY

Retorna:

NIL

_internalObject(idObject : string): hObject; virtual;

Parámetros:

idObject : string Especificación de objeto interno, no es usado.

Acción:

Nada, este método es reimplementado por las subclase CONTEXT

Retorna:

A si mismo

_makeConnWith(anObject : hObject); virtual;

Parámetros:

anObject : hObject Objeto con el cual conectarse.

Acción:

Nada, este método es reimplementado por la subclase Channel.

Retorna:

Este método no retorna nada.

_connect(cConx : string; anObject : hObject); virtual;

Parámetros:

cConx : string Nombre de la conexión.

anObject : hObject Objeto de la hiperhistoria con el cual se debe conectar.

Acción:

Nada, este método es reimplementado en las subclases de Entity definidas por el usuario y para las cuales se definan conexiones

Retorna:

No retorna nada.

_addObject(anObject : hObject);

Parámetros:

anObject : hObject Objeto para agregar a la colección interna.

Acción:

Este método agrega el objeto pasado como parámetro a la colección interna de objetos.

Retorna:

Este método no retorna nada.

_deleteObject(anObject : hObject);

Parámetros:

anObject : hObject Objeto a eliminar de la colección interna.

Acción:

Elimina de la colección interna el objeto pasado como parámetro

Retorna:

Este método no retorna nada.

_setContext(anContext : hObject);

Parámetros:

anContext : hObject Contexto en el cual estará incluido el objeto.

Acción:

Asigna con el valor del parámetro el atributo *myContext*, con esto especifica, para el objeto al cual se le ejecute este método, el contexto en el que será incluido.

Retorna:

Este método no retorna nada.

_makeLink(aLink : LINK); virtual;

Parámetros:

aLink : LINK Link con el cual se relacionara el objeto (que deberá ser un contexto)

Acción:

Nada, este método es reimplementado por la subclase Context.

Retorna:

Este método no retorna nada.

_aceptaEvento(_event : _tEvent; sender : hObject;var _params : string):boolean; virtual;

Parámetros:

_event : _tEvent Nombre del evento enviado al objeto.

sender : hObject Objeto que envía el evento.

_params : string Parámetros del evento.

Acción:

Nada, este evento se reimplementa en todas las subclases que tengan algún comportamiento, incluso a las definidas por el usuario.

Retorna

Para la implementación en esta clase siempre retorna false, lo que significa que el evento es rechazado.

4.7 - Clase Channel

Esta clase de objeto se utiliza para realizar una conexión explícita entre dos o mas objetos para que estos puedan enviarse eventos a través del canal.

La forma en que se enviarán los eventos es simplemente enviando un evento al canal, este, por el comportamiento predefinido que tiene, lo retransmitirá a todos los objetos conectados al canal excepto al generador del evento.

4.7.1 - Atributos.

Veremos a continuación los atributos de esta clase, entre paréntesis, antes del nombre del atributo pondremos la superclase de cual lo hereda en caso que no sea declarado en la clase en cuestión:

(hObject) Id : string

Nombre del canal dado por el programador.

(hObject) Icl : string

Siempre tiene el valor 'CHANNEL'.

(hObject) _internalId : word

Identificación interna dada por el Kernel de la hiperhistoria al crearlo

(hObject) myContext : hObject

Referencia al contexto en el que esta incluido el objeto

(hObject) _myObjects : pCollection

Colección de los objetos de la hiperhistoria que están ligados al conector, es decir, a aquellos a los que, al recibir un evento el conector, se los reenviará.

(hObject) _indiceCurr : word

Este atributo tiene el mismo uso que en la superclase.

(hObject) _sendInit : boolean

Siempre False.

Esta clase no tiene atributos nuevos, solo los que hereda de la superclase, lo que varía es el comportamiento.

4.7.2 - Métodos.

Los siguientes son los métodos a los que responden las instancias de esta clase, solo veremos aquellos en los que varía la implementación con respecto a la superclase.

_makeConnWith(anObject : hObject); virtual;

Parámetros:

anObject : hObject Objeto con el cual conectarse.

Acción:

Agrega el objeto pasado como parámetro a la colección interna de objetos del canal, es decir, *myObjects*.

Retorna:

Este método no retorna nada.

_aceptaEvento(_event : _tEvent; sender : hObject; var _params : string):boolean; virtual;

Parámetros:

_event : _tEvent Nombre del evento enviado al objeto.

sender : hObject Objeto que envía el evento.

_params : string Parámetros del evento.

Acción:

Envía a cada objeto de la colección interna, excepto al generador del evento, el evento recibido.

Retorna:

True.

4.8 - Clase Entity

Esta clase, que es subclase de hObject, define los atributos y comportamiento básico que tendrán todos los objetos de la hiperistoria programados por el usuario, incluso el de los contextos y links, los cuales son subclases de ella.

4.8.1 - Atributos.

Los atributos y métodos de esta clase son los siguientes:

(hObject) id : string

Nombre de la entidad dado por el programador.

(hObject) icl : string

Contiene el nombre de la subclase definida por el usuario de la cual el objeto en cuestión es instancia o 'ENTITY' si el objeto es instancia directa de esta clase.

(hObject) _internalId : word

Identificación interna dada por el Kernel de la hiperhistoria al crearlo

(hObject) myContext : hObject

Referencia al contexto en el que esta incluido la objeto

(hObject) _myObjects : pCollection

Colección de los objetos de la hiperhistoria que están tomados por la entidad.

(hObject) _indiceCurr : word

Este atributo tiene el mismo uso que en la superclase.

(hObject) _sendInit : boolean

True o False según la entidad que es subclase implemente en su comportamiento una regla cuyo encabezado es *sys_init*.

owner : hObject

Referencia al objeto de la hiperhistoria que es "dueño" de la entidad, este atributo toma valor cuando otra entidad toma a esta.

_bloques : dictionary

Este diccionario contiene como claves los nombres de los bloques definidos en el comportamiento de la clase.

_hijos : pIntCollection

Colección de enteros que representa el primer bloque anidado en cada bloque de comportamiento.

A cada bloque le corresponde una posición en la colección, la cual se corresponde con las posiciones de los nombres de los bloques en el atributo anterior, el valor que tiene la colección en esa posición es el número del primer bloque anidado en el bloque correspondiente a esa posición.

En caso de que un bloque no tenga bloques anidados el valor en la colección, en la posición correspondiente a ese bloque es -1.

_padres : pIntCollection

Colección de enteros que representa, por cada bloque, el numero del bloque que lo engloba, esta colección tiene la misma cantidad de elementos que la anterior, ya que cada elemento se corresponde con un bloque.

En caso del bloque mas externo, que no tiene bloque que lo englobe, el valor correspondiente a su posición es -1.

_siguientes : pIntCollection

Colección de enteros que representa el siguiente bloque, dentro del mismo nivel de anidamiento y siempre que el bloque inmediatamente superior al bloque siguiente sea el mismo que el actual.

En caso del último bloque, el valor que toma la posición correspondiente a esta es -1.

La cantidad de elementos de esta colección siempre será igual que las dos anteriores.

_flags : pBoolCollection

Colección de valores lógicos que indica si los bloques correspondientes a la cada posición están o no activos.

Estas cuatro colecciones vistas se usan para mantener la estructura de bloques del comportamiento de cada objeto de la hiperhistoria implementado por el usuario.

Estas colecciones serán inicializadas por cada objeto de la hiperhistoria que implemente un comportamiento y a la inicialización agregarán los datos de los bloques de la superclase correspondiente, luego serán usadas en todas las subclases de entidad de la misma manera, pero la cantidad de elementos que estas contengan al momento de la ejecución ira aumentando a medida que se baje en la jerarquía ya que cada subclase heredará los valores correspondientes a los bloques de las superclases y agregará a estos los valores correspondientes a los bloques que tenga definidos.

Dada una clase y una subclase de ésta, veamos un ejemplo de como serán los valores que tomen estas colecciones en las instancias de la clase y de la subclase.

Supongamos tener la clase A con la siguiente estructura de bloques:

```
BLOCK main
    ...
    BLOCK A1
        ...
    END
    BLOCK A2
        ...
    END
END
```

Y luego la clase B, la cual es subclase de A con la siguiente estructura de bloques:

```
BLOCK main
    ...
    BLOCK A1
        BLOCK B11
        END
        BLOCK B12
        END
    END
    BLOCK B2
        ...
    END
END
```

Esta subclase hereda la estructura de bloques de la anterior y agrega a esta lo siguiente, dentro del bloque A1, los bloques B11 y B12 y luego del bloque A1 el bloque B2.

Para la clase A, la colección tendrá tres elementos, y será inicializada como sigue

Bloque	Posición	Hijos	Padres	Siguientes
main	0	1	-1	-1
A1	1	-1	0	2
A2	2	-1	0	-1

Para la clase B se mantendrán las entradas correspondientes a la superclase, ya que hereda la estructura de bloques de esta, y agregará las suyas, modificando algunos valores para que las colecciones reflejen la nueva estructura, éstas, entonces, quedarán como sigue:

Bloque	Posición	Hijos	Padres	Siguientes
main	0	1	-1	-1
A1	1	3	0	5
A2	2	-1	0	-1
B11	3	-1	1	4
B12	4	-1	1	-1
B2	5	-1	0	2

La colección de flags va a tener, para las instancias de la clase A, tres elementos y, para las instancias de la clase B, seis elementos.

Al iniciar la hiperhistoria y después de inicializar las colecciones se activarán los flags correspondientes, siguiendo el esquema de activación de bloques visto en el capítulo anterior.

Mas adelante veremos como funcionan los métodos que activan y desactivan los bloques representados en estas colecciones.

4.8.2 - Métodos.

Ahora veremos cuales son los métodos implementados en la clase entity.

_getExternalContext: hObject;

Parámetros:

Este método no recibe parámetros.

Acción:

Este método no hace nada.

Retorna:

Retorna el contexto en el que esta incluido el objeto o NIL en caso de que este tomado por otra entidad.

_getContext: hObject;

Parámetros:

Este método no recibe parámetros.

Acción:

Este método no hace nada.

Retorna:

Retorna el contexto en el que esta incluido el objeto o NIL en caso de que este tomado por otra entidad.

Este método hace lo mismo que el anterior, cambiará su función en la subclase Context

_aceptaEvento(_event : _tEvent; sender : hObject;var _params : string):boolean;

Parámetros:

_event : _tEvent	Nombre del evento enviado a la entidad.
sender : hObject	Objeto que envía el evento.
_params : string	Parámetros del evento.

Acción:

Maneja los eventos predefinidos para las entidades en caso que ninguna subclase de esta lo haga, los eventos que maneja y las acciones que realiza para cada uno son las siguientes:

GETNAME(var n : string)

Devuelve en el parámetro el nombre del objeto receptor del evento.

GETCLASS(var n : string)

Devuelve en el parámetro el nombre de la clase del objeto receptor del evento.

GETCONTEXT(var c : hObject)

Devuelve en el parámetro el contexto en el que esta incluido el objeto receptor del evento.

GETOWNER(var e : hObject)

Devuelve en el parámetro la entidad que tiene tomado al objeto receptor del evento.

OBJFIRST

Posiciona un puntero interno en el primer elemento de la lista de objetos tomados por la entidad receptora del evento.

OBJNEXT

Mueve el puntero mencionado para que apunte al próximo elemento en la lista de objetos tomados por la entidad.

OBJCURR(var o : hObject)

Devuelve en el parámetro el objeto que esta siendo apuntado por el puntero interno.

OBJISLAST(var ultimo : boolean)

Devuelve en el parámetro el valor true en caso de que el objeto apuntado por el puntero interno sea el último de la lista.

TAKEBY(o : hObject)

Hace que el objeto pasado como parámetro del evento agregue al objeto receptor del evento a su lista de objetos internos, esto se interpreta como que el objeto pasado como parámetro toma al objeto receptor del evento.

El nuevo dueño del objeto receptor del evento será el objeto pasado como parámetro.

El nuevo contexto del objeto receptor del evento será NIL.

Al ejecutarse este evento, se envía al contexto en el que está incluído el objeto receptor el evento *OTAKED* y manda como el otro objeto involucrado (el objeto receptor de este evento lo verá en el atributo *sender*) a si mismo. Este mensaje se envía de manera asincrónica.

LEAVEBY(o : hObject)

Hace que el objeto pasado como parámetro elimine de su colección interna al objeto receptor del evento, esto se interpreta como que el objeto pasado como parámetro suelta al objeto receptor del evento.

El atributo *owner* que representa al objeto que tiene tomado al receptor se hace NIL.

El atributo *myContext* se asigna con el contexto en el que esta incluido el objeto pasado como parámetro, o sea, el que lo soltó.

Al ejecutarse este evento, se envía al contexto en el que están incluídos los objetos involucrados el evento *OLEAVED*, que este podrá acceder a través del atributo *sender*. Este evento se envía de manera asincrónica.

Retorna:

True, en caso de que el evento recibido sea alguno de los mencionados, en caso contrario, retorna el valor que devuelva la función *_aceptaEvento* implementada en la clase *hObject*.

_activarBloque(s : string);

Parámetros:

s : string nombre del bloque que se quiere activar

Acción:

Desactiva todos los bloques que pudieran estar activos en ese momento y activa el bloque cuyo nombre es pasado como parámetro y también hace la activación de los demás bloques, los que engloban al receptor y los englobados por este, según el esquema de activación de bloques mencionado en el capítulo anterior.

Esto último lo hace usando los dos métodos que explicamos luego de este.

Retorna:

Nada

_activaBajando(n : integer);

Parámetros:

n : integer Indice en la colección de bloques del bloque que se quiere activar.

Acción:

Realiza la activación del bloque cuyo Indice en las colecciones internas es pasado como parámetro y vuelve a llamar a este método, de manera recursiva, para activar el primer bloque interno al recibido como parámetro.

Retorna:

Nada

_activaSubiendo(n : integer);

Parámetros:

n : integer Indice en la colección de bloques del bloque que se quiere activar.

Acción:

Activa el bloque cuyo Indice corresponde al numero pasado como parámetro y si el bloque en cuestión esta englobado por otro, se hace la llamada, de manera recursiva al mismo método pero para activar este segundo bloque.

Retorna:

Nada

_bloqueSiguiente(activo : integer): string;

Parámetros:

s : string nombre del bloque que se quiere activar

Acción:

Nada

Retorna:

El nombre del bloque siguiente al bloque correspondiente al pasado como parámetro, esto según la definición de bloque siguiente dada en el capítulo anterior.

4.9 - Clase Context

Esta clase es subclase de Entity y representa ambientes que pueden ser habitados por entidades, estos ambientes, comunicados por links, formarán la estructura de navegación de la hiperhistoria.

4.9.1 - Atributos.

Los atributos que tiene esta clase son los siguientes (enumeraremos en este caso solo aquellos que son nuevos o que tienen una interpretación distinta que en la superclase) :

(hObject) id : string

Nombre del contexto dado por el programador al definir los contextos internos a otro contexto o el nombre de la clase del contexto mas externo de todos.

(hObject) icl : string

Contiene el nombre de la subclase del contexto definida por el usuario de la cual el contexto en cuestión es instancia o 'CONTEXT' si el contexto es instancia directa de esta clase.

(hObject) _myObjects : pCollection

Colección de los entidades de la hiperhistoria y links en caso de que el contexto no tenga contextos internos o de contextos en caso de que sea un contexto que englobe a otros.

(Entity) owner : hObject

Este atributo no tiene significado para los contextos, su valor en estos es indeterminado.

_entrys

Colección que se completará, en las subclases definidas por el programador, con referencias a los contextos internos a los cuales un objeto que penetre por esa entrada debe ir cuando ingresa al contexto.

Para los contextos en los cuales no se definen entradas esta colección permanece vacía.

4.9.2 - Métodos.

Luego, los métodos implementados para esta clase son los siguientes:

_init(oid, ocl : string; enContext : hObject)

Parámetros:

oid : string	Nombre del contexto dado por el programador
ocl : string	Nombre de la clase a la que pertenece el contexto
enContext : hObject	Referencia al contexto en el cual esta incluido.

Acción:

Inicializa la colección de entradas del contexto y luego ejecuta el procedimiento de inicialización, que tiene el mismo nombre que este, que hereda de la clase *hObject*.

Retorna:

El contexto en el que esta incluido el objeto o NIL en caso de que este tomado por otra entidad.

_internalObject(idObject : string):hObject

Parámetros:

idObject : string Especificación de un objeto de la hiperhistoria, este string tiene la siguiente forma: *nombreContexto.nombreContexto. ... .nombreObjeto*,

Acción:

Este método no hace nada.

Retorna:

El objeto de la hiperhistoria que se corresponde con la especificación pasada como parámetro o NIL en caso de que el objeto especificado no exista en el contexto buscado.

_getContext: hObject

Parámetros:

Este método no recibe parámetros.

Acción:

Este método no hace nada.

Retorna:

Retorna una referencia a sí mismo.

_makeLink(aLink : Link)

Parámetros:

aLink : link Link con el que se va a comunicar el contexto.

Acción:

Si el contexto no tiene entradas definidas, agrega el link a la colección interna del contexto y luego indica al link que haga lo propio con el contexto. En caso de tener entradas definidas, conecta al link con el contexto cuya referencia esta almacenada en el primer lugar de la colección de entradas.

Retorna:

El contexto en el que esta incluido el objeto o NIL en caso de que este tomado por otra entidad.

_aceptaEvento(_event : _tEvent; sender : hObject; var _params : string):boolean

Parámetros:

_event : _tEvent Nombre del evento enviado al contexto.

sender : hObject Objeto que envía el evento.

_params : string Parámetros del evento.

Acción:

Primero verifica si el evento recibido no es alguno de los predefinidos para las entidades, ejecutando el método *_aceptaEvento* de su superclase, si este retorna false, lo que quiere decir que no lo acepta, reenvía el evento recibido a todos los objetos que, en ese momento, estén habitando el contexto.

Retorna:

True.

4.10 - Clase Link

Esta clase es subclase de Entity y representa a aquellos objetos de la hiperhistoria que comunican dos o más contextos y que permiten que las entidades, a través de ellos, pasar de un contexto a otro.

4.10.1 - Atributos.

Los atributos de los links son los siguientes (mencionaremos solo los heredados que cambian de significado con respecto a sus superclases y los nuevos):

(hObject) id : string

Nombre del link que es el dado cuando se los declara en la sección *LINKS* de los contextos.

(hObject) icl : string

Contiene el nombre de la subclase definida por el usuario de la cual el objeto en cuestión es instancia o 'LINK' si el objeto es instancia directa de esta clase.

(hObject) myContext : hObject

Referencia al contexto en el que esta incluído la objeto, en el caso de los links es el contexto que lo engloba a el y a todos los contexos que comunica.

(hObject) _myObjects : pCollection

Colección de los contextos que el link comunica.

(Entity) owner : hObject

Como un link no puede ser tomado, el valor de este atributo es indefinido para las instancias de esta clase.

fstContext : Integer

Indice, en la colección interna de contextos que comunica el link, del primero de los contextos que comunica.

sndContext : integer

Similar al atributo anterior pero para el segundo de los contextos que comunica el link.

4.10.2 - Métodos.

Ahora veremos los métodos que esta clase tiene implementados:

_addContext(aContext : Context)

Parámetros:

aContext : Context Contexto a agregar a la lista de contextos que comunica el link

Acción:

Agrega el contexto pasado como parámetro a la lista de contextos que comunica el link.

Retorna:

Nada.

_init(oid, ocl : string; enContext : hObject)

Parámetros:

oid : string Nombre del link dado por el programador

ocl : string Nombre de la clase a la que pertenece el link

enContext : hObject Referencia al contexto en el cual esta incluido.

Acción:

Efectúa la iniclalización de su superclase y luego iniclaliza los atributos *fstContext* y *sndContext* en cero y uno respectivamente.

Retorna:

Nada.

_aceptaEvento(_event : _tEvent; sender : hObject; var _params : string):boolean

Parámetros:

_event : _tEvent	Nombre del evento enviado al link.
sender : hObject	Objeto que envía el evento.
_params : string	Parámetros del evento.

Acción:

Maneja los eventos predefinidos para las entidades en caso que ninguna subclase de esta lo haga, los eventos que maneja y las acciones que realiza para cada uno son las siguientes:

FCONTEXT(var c : hObject)

Devuelve en el parámetro el primero de los contextos que comunica el link.

SCONTEXT(var c : hObject)

Devuelve en el parámetro el segundo de los contextos que comunica el link.

CROSSBY(o : hObject)

Hace que la entidad pasada como parámetro pase de un contexto a otro a través del link receptor del evento.

Al hacer esto genera un evento, *ENTITYOUT*, para el contexto que acaba de abandonar la entidad que cruza el link, como otro objeto involucrado se envía al objeto que cruza el link y como primer parámetro al link por el cual cruza el objeto.

De la misma manera se genera el evento *ENTITYIN*, para el contexto en el cual el objeto entra, con los mismos parámetros que el anterior.

Por último se genera un evento para el mismo link por el cual cruzó el objeto, este evento es *ENTCROSS* y como otro objeto involucrado se envía al objeto que cruza y como primer parámetro al contexto del cual proviene dicho objeto.

Retorna:

True en caso de que reciba uno de los eventos mencionados o que el evento que reciba lo maneje alguna de sus superclases, en caso contrario, retorna False.

De esta manera se describieron todos los objetos básicos de la hiperhistoria, luego, el programador, al implementar la hiperhistoria, debe definir clases nuevas, las cuales deben ser subclases de estas tres últimas.

4.11 - Clases definidas por el programador.

Ahora veremos como sería la estructura de las clases generadas a partir de la definición de objetos por parte del programador de la hiperhistoria, explicaremos como seria una clase genérica pero las clases que resulten de la traducción pueden tener ligeras diferencias según se trate de una entidad, un contexto o un link.

4.11.1 - Atributos.

Los atributos que tendrá, además de los que hereda de las superclases, dependen de los atributos declarados en la definición de la clase o las conexiones definidas en esta.

Por cada atributo común declarado en la clase, se genera un atributo en la traducción de la clase, del tipo correspondiente al definido, ejemplo, si a una clase se le define un atributo como sigue:

```
ATTRIBUTES
    estado : integer;
```

En la clase traducida se coloca el siguiente atributo:

estado : integer;

Si en cambio el atributo es interno, genera en la traducción dos atributos, como sigue:

estado : Integer; _x_estado : Integer;

La declaración *_x_estado* es una variable auxiliar utilizada para determinar el cambio de valor del atributo con el objeto de notificar a la interface.

Otra declaración que genera atributos en la traducción de la clase es la declaración de conexiones para un objeto, veamos el siguiente ejemplo:

CONNECTIONS
output;

Aquí se declara una conexión para que luego, cuando la hiperhistoria este funcionando, el objeto pueda, a través de la conexión, enviar eventos a otro objeto.

La traducción de esta cláusula genera el siguiente atributo en el código Pascal de la clase.

output : hObject;

Este atributo tomara valor cuando en la sección de inicialización de algún contexto que contenga a los dos objetos a conectar se ejecute la cláusula:

<objeto1> CONNECT output WITH <objeto2>

Mas adelante veremos como se traduce esta especificación.

En caso de los contextos se tienen declaraciones especiales dentro de la plantilla de definición, estas son las declaraciones de entradas al contexto y las declaraciones de links y contextos Internos.

Comencemos por la declaración de entradas que se efectúa para especificar que cuando una entidad penetre al contexto en realidad lo hará a un contexto interno a este. La declaración, como vimos en el capítulo anterior se escribe como sigue :

ENTRYS nomEntrada1 [, nomEntrada2]*;

Esta declaración no tiene una traducción en lenguaje Pascal, simplemente es necesario definir los nombres de las entradas para que luego, cuando se genere la función de inicialización para el contexto se sepa como traducir la declaración de comunicación entre contextos, la cual veremos mas adelante.

Luego se definen los contextos y links internos que tendrá el contexto, estas declaraciones se traducen en código en la función de inicialización de la clase traducida, las declaraciones en esta función la veremos en detalle más adelante.

4.11.2 - Métodos.

Las clases definidas tendrán métodos, además de los que heredan, para implementar su comportamiento, estos métodos son los siguientes:

_init(oid, ocl : string; enContext : hObject)

Parámetros:

oid : string Nombre del objeto dado por el programador

ocl : string Nombre de la clase a la que pertenece el objeto

enContext : hObject Referencia al contexto en el cual está incluido.

Acción:

Efectúa la inicialización que hereda de su superclase y luego inicializa los atributos de la clase, si los tiene, con los valores especificados en la sección *INITIALIZATION* de la declaración de la clase en el modelo.

Aquí se inicializan las estructuras que almacenan la información acerca de los bloques de comportamiento del objeto y su activación.

También activa el bloque mas externo del comportamiento del objeto.

Si el método pertenece a un contexto, en este también se incluyen la inicialización de los contextos y links internos que pudiera tener, la inclusión de objetos en si mismo y en contextos internos, la inicialización de las comunicaciones entre contextos a través de links y el conexionado entre objetos internos al contexto.

Retorna:

Nada.

_aceptaEvento(_event:_tEvent;sender:hObject;var _params:string):boolean

Parámetros:

_event : _tEvent Nombre del evento recibido por objeto.

sender : hObject Objeto que envía el evento.

_params : string Parámetros del evento.

Acción:

Ejecuta la regla correspondiente al evento recibido, si esta se encuentra en un bloque activo y si la precondición de esta es verdadera.

Si ninguna de las reglas cumple esta condición, pasa ejecuta la misma función pero en la superclase y retorna el valor que este método retorne, esto se hace así hasta que se ejecute alguna regla o hasta que se ejecute el método *_aceptaEvento* correspondiente a la clase *hObject* , el cual retorna siempre falso, lo que hará que esta función retorne falso, lo que significa que el evento fue rechazado.

Retorna:

True en caso de que acepte el evento, False en caso contrario.

_connect(cConx : string; anObject : hObject)

Parámetros:

cConx : string Nombre de una conexión.

anObject : hObject Objeto que se debe conectar a la conexión especificada.

Acción:

Verifica que la conexión especificada sea una de las definidas para el objeto y si es así, asigna a esta el objeto especificado en el parámetro *anObject*.

Si la conexión especificada no es ninguna de las definidas en el objeto, ejecuta el mismo método pero de su superclase, esto sigue así hasta que se encuentre alguna conexión que coincida con la especificada o hasta que se llegue a una superclase que no tenga conexiones, en este caso, el método no hace nada.

Retorna:

Nada.

Luego viene un método por cada regla que tenga definida el objeto en su comportamiento, estos métodos tienen la siguiente forma:

_rg7(_event:_tEvent;sender:hObject;var _params:string):boolean

Parámetros:

_event : _tEvent Nombre del evento recibido por objeto.

sender : hObject Objeto que envía el evento.

_params : string Parámetros del evento.

Acción:

Verifica si el evento de la regla se corresponde con el especificado en el parámetro *_event*, si es así, evalúa la precondición de la regla y si su valor es verdadero, ejecuta el comportamiento que esta regla tiene definido y, si se especifica alguna postcondición, modifica la estructura que almacena la información acerca de los bloques activos para reflejar el cambio.

Retorna:

True en caso de ejecutar el comportamiento definido en la regla, False en caso contrario.

El nombre de los métodos que se implementan para estas clases comienzan con la cadena 'rg' y siguen con un número de regla único en toda la Implementación, el nombre 'rg7' de la descripción es sólo un ejemplo.

4.12 - Conjuntos de objetos.

Tenemos otros componentes de la hiperhistoria que se definen a efectos de la inicialización, son los *SET*, o sea, los conjuntos de objetos que se deben especificar para incluirlos luego en contextos.

Los sets no se traducen a clases en lenguaje Pascal sino a procedimientos, los cuales reciben como parámetro al contexto al cual se le agregaran los objetos, y su acción es crear los objetos que tiene definidos y agregarlos a la colección de objetos internos del contexto especificado.

Veamos un ejemplo de la declaración de un *SET* y su traducción a código Pascal, supongamos tener la siguiente declaración en el lenguaje del modelo:

```
SET lucesYLlaves
    luz1, luz2 : LUZ;
    llave1 : LLAVE;
END;
```

Esto se traduciría a código Pascal como sigue:

```
procedure lucesYLlaves( enContext : hObject );
    begin
        enContext^.aCollection^.insert( new( LUZ, _Init( 'LUZ1','LUZ', enContext ) ) );
        enContext^.aCollection^.insert( new( LUZ, _Init( 'LUZ2','LUZ', enContext ) ) );
        enContext^.aCollection^.insert( new( LLAVE, _init( 'LLAVE1','LLAVE', enContext ) ) );
    end;
```

Este procedimiento será luego invocado cuando en la inicialización de un contexto se declare la cláusula *INCLUDES*.

Con esto tenemos descripta la arquitectura de la hiperhistoria y cual es la estructura y características de los objetos componentes de dicha arquitectura, ahora veremos como estos objetos interactúan al iniciar la hiperhistoria para inicializarse y durante la ejecución de esta, en lo que respecta a despacho de eventos y comunicación con la interface.

4.13 - Implementación de los mecanismos principales de las hiperhistorias.

En esta sección se explicará la implementación de los diferentes mecanismos que hacen que la hiperhistoria funcione, esto es, dada la sintaxis en el lenguaje del modelo se explicará como se traduce eso a lenguaje Pascal y como funciona el código traducido, también se describirán mecanismo de inicialización y funcionamiento internos de la hiperhistorias como así también el modo en que debe funcionar una Interface para conectarse a la hiperhistoria.

4.13.1 - Implementación de la inicialización de los objetos de la hiperhistoria.

Los mecanismos de inicialización que ahora veremos se implementarán en la función *_init* de cada uno de los objetos que sean definidos por el programador y serán una traducción de las sentencias especificadas en la cláusula *INITIALIZATION* en cada clase, según corresponda para esos objetos la especificación de la inicialización descripta.

4.13.2 - Inicialización de atributos.

Cuando se inicializa un atributo de una clase se agrega a la función de inicialización de la clase una línea con la asignación correspondiente, en caso de los atributos internos también se inicializa la copia del atributo con el mismo valor que el atributo principal.

4.13.3 - Inclusión de objetos.

Esta inicialización se hace sólo en los contextos, veamos una inclusión genérica y como se traduciría en código Pascal:

<nombre del contexto> INCLUDES <nombre del conjunto>;

Este tipo de inicialización se traducirá como sigue:

```
contextoAux := self.internalObject( '<nombre del contexto>' );
<nombre del conjunto>( contextoAux );
```

El nombre del contexto puede no existir, en cuyo caso se pasará como parámetro de la función *internalObject* el string nulo con lo cual esta devolverá el mismo contexto y no uno interno a este. En caso de especificarse un contexto interno o un nombre compuesto de contextos anidados separados por puntos (ver *Inclusión de objetos* en el capitulo anterior), la función retornara una referencia al contexto especificado.

Luego, el contexto obtenido es pasado como parámetro hacia la función que representa al set especificado y esta agregará a la lista de objetos internos al contexto los objetos que esta función cree.

En caso que se especifique mas de un conjunto separados por el signo '+' como nombre de conjunto, simplemente se repetirá la segunda sentencia tantas veces como nombres distintos haya, cada una de las repeticiones se hará invocando a una función cuyo nombre corresponderá a cada nombre de conjunto. Veamos el siguiente ejemplo que muestra lo anterior, si tenemos una especificación de inclusión como sigue:

casaGrande.living INCUDES conjuntoLlaves+conjuntoLamparas;

La traducción resultante será la siguiente:

```
contextoAux := self.internalObject( 'casaGrande.living' );
conjuntoLlaves( contextoAux );
conjuntoLamparas( contextoAux );
```

En la primera de las tres sentencias se obtiene el contexto especificado y en las otras dos se le agregan los objetos especificados en los conjuntos correspondientes.

4.13.4 - Comunicación de contextos a través de links.

Esta inicialización, que se hace sólo en los contextos, sirve para indicar que contextos están comunicados a través de que links.

Como se vio en el capitulo anterior, una inicialización de una comunicación genérica seria:

<nombreContexto> LINKED WITH <nombreLink/Entrada>

La traducción de esta sentencia en código Pascal seria la siguiente:

<nombreContexto>^._makelink(<nombreLink>);

En esta traducción lo que se hace es ligar al link especificado en *nombreLink* a uno de los contextos que este comunicará. En caso de ser el nombre de una entrada a la que se hace referencia, la traducción seria la siguiente:

_entrys^.insert(<nombreContexto>);

Esta sentencia agrega a la colección de entradas al contexto el contexto interno al cual se debe ligar un link externo cuando, mas adelante, se lo quiera conectar con el contexto externo.

En la colección *_entrys* puede haber indicado mas de un contexto, el orden en que estos se ligarán será según la ubicación de estos en la colección, es decir, el contexto que este en la posición uno, será el primero que se conecte cuando desde afuera se quiera ligar el contexto principal con un link, y así sucesivamente.

El siguiente es un ejemplo de la traducción de esta sentencia:

living LINKED WITH puertaEntrada, puertaCocina, puertaPieza;

Suponemos que esta declaración se realiza en la sección de inicialización de un contexto *CASA* el cual tiene declarado a *living* como contexto interno, a *puertaCocina* y *puertaPieza* como links internos y a *puertaEntrada* como una entrada. La traducción a código Pascal de esta declaración será la siguiente:

```
_entrys^.insert( living );
living^._makeLink( puertaCocina );
living^._makeLink( puertaPieza );
```

4.13.5 - Conexionado de objetos.

Esta declaración se hace en la definición de los contextos y se usa para establecer comunicaciones explícitas, entre objetos de la hiperhistoria incluidos en dicho contexto o el contexto mismo, de manera directa o a través de *canales*, los cuales reenviarán los eventos que reciban a todos los objetos conectados al canal excepto el que generó el evento.

La forma de especificar una conexión en forma directa entre dos objetos, según lo visto en el capítulo anterior seria la siguiente :

<objeto1> CONNECT [<nomConexion>] WITH <objeto2>;

Lo cual significaría que el *objeto1* puede enviar mensajes hacia el *objeto2* siempre y cuando el primer objeto tenga definida una conexión y se la especifique en esta declaración, en caso contrario la declaración no tendría sentido ya que ninguno de los objetos, al no tener conexiones definidas, le puede enviar eventos al otro, esto en caso de que el segundo objeto no sea un canal, ya que si este fuera el caso, el primer objeto podría recibir eventos a través de este.

La traducción de esta declaración seria la siguiente :

```
aux := self._internalObject( 'objeto1' ) ;
aux^._connect( 'nomConexion', self._InternalObject( 'objeto2' ) ) ;
```

Esto lo que hace es, en primer lugar, obtener el primer objeto incluido en el contexto en el que se haga la declaración y luego

conectarlo con el segundo a través de la conexión, en caso de que esta se especifique.

Luego de esta declaración el objeto1 puede recibir eventos del objeto2 en caso de que este sea un canal o puede enviar eventos al objeto2 en caso de que se le haya definido una conexión al objeto1 y esta sea especificada en la declaración precedente.

4.13.6 - Envío de un evento.

Veremos ahora como es la implementación de la sentencia de envío de un evento de un objeto a otro y como se utilizan algunos de los métodos descriptos anteriormente para los objetos componentes de la hiperhistoria.

4.13.6.1 - Envío sincrónico.

Comencemos por recordar la sintaxis de la declaración, supongamos tener visible un objeto que llamamos *llave* y queremos enviar a este el evento *activar* desde otra entidad componente de la hiperhistoria, primero veremos el envío de este evento de manera sincrónica, el modo de enviar este evento sería el siguiente :

llave<--activar ;

Esta declaración generará el siguiente código en lenguaje Pascal :

```
_paramStr := '' ;
ok := llave^._aceptaEvento( 'ACTIVAR', addr(self), _paramStr );
```

A la variable *ok* se le asigna el resultado del envío para que pueda ser consultado por la sentencia siguiente y tomar alguna decisión según su valor, en este caso dicho valor tendrá sentido, el de determinar si el evento fue o no aceptado, en caso de eventos asincrónicos, a través de conectores o hacia la interface, este valor será siempre *true*.

Luego se ejecuta del objeto *llave* la función *aceptaEvento* que dicho objeto la tendrá implementada si tiene algún comportamiento definido, en caso contrario se ejecutará la siguiente función subiendo en la jerarquía de clases de los objetos de la hiperhistoria.

El parámetro *addr(self)* es utilizado para enviar al receptor una referencia del objeto que genera el evento y, por último, el parámetro *_paramStr* contendrá la lista de parámetros enviados con el evento, en el caso del ejemplo será un string nulo ya que no se envían parámetros, en caso de ser así veremos mas adelante un ejemplo de implementación.

El objeto destinatario del evento, al ejecutar la función correspondiente, ya sea una propia o heredada, llevará a cabo una serie de acciones y luego de terminar, el control retorna al objeto llamante, o sea, el que generó el evento.

El objeto destinatario del evento puede ser otro, el caso del ejemplo, o puede ser el mismo que genera el evento, usando la palabra reservada *self* se especifica este segundo caso, la traducción para un envío hecho de esta manera sería la siguiente :

```
_paramStr := '' ;
ok := self._aceptaEvento( 'ACTIVAR', addr(self), _paramStr );
```

La ejecución de la función *_aceptaEvento* en este caso se realiza de la misma forma que la anterior y cuando termina el control vuelve a la regla que hizo la llamada.

El objeto que recibe la llamada puede ser cualquier referencia válida a un objeto de la hiperhistoria, incluso a un conector.

4.13.6.2 - Envío asincrónico.

En caso de los eventos asincrónicos, la implementación es diferente, en este caso el receptor del evento es siempre la cola de eventos la cual lo encola hasta que le llegue el turno de ser despachado.

Supongamos el evento anteriormente descripto pero enviado de manera asincrónica, este se escribiría de la siguiente manera :

llave<==activar

En este caso, la traducción a código Pascal sería la siguiente:

```
_paramStr := '' ;
ok := queue^.pushEvent( llave, 'ACTIVAR', addr(self), _paramStr ) ;
```

En este caso el valor de la variable *ok* será siempre true, ya que la cola de eventos siempre acepta el método *pushEvent* y el control vuelve al que lo envía antes de que el evento sea realmente ejecutado por el destinatario.

En esta declaración se agrega al principio otro parámetro en el cual se envía a la cola de eventos una referencia al objeto destinatario del evento para que luego esta pueda enviarlo efectivamente, los restantes parámetros tienen el mismo sentido que en el ejemplo anterior.

Como identificación del destinatario puede ir una referencia a otro objeto, como se ve en el ejemplo, *self*, en caso de que el evento sea enviado de manera asincrónica hacia el mismo objeto o una referencia a un canal.

4.13.6.3 - Eventos con parámetros.

La variable *_paramStr* en ambos ejemplos se asigna en caso de que el evento enviado lleve parámetros, sea asincrónico o no y no importando la naturaleza del destinatario. Veamos un ejemplo de la asignación de esta variable, primero escribimos la sentencia en lenguaje del modelo :

llave<==activar(2, estado)

En este caso se envía a la entidad *llave* el evento *activar* con dos parámetros, uno numérico, el cual se interpreta como entero por defecto, y otro, una cadena de caracteres, llamado *estado*. La forma en que esto de traduce es la siguiente :

```
_paramStr := '';
_paramStr := _paramStr + _strINTEGER( 2 );
_paramStr := _paramStr + _strSTRING( ESTADO ) ;
ok := queue^.pushEvent( llave, 'ACTIVAR', addr(self), _paramStr ) ;
_getINTEGER( _paramStr ) ;
ESTADO := _getSTRING( _paramStr ) ;
```

El string de parámetros se va formando con funciones las cuales toman el dato que se quiere enviar y lo devuelve como una cadena de caracteres convenientemente formateada para luego ser interpretada por el receptor del evento.

El formato con que se almacena cada parámetro en el string mencionado es el siguiente :

- Un byte que indica el tipo del parámetro que se almacena, este byte puede ser :
- I - Entero. (dos bytes de longitud)
- W - Word. (dos bytes de longitud)
- B - Byte. (un byte de longitud)
- L - Long (cuatro bytes de longitud).
- T - Booleano (dos bytes de longitud)
- S - String.(longitud variable, termina en cero)
- O - Objeto de la hiperhistoria (en realidad en un Word, o sea de dos bytes de longitud, y representa la identificación interna del objeto).

- Luego viene el valor correspondiente al tipo del parámetro pasado, la cantidad de bytes que ocupe este valor depende, por supuesto, del tipo del parámetro.

Por último se saca los valores de la cadena de parámetros enviada a la función en caso de que esta la haya modificado y se asignan los valores de esta a aquellos parámetros que eran variables.

4.13.7 - Recepción de un evento.

En esta parte se explicará como es el mecanismo de recepción de un evento por parte de un objeto de la hiperhistoria, es decir, que métodos del objeto se ejecutan cuando esto sucede y en que orden y cuales son los atributos involucrados en la acción.

La implementación de la recepción de un evento no varia ya sea que este provenga desde un objeto mediante un envío sincrónico, o a través de un envío asincrónico o desde la interface, cualquiera sea la forma, la implementación de la recepción es la misma.

La recepción de un evento por parte de un objeto hace que se ejecute la función *_aceptaEvento* del objeto receptor de éste, a esta función se le envía como parámetros el nombre del evento, una referencia al objeto que lo envía, en caso de ser un evento de interface esta de NIL, y un string con los parámetros que recibe el evento.

La implementación de una función típica de recepción de eventos es la siguiente :

```
function _PUERTA._aceptaEvento(_event:_tEvent;sender:hObject;var _params:string):boolean;
    var sigue : boolean;
    begin
        sigue := true;
        if _flags^.at( 0 ) then begin
            if sigue then begin
                sigue := false;
                if _rg2( _event, sender, _params ) then
                else if _rg3( _event, sender, _params ) then
                else if _rg4( _event, sender, _params ) then
                else sigue := true
            end
        end;
        if sigue then sigue := _LINK._aceptaEvento(_event,sender,_params );
        _aceptaEvento := not sigue
    end;
```

La función que se ve aquí corresponde a la implementacion de una puerta, esta define tres eventos a los cuales responde y que se reflejan en el código anteriormente visto

Cada uno de las funciones cuyo nombre empieza con *_rg* corresponde a la traducción de una de las reglas implementadas en el objeto en el cual aparece, veamos ahora el fuente en lenguaje del modelo de una de las reglas correspondientes al objeto cuya función de proceso del evento se reprodujo arriba :

```
( abrir( clave : string ), (not abierta) and (clave='DFGS'),  LOCAL lastContext : string;
    abierta := true;
    sender<--getContext( lastContext );
    INTERFACE<--meAbren( lastContext );, NULL )
```

Esta regla implementa, en lenguaje del modelo, la respuesta del objeto *Puerta* al evento *abrir*, esta regla esta formada por el encabezado, el cual no especifica parámetros, cuando llegue un evento que coincida con este encabezado, se continuará con el procesamiento de esa regla, en caso contrario se buscará otra.

Al encabezado le sigue la precondición, en este caso es *(not abierta) and (clave='DFGS')*, la variable *abierta* es un atributo

interno del objeto implementado y *clave* es un parámetro recibido por el evento, si esta precondición es verdadera en el momento que se recibe el evento, se ejecuta la regla.

El comportamiento de esta regla, o sea, la lista de acciones que ejecuta hacen lo siguiente, primero, se declara una variable local a la regla de tipo *string*, luego se pone en *true* el atributo *abierta*, para indicar que la puerta se abre.

Las dos sentencias siguientes se usan para notificar a la interface el cambio de valor del atributo, esto se hace así en lugar de hacerlo con atributos normales del objeto por una dificultad en la implementación de la comunicación con la interface, la cual resultaba mas sencillo solucionar de esta manera.

En la primer sentencia mencionada obtenemos el contexto, en la variable *lastContext*, en el cual esta incluido el objeto que envió el mensaje abrir y en segundo lugar se envía al objeto que representa a la puerta en la interface el evento *meAbren* para informar a esta del cambio de valor del atributo para que lo refleje de manera adecuada.

Ahora veremos como es la traducción que se genera a partir del código fuente anterior :

```
function _PUERTA._rg2(_event:_tEvent; sender:hObject; var _params:string):boolean;
    var
        CLAVE : STRING ;
        LASTCONTEXT : STRING;
        _paramStr, _xxStr : string;
        ok : boolean;
    begin
        LASTCONTEXT := '';
        _rg2 := false;
        if _event = 'ABRIR' then
            begin
                _xxStr := _params;
                CLAVE := getSTRING( params ) ;
                if (NOT ABIERTA) AND (CLAVE='DFGS') then
                    begin
                        ABIERTA := TRUE ;
                        _paramStr := '';
                        _paramStr := _paramStr + _strSTRING( LASTCONTEXT );
                        if SENDER <> NIL then
                            ok := SENDER^._aceptaEvento( 'GETCONTEXT', addr(self), _paramStr );
                        LASTCONTEXT := _getSTRING( _paramStr );
                        _paramStr := '';
                        _paramStr := _paramStr + _strSTRING( LASTCONTEXT );
                        ok := Kernel^.sendToInter( _internalId, 'MEABREN', _paramStr, 1 );
                        _rg2 := true;
                        _params := '';
                    end
                else
                    _params := _xxStr
            end
    end;
```

Este método se ejecutará cada vez que le corresponda según este activo el bloque que contiene a la regla correspondiente, por cada vez que se ejecute devolverá *true* si se llevo a cabo la traducción de la lista de acciones internas de la regla.

Aquí se ve un ejemplo de como se traduce la regla, como se determina si el evento recibido se corresponde con el de la regla, como se reciben los parámetros y como se implementa la precondición y como se traduce la lista de acciones internas.

En esta traducción se ve un envío de evento hacia la interface, este lo explicaremos en detalle mas adelante.

4.13.8 - Conexión de la aplicación con la interface.

Las dos partes de la hiperhistoria son dos programas que están ejecutándose, para que ambos se mantengan sincronizados, es decir, que la interface refleje fielmente lo que este pasando en la hiperhistoria y que la aplicación reciba los eventos que origina el usuario del sistema y se responda en consecuencia, es necesario implementar un mecanismo por el cual ambos módulos se comuniquen.

La implementación de esta comunicación, siempre hablando en el ambiente Windows, consiste en tener una DLL que será usada por ambos módulos y que servirá para que estos intercambien las identificaciones de las tareas que tendrá cada una y para inicializar ciertos aspectos de la comunicación, como ser el mensaje con el cual se comunicarán.

4.13.8.1 - DLL Común.

Esta DLL sera usada por ambas partes de la hiperhistoria para comunicarse, sera como un administrador que tendrá los datos de la aplicación y de las interfaces que en cada momento se encuentren activas y servirá como nexo para que las interfaces conozcan a la aplicación y viceversa.

Las variables internas de la mencionada DLL son las siguientes :

HHM : word;

Es el handler del mensaje de Windows que usaran tanto la aplicación como las interfaces para comunicarse.

tpHandle : tHandle;

Hander de la tarea Windows correspondiente a la aplicación.

tbkHandles : array[1..20] of tHandle;

Handlers de las tareas Windows correspondientes a las interfaces que haya activas en cada momento (una o más).

tbkCount : byte;

Contador el cual indica la cantidad de interfaces activas en cada momento.

regPas : boolean;

Variable que indica si la aplicación esta activa.

Luego en esta DLL se implementan una serie de procedimientos y funciones que detallaremos a continuación :

function resetComm: word; export;

Inicializa la DLL, esta función no debe ser ejecutada por las interfaces.

function initLib: word; export;

Registra el mensaje común a la interface y la aplicación que se usará en Windows, si no estaba registrado, y retorna el handler correspondiente, esta función la ejecutarán tanto la aplicación como las interfaces que se quieran comunicar con esta.

function getCount: byte; export;

Retorna la cantidad de interfaces activas que haya en el momento de la invocación, esta función la usa sólo el Kernel.

function registrarTool(hWin : THandle): bool; export;

Esta función debe ejecutarla cada interface, cuando inicie su ejecución, para registrarse y determinar si la aplicación con la que se quiere comunicar esta activa.

function registrarPas(hWin : THandle): bool; export;

Esta función sera ejecutada por el Kernel cuando inicie su ejecucion para determinar si existe alguna interface activa en ese momento.

function toolHandle(numInt:byte): tHandle; export;

Esta función la debe ejecutar el Kernel, una vez que se inicio la ejecución de la hiperhistoria, cada vez que quiera obtener el handler de la interface identificada con el numero pasado como parámetro.

function pasHandle: tHandle; export;

Esta función la debe ejecutar cada interface, una vez que este conectada con la aplicación, para obtener el handler de esta y poder, de esta manera, enviarle eventos.

function endInterface(hWin : tHandle): word; export;

Esta función la deben ejecutar aquellas interfaces que dejen de estar activas, al hacerlo, el Kernel no intentara enviarle ningún evento más a estas.

4.13.8.2 - Uso de DDE.

La forma en que la aplicación y las interfaces se comunican es mediante el uso del Intercambio Dinámico de Datos (DDE) de Windows, esto es, el envío de un mensaje de Windows entre la aplicación y las interface, este mensaje es el que registran ambos con la DLL común que se vio arriba, entonces, cada vez que se recibe este mensaje, la aplicación o alguna interface, el que reciba el evento, deben sacar datos de un área de memoria que reservó la aplicación que envió el evento y en la cual se almacenó la información de este.

4.13.9 - Envío de eventos hacia la interface.

Al estar funcionando la hiperhistoria, la aplicación necesita enviarle eventos a las interfaces que estén conectadas a ella para que estas puedan representar lo que esta pasando.

En esta parte se explicará como se implementa la comunicación entre la aplicación y la interface, como se ponen de acuerdo para identificar los objetos, como se traduce la sentencia de envío de eventos a la interface en forma explícita y como se hace cuando el envío es implícito por la modificación de atributos comunes de cada objeto, cuales son y como se pasan los datos que son enviados a cada interface y como hace el kernel para efectuar la comunicación.

4.13.9.1 - Archivo de nombres.

Para enviar eventos hacia la interface debemos conocer de esta como se identifican los objetos que, en ella, se corresponden con los de la hiperhistoria para, de esta manera, hacer que estos eventos los reciban los objetos de interface que correspondan.

A cada objeto de la hiperistoria definido en la aplicación se le asigna un nombre, el cual esta compuesto por la secuencia de contextos en el cual el objeto esta incluido cuando arranca la hiperhistoria además del nombre del objeto propiamente dicho, por ejemplo, veamos el ejemplo de una silla incluida en el living de una casa, el nombre completo que esta tendría seria:

CASA.LIVING.SILLA

Esto siempre suponiendo que al momento de iniciar la hiperhistoria el contexto mas externo sea *CASA*.

El Kernel de la hiperhistoria le asigna a cada objeto una identificación interna consistente en un número entero de uno en adelante a medida que los objetos se van creando cuando arranca la hiperhistoria, el orden de creación de los objetos es siempre el mismo, por lo tanto en ejecuciones sucesivas de la misma hiperhistoria cada objeto tendrá la misma identificación interna.

Para que en la interface se conozcan las identificaciones internas de cada objeto, el kernel genera un archivo en el que contiene los nombres de todos los objetos de la hiperhistoria que se crean, estos nombres forman una lista en la cual el primero que aparece corresponde al objeto con identificación interna cero , el segundo al uno y así sucesivamente.

Este archivo se debe completar con los nombres de los objetos de interface correspondientes a cada objeto y luego, cada interface deberá leer una copia de este archivo correspondiente a su esquema de nombres y de esta manera determinar la identificación interna de cada objeto.

La forma en que el archivo de nombres para cada interface se complete dependerá de la interface y su manera de nombrar y acceder a los objetos que se crean en ella que representen a objetos de la aplicación, el siguiente es un ejemplo de como se completa el archivo para una implementación particular de una hiperhistoria :

(0)MUNDO
(1)MUNDO.UNACASA
(2)MUNDO.UNACASA.LIVING/LIVING
(3)MUNDO.UNACASA.COCINA/COCINA
(4)MUNDO.UNACASA.PIEZA/PIEZA

(5)MUNDO.UNACASA.LIVCOC/LIVCOC
(6)MUNDO.UNACASA.LIVPIE/LIVPIE
(7)MUNDO.UNACASA.LIVING.PERSONAJE/FOTO
(8)MUNDO.UNACASA.LIVING.LAPE/BALON
(9)MUNDO.UNACASA.LIVING.ELHOGAR/FUEGO
(10)MUNDO.UNACASA.LIVING.LALLAVE/LLAVE
(11)MUNDO.UNACASA.LIVING.ELCABLE
(12)MUNDO.UNACASA.LIVING.LALAMPARA/LAMPARA
(13)MUNDO.UNACASA.COCINA.CLOCK/RELOJ
(14)MUNDO.UNACASA.COCINA.LALLAVE/CLLAVE
(15)MUNDO.UNACASA.COCINA.ELCABLE
(16)MUNDO.UNACASA.COCINA.LALAMPARA/CLAMPARA
(17)MUNDO.UNACASA.PIEZA.VAS/VASO
(18)MUNDO.UNACASA.PIEZA.LALLAVE/PLLAVE
(19)MUNDO.UNACASA.PIEZA.ELCABLE
(20)MUNDO.UNACASA.PIEZA.LALAMPARA/PLAMPARA
(21)MUNDO.BARRIO/BARRIO
(22)MUNDO.PLAZA/PLAZA
(23)MUNDO.BARCOC/BARCOC
(24)MUNDO.BARPLA/BARPLA
(25)MUNDO.PLAZA.LALLAVEPUER/LLAVEPUERTA

Así es como queda el archivo de nombres generado por el kernel de la hiperhistoria y luego completado para una interface particular, aquí se ve que los nombres que se indican para la interface, (los que están después de la barra) no tienen por que corresponder con los que se definen en la hiperhistoria, también, para esta Implementación, los nombres que no están completados con su correspondiente de interface, representan objetos que existen en la hiperhistoria pero no tienen representación en la interface.

En este ejemplo, la interface al arrancar leerá este, luego, cada vez que reciba un mensaje del kernel de la hiperhistoria, obtendrá la identificación del objeto al cual se refiere el mensaje, que viene con el evento, determinará el nombre de interface correspondiente a este basándose en el archivo de nombres y ejecutará la acción de interface correspondiente al evento recibido y al objeto que lo recibe que corresponda.

En este archivo también se debe basar la interface para enviar eventos a objetos de la hiperhistoria, mas adelante veremos la explicación de este mecanismo.

4.13.9.2 - Traducción de la sentencia.

Aquí veremos como se traduce la sentencia de envío explícito de eventos por parte de los objetos de la hiperhistoria hacia la interface, como vimos, una sentencia de envio en el lenguaje del modelo puede ser la siguiente :

```
INTERFACE<--meAbren( lastContext );
```

Esta sentencia significa que el objeto que la implementa en su comportamiento envía a la representación de este objeto en la interface el evento *meAbren* con la variable *lastContext* como parámetro.

La traducción de esta sentencia, que ya vimos mas arriba pero la reproducimos aquí, es la siguiente :

```
_paramStr := '';
_paramStr := _paramStr + _strSTRING( LASTCONTEXT );
ok := Kernel^.sendToInter( _internalId, 'MEABREN', _paramStr, 1 );
```

En esta traducción lo primero que se hace es armar el string de parámetros, este paso es exactamente igual para todas las

formas de envío de eventos.

Luego, en la última sentencia, se le indica al Kernel que envíe a la interface el evento, el método *sendToInter* es el que hace el envío y los parámetros que se le pasan son : la identificación interna del objeto, el nombre del evento, el string con los parámetros y un número que indica la cantidad de parámetros que se pasan en el.

Este método del kernel lo que hace es pedirle a windows que reserve un área de memoria, copia los datos que se van a pasar a la interface, éstos datos son la identificación del objeto destino, el nombre del evento y los parámetros, esto por cada una de las interfaces activas, para esto usa el contador de interfaces activas de la DLL común que vimos mas arriba.

Luego envía un mensaje windows a cada una de las interfaces activas y le manda como parámetro de ese mensaje el handler del área de memoria donde copio los datos del evento.

4.13.9.3 - Envío implícito de eventos.

Este envío se produce cuando se modifica alguno de los atributos comunes del objeto en la lista de acciones internas de una regla, el modo en que se manda el evento a la interface es similar al del envío explícito solo que ahora el nombre del evento que se envía es el mismo que el nombre del atributo interno que se modifica y como primer parámetro se manda el valor de dicho atributo.

4.13.9.4 - Formato de los datos enviados a la interface.

En las áreas de memoria que reserva el kernel para pasar los datos hacia la interface se copian los datos de este en un formato predeterminado y fijo, este formato es el siguiente ;

- Dos bytes para la identificación del objeto destino del evento, en el primer byte se encuentra la parte baja del numero y en el segundo la parte alta.
- Luego una cadena de caracteres terminada en cero que es el nombre del evento.
- Después dos bytes con el numero de parámetros enviados, en el primer byte se envía la parte baja del numero y en el segundo la parte alta.
- Por último va el string con los parámetros que tiene el mismo formato que el mencionado antes.

4.13.9.5 - Modo de envío por parte del Kernel.

Los pasos a seguir por el kernel para enviar el evento a la interface son los siguientes :

- Arma el string con los datos que se deben enviar a cada interface.
- Por cada interface activa (usa la función *getCount* de la DLL común) hace lo siguiente :
- Solicita a windows y lockea un área de memoria de tamaño conveniente para los datos.
- Copia los datos en esta área de memoria.
- Libera (deslockea) el área de memoria reservada.
- Envía a la interface en cuestión el mensaje de windows con el que se comunican tanto la aplicación como la interface pasándole como parámetro de tipo *word* el handler del área de memoria reservada. (el handler de la interface lo obtiene de la DLL común ejecutando la función *toolHandle* y indicándole el número de interface que esta procesando)

Estos pasos se implementan en el método *sendToInter* del Kernel, notar que por cada interface se reserva un área de memoria distinta y que los datos de cuantas y cuales interfaces hay activas en cada momento los lleva la DLL común.

Esta implementación del kernel envía a todas las interfaces activas que haya en cada momento todos los eventos que se produzcan para las interfaces, es posible que sea innecesario enviar ciertos eventos hacia alguna de las interfaces, por la naturaleza de estas, por lo tanto, si esto pasa, en esas interfaces simplemente hay que ignorar estos eventos.

4.13.10 - Implementación básica de una interface.

Para hacer una interface que pueda funcionar con aplicaciones hechas a partir del modelo descripto esta tiene que cumplir ciertas pautas que detallaremos a continuación.

4.13.10.1 - Rutinas de inicialización de la interface y conexión con la hiperhistoria.

Al iniciarse la ejecución de la interface se deben llevar a cabo ciertas tareas con el objeto de conectarse con la hiperhistoria en caso de que esta ya este ejecutándose o esperar a que esta comience, los pasos que tendría que llevar a cabo en esta etapa son los siguientes :

- Usar la DLL comun a las interfaces y a las aplicaciones, de esta DLL la interface usara las siguientes funciones :
  - initlib() : word
  - registrarTool(word) : word
  - pasHandle() : word

- Usar las siguientes funciones de la DLL Kernel de Windows :
  - globalAlloc(word, dword) : word
  - globalLock(word) : pointer
  - globalUnlock(word) : word
  - globalFree(word) : word

- Determinar la existencia de la lista de nombres correspondiente a la interface que esta siendo programada, luego si esta existe podría pasarse la información que esta contiene a una estructura de datos interna de la interface a la cual se podrá acceder para determinar dado el nombre de un objeto la identificación interna de este y viceversa. Si la lista de nombres no existe se debe abortar la ejecución de la interface.

- Ejecutar la función *initLib* de la DLL común y guardar en una variable el valor que esta retorna ya que es el handler al mensaje de Windows que se usará para la comunicación entre la aplicación y la interface.

- Ejecutar la función *registrarTool* de la DLL común y pasarle como parámetro el handler de la ventana que representa a la interface el cual será necesario para que luego el kernel le envíe los eventos que se generen en la hiperhistoria. Esta función retornara cero si no está la hiperhistoria activa o el handler de la hiperhistoria en caso de que esta ya este funcionando.

- En caso de que la función anterior devuelva cero, se debe quedar la interface a la espera del mensaje convenido en el cuarto punto y cuando este llegue quiere decir que la conexión se inicio.

- En caso de que la función del quinto punto retorne distinto de cero, quiere decir que la aplicación ya esta activa, con lo que la conexión ya esta iniciada.

Una vez que la conexión se inicie, en la interface se podrán enviar eventos hacia la hiperistoria y se podrán recibir eventos de esta dirigidos a los objetos que son representación en la interface de entidades de la hiperhistoria.

4.13.10.2 - Forma de enviar eventos a la hiperhistoria.

Cuando se inicio la ejecución de la hiperhistoria y alguna interface, el usuario puede comenzar a actuar sobre esta y las acciones que realice generarán eventos que deben ser enviados al kernel para que la hiperhistoria avance, los pasos para, desde la interface, enviar eventos hacia los objetos en la aplicación son los siguientes :

- Primero se debe interpretar la acción del usuario sobre el o los objetos de interface involucrados y determinar que evento es el que debe enviarse, cual es el otro objeto involucrado, si lo hay, en el evento y con que parámetros debe mandarse. Esto no depende de la hiperhistoria sino de cada implementación particular de cada interface.
- Determinar, basándose en la tabla de nombres o en la estructura interna de datos en donde se almacenó la información, la identificación interna del objeto al cual va dirigido el mensaje.
- Se debe armar la cadena de datos que se enviará a la hiperhistoria, esta tiene el siguiente formato :
- Dos bytes para el objeto destinatario del evento.
- Una cadena de longitud arbitraria de bytes terminada en cero para el nombre del evento.
- Dos bytes para la cantidad de parámetros que lleve el evento.
- El string de parámetros con el mismo formato que el visto anteriormente.
  El primero y tercero de los items, que son números, se almacenan con la parte baja de la palabra en primer lugar y luego la parte alta de la misma.
- Se debe pedir a windows y lockear un área de memoria donde se copiarán los datos y obtener el handler de esta.
- Deslockear el área de memoria
- Enviar a la hiperhistoria el mensaje convenido y mandar como parámetro el handler al área de memoria reservada.

Estos son todos los pasos necesarios para implementar el envío de un evento desde la interface hacia la hiperhistoria, esta es la que se encarga de liberar el área de memoria reservada para el pasaje de los datos.

4.13.10.3 - Recepción de eventos desde la hiperhistoria.

Cada interface debe implementar una rutina con la que responder a la llegada del mensaje de Windows convenido, el cual enviará el kernel a cada una de las interfaces cuando haya un evento que se dirija a la representación en estas de un objeto de la hiperhistoria.

Los pasos básicos que debe implementar esta rutina son los siguientes :

- Lockear y obtener la dirección del área de memoria reservada para el pasaje de los datos desde la hiperhistoria, el handler a dicha área de memoria se pasa en el parámetro word del mensaje Windows.
- Obtener la identificación interna del objeto hacia el cual va dirigido el evento, el nombre del evento, la cantidad de parámetros y cada uno de ellos, si existen, el formato de los datos almacenados aquí es el mismo que el visto arriba en el punto "*Forma de enviar eventos a la hiperhistoria.*".
- Deslockear y liberar el área de memoria de intercambio de datos.
- Luego, en este punto, la implementación de lo que hay que hacer con los valores obtenidos depende del estilo de interface, pero en general se debe hacer que el objeto que representa en la interface al enviado en la identificación interna responda mediante un procedimiento, método, etc. al evento enviado, lo cual ocasionará un cambio en la interface que será, generalmente, visible al usuario o que actualizará valores que influirán en como se verá esta.

4.13.11 - Envío de eventos al Timer.

Ahora veremos el modo en que los envíos de eventos al Timer son implementados y como hace este para enviar estos eventos hacia el objeto destinatario de los mismos. El Timer se debe tratar como si fuese otro objeto más de la hiperhistoria al cual se le pueden enviar eventos, los cuales tienen ya un comportamiento predefinido.

Cada evento enviado al timer se traduce como una llamada a una de los métodos que este objeto implementa, por lo tanto la traducción a lenguaje Pascal de esto es sencilla.

Luego, cuando se cumple el tiempo fijado para alguno de los objetos que tengan un registro en este, el timer envía los eventos correspondientes a la cola de eventos y luego esta se encargará de despacharlos cuando les llegue el turno.

Con esto se termina la explicación de como se implementa el kernel de la hiperhistoria, luego veremos un ejemplo de implementación de la interface hecha en una herramienta particular pero que servirá de guia para la implementación de interfaces es cualquier herramienta que provea las facilidades mínimas.

CAPITULO V.

HERRAMIENTAS PARA LA IMPLEMENTACION DE HIPERHISTORIAS.

Introducción.

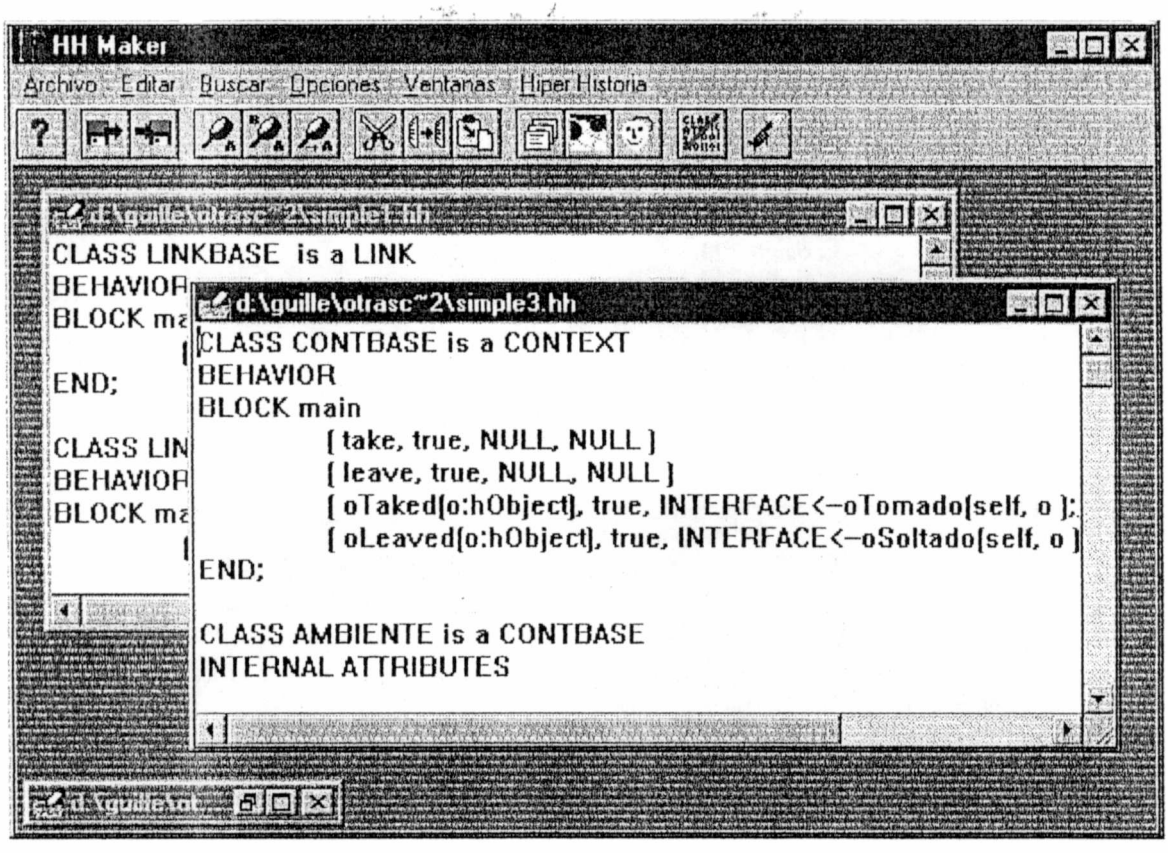
este capitulo veremos las herramientas mínimas necesarias para la implementación de las hiperhistorias. Con ellas se realizar una descripción de manera textual y, luego, generar a partir de ella una aplicación ejecutable la que comunicada rde con la interface correspondiente conformaran una hiperhistoria completa.

Traductor.

traductor permite escribir una descripción textual de la hiperhistoria mediante un sencillo editor que forma parte de éste mediante el mismo traductor se deben especificar, en un archivo separado, los módulos con los que contará el código de la storia y luego, al indicarle que compile el texto, este generará la aplicación ejecutable y el archivo de nombres necesaric posterior comunicación con la interface.

- Interface del traductor, editor.

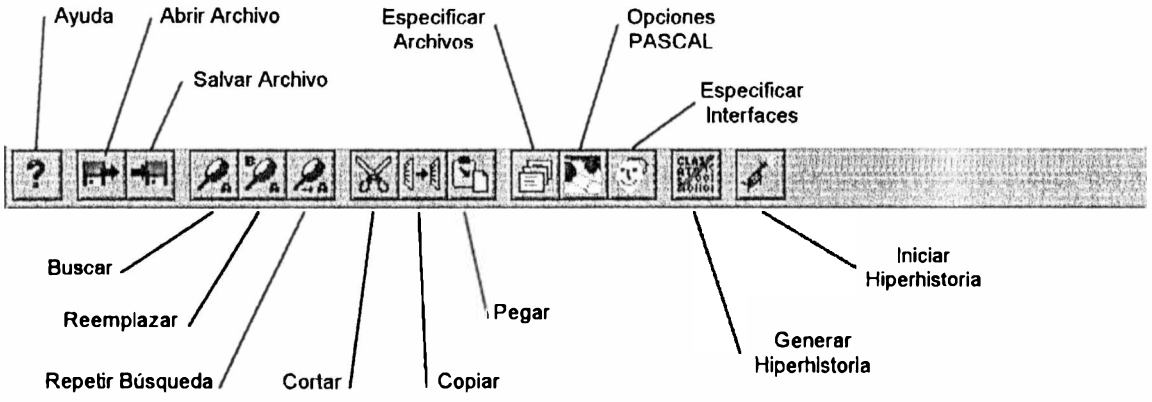
ta es la parte visible del traductor, esta consiste en una aplicación multiventanas en la cual se pueden editar los distintos de la hiperhistoria, el aspecto de esta interface es el siguiente :



- Uso.

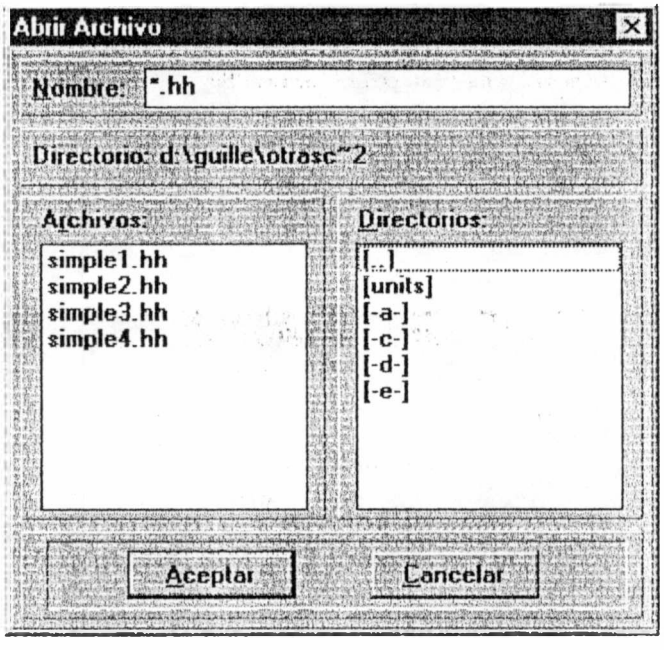
te compilador es un simple editor con el agregado de las opciones para hacer la compilación y para ejecutar las interface manejara con las opciones en la barra de menú, alguna de las cuales de pueden acceder de manera mas rápida con le botones en la parte superior de la ventana.

Aquí vemos como esta dispuesta la barra de botones del editor y para que sirven cada uno de los botones en ella.

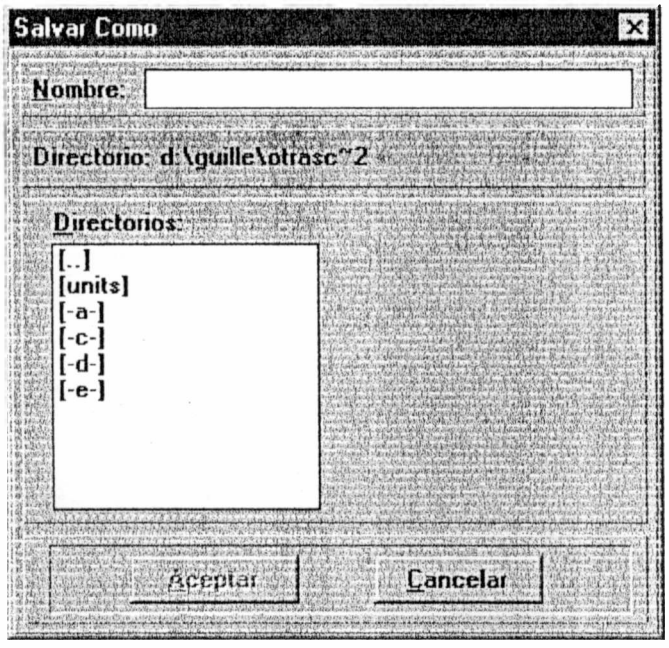


Las diferentes funciones del editor son las siguientes :

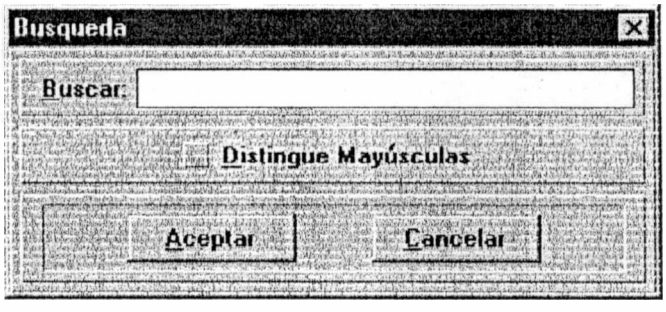
- Abrir archivo : abre una ventana de diálogo con la que se puede abrir un archivo de texto que puede ser un fuente de la hiperhistoria o un archivo de configuración de la misma o cualquier archivo de texto, el cuadro de diálogo que muestra es el siguiente:



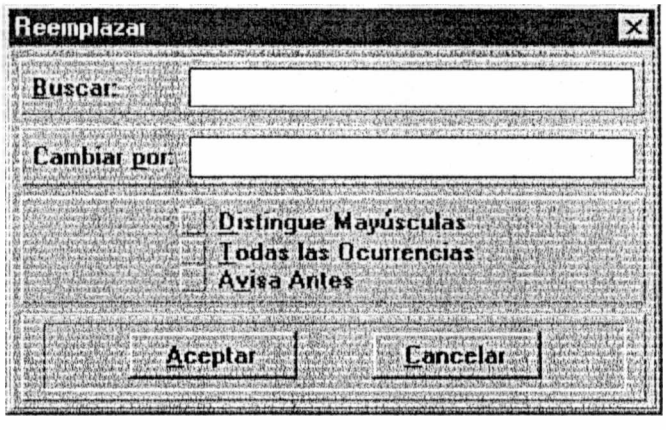
- Salvar Archivo : salva el contenido del archivo en disco, se accede con la opción del menú Archivo | Guardar o desde la barra de botones, al hacerlo, si el archivo tiene nombre, lo guarda, sino muestra un cuadro de diálogo como el siguiente para que se ingrese el nombre con el cual se quiere que se grabe el archivo.



- Buscar: Realiza la búsqueda de una cadena de caracteres en la ventana activa del editor, el cuadro de diálogo que muestra par solicitar la cadena a buscar es el siguiente.

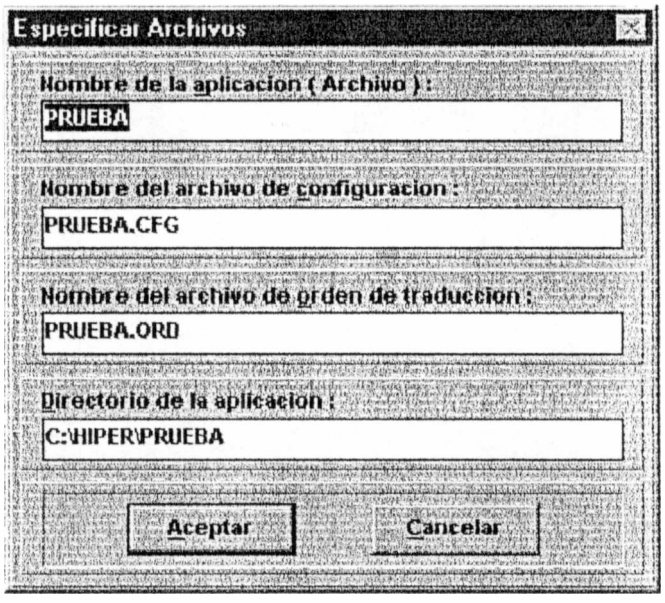


- Reemplazar: Busca la palabra especificada y la reemplaza por lo que se indique en el cuadro de diálogo que se abre cuando se elige esta opción, dicho cuadro se ve como el que sigue.



- Repetir Búsqueda: Repite la última búsqueda realizada a partir de la posición del cursor de texto.
- Cortar :Corta el texto seleccionado en la ventana activa y lo guarda en el portapapeles de Windows.
- Copiar : Copia el texto seleccionado en el portapapeles de Windows.

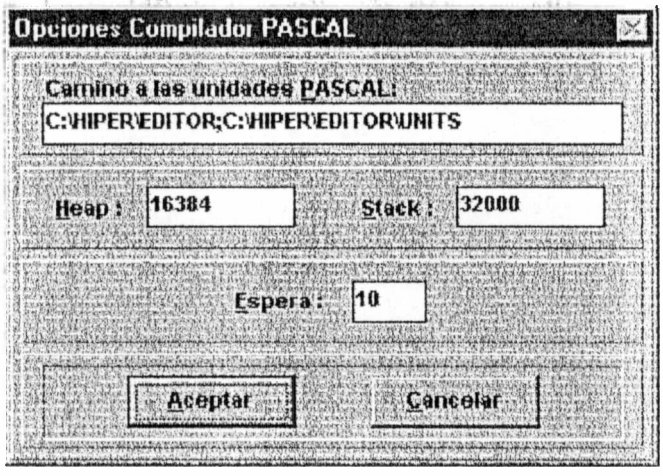
- Pegar : Copia el contenido del portapapeles en la posicion actual del cursor en la ventana activa.
- Especificar Archivos : abre una ventana de diálogo como se muestra en el siguiente gráfico:



En esta ventana se ven los siguientes campos que se refieren a datos de la aplicación a traducir.

1. Nombre de la aplicación (Archivo) : Este nombre se usa para decir al traductor que nombre llevará el ejecutable de la aplicación, no poner la extensión.
2. Nombre del archivo de Configuración : nombre del archivo de configuración que generara la aplicación y que contendrá una lista de los objetos para que sea usada por las interfaces.
3. Nombre del archivo de orden de traducción: Nombre de un archivo de texto que debe contener en cada línea el nombre con la extensión de cada uno de los archivos que se deben compilar para generar la aplicación y en el orden en que deben ser traducidos.
4. Directorio de la aplicación : directorio en el cual estarán los fuentes de la hiperhistoria y los demás archivos que se hacen referencia mas arriba y donde el editor grabara el ejecutable de la aplicación y el archivo de configuración.

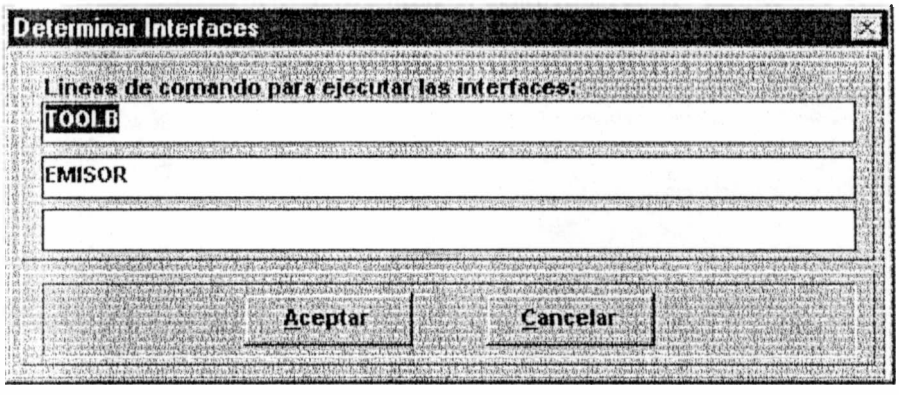
- Configuración Pascal : Con esta opción se abre un cuadro de diálogo como el siguiente:



En el se solicitan los siguientes datos:

1. Camino a las unidades PASCAL : directorios donde debe buscar el compilador pascal las unidades necesarias para generar la aplicación.
2. Heap : Tamaño del área de alocación dinámica que se quiere que se reserve para la aplicación final.
3. Stack : tamaño que se quiere que tenga la pila de ejecución de la aplicación final, agrandar este valor si al ejecutar la aplicación da el error stack fault.
4. Espera : Cantidad de veces que el traductor preguntara por la culminación de la compilación Pascal de la aplicación, pasada esta cantidad de veces, se asumirá que hubo un error en la compilación.

- Especificar Interfaces : Con esta opción se abre un cuadro de diálogo como el siguiente:



En el se tienen tres campos de texto donde se debe especificar en cada uno la línea de comando para iniciar una de las interfaces que se ejecutaran con la aplicación, esta opción da la facilidad de especificar las interfaces y luego, simplemente pulsando un botón, arrancará la aplicación y todas las interfaces especificadas. Se pueden especificar desde una hasta tres interfaces simultáneas.

- Traducción : Con esta opción se ejecuta el proceso que toma los fuentes y genera código Pascal, luego ejecuta el compilador Pascal para generar la aplicación ejecutable y por último ejecuta esta aplicación en un modo especial con el objeto de generar el archivo de configuración para las interfaces.
- Ejecución de la aplicación: Con esta opción se ejecuta la aplicación junto con todas las interfaces especificadas con la opción anterior.

5.2.1.2 - Configuración.

Para generar una hiperhistoria, hay que personalizar el editor para que acceda en forma correcta a los fuentes de la misma y a los unidades PASCAL necesarias para generar el ejecutable de la misma, los pasos para implementar una hiperhistoria usando este editor son los siguientes:

- Abrir el diálogo **Especificar Archivos** e indicar el nombre de la aplicacion, el nombre del archivo de configuración, el nombre del archivo de orden de traducción y el directorio de la aplicación.
  Un ejemplo del contenido del archivo de orden de traducción puede ser el siguiente:

```
LINKS.HH
CONTEXTOS.HH
ENTIDADES.HH
CONJUNTOS.HH
PRINCIPAL.HH
```

Estos son archivos en cada uno de los cuales hay codigo en el lenguaje del modelo definido anteriormente, el traductor procesa estos archivos en el orden en que están especificados, por lo tanto, si en un archivo se hace referencia a una declaración que esta en otro archivo, este debe colocarse antes en la lista de orden de traducción.

- Escribir los fuentes de la hiperhistoria y grabarlos en el directorio que se indicó para la aplicación.
- Abrir el diálogo de **Configuración PASCAL** y especificar los directorios donde el compilador PASCAL debe buscar para encontrar las unidades necesarias, el tamaño del heap, el tamaño del stack y la cantidad de ciclos de espera. Estas opciones son comunes a cualquier hiperhistoria.
- Generar la hiperhistoria una vez que se escribieron los fuentes.
- Buscar en el directorio de la aplicación el archivo de configuración de la hiperhistoria y completarlo como corresponda escribiendo los nombres de los objetos de inteface que representan cada objeto del kernel. Este paso no debe ser repetido para posteriores traducciones de la misma hiperhistoria a menos que se agregen objetos, en este caso se debe especificar en el archivo el nombre de los objetos de interface para los objetos del kernel nuevos.
- En caso de querer ejecutar en ese momento el kernel., abrir el diálogo de **Especificar Interfaces** y escribir las lineas de comando, una o mas, de las interfaces que se vayan a usar con él, si no se tiene desarrollada ninguna interface, se puede ejecutar el kernel y probarlo invocando las interfaces genericas, simplemente especificando en uno de los campos TOOLB.EXE para la interface receptora y EMISOR.EXE para la interface emisora de eventos, en cualquier orden.
  Las aplicaciones que inicia el editor son independientes de este, por lo tanto una vez que arranquen, puede terminarse la ejecución del editor si asi se lo desea.

5.2.1 - Implementación del traductor.

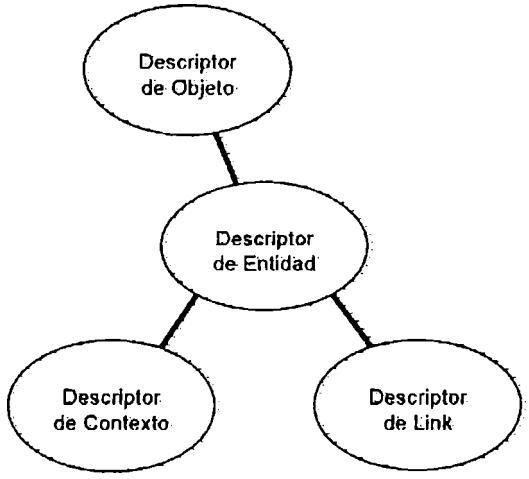
Una vez que el fuente de la hiperhistoria se escribió y al iniciar el proceso de traducción, el traductor leerá todos los fuentes que se le haya indicado que formaban parte de la hiperhistoria y generara a partir de estos el código Pascal necesario.

Luego el mismo traductor ejecutará el compilador Pascal, el cual usando los fuentes generados por el traductor y las librerías Pascal especificas de las hiperhistorias, además de las necesarias del lenguaje, generará un archivo ejecutable.

Si este proceso termino sin error, el mismo traductor correrá el ejecutable generado en un modo especial, con lo cual este generara el archivo de nombres de los objetos de hiperhistoria internos de la aplicación, necesario luego para la comunicación con la interface.

5.2.1.1 - Jerarquía interna de objetos del traductor.

El traductor, para llevar a cabo su proceso de traducción, generará descriptores internos de cada uno de los objetos de la hiperhistoria definidos en los textos leídos, para esto, tiene una jerarquía básica de descriptores, la cual es similar a la jerarquía base de objetos de la hiperhistoria en la aplicación final. La jerarquía de descriptores básicos predefinidos es la siguiente :



De estos objetos indicados en el gráfico se desprenderán los que se generen a partir de la descripción textual, una vez que esta jerarquía este completa, es decir, cuando se terminen los fuentes a procesar, se genera a partir de la estructura de descriptores conformadas el código Pascal necesario para generar la estructura de objetos descripta en la aplicación ejecutable.

5.2.1.2 - Generación de código Pascal.

El traductor genera código Pascal en dos etapas, primero, a medida que va leyendo los archivos fuente, va grabando en un archivo temporal la traducción de las reglas que va leyendo, luego, una vez que genero los descriptores de todos los objetos que conforman la hiperhistoria, vuelca en una unidad pascal el código necesario para realizar las declaraciones de cada una de las clases y tambien graba la implementación de algunos métodos de cada clase segun los parametros que contengan los descriptores de cada uno de los objetos.

El siguiente es un ejemplo del codigo que se genera a partir de los descriptores:

```
PUERTA = ^_PUERTA;
_PUERTA = object( _LINKBASE )
        ABIERTA : BOOLEAN;
        constructor _init( oid, ocl : string; enContext : hObject );
        function _aceptaEvento(_event:_tEvent;sender:hObject;var _params:string):boolean; virtual;
        procedure _connect( cConx:string; anObject:hObject ); virtual;
        function _rg3(_event:_tEvent;sender:hObject;var _params:string):boolean;
        function _rg4(_event:_tEvent;sender:hObject;var _params:string):boolean;
        function _rg5(_event:_tEvent;sender:hObject;var _params:string):boolean;
        function _rg6(_event:_tEvent;sender:hObject;var _params:string):boolean;
end;
```

Este es la declaración de un objeto puerta que sale de los descriptores, en ella se van a especificar los atributos que se hayan declarado en la definición del modelo y ademas el encabezado de los metodos que tendra esta clase, los tres primeros (_init, _aceptaEvento y _connect) son similares en todas las entidades, contextos y links, los encabezados siguientes corresponden uno por cada regla que tenga definida.

Luego se graba la implementación de los tres primeros métodos que serán como sigue:

```
constructor _PUERTA._init( oid, ocl : string; enContext : hObject );
        var aux : hObject;
begin
        _LINKBASE._init( oid, ocl, enContext );
        ABIERTA := FALSE;
        _bloques:= new( dictionary, init( 1,1 ) );
        _padres := new( pIntCollection, init( 1,1 ) );
        _hijos:= new( pIntCollection, init( 1,1 ) );
        _siguientes := new( pIntCollection, init( 1,1 ) );
        _flags:= new( pBolCollection, init( 1,1 ) );

        _bloques^.atPut( 'MAIN', '' );
        _padres^.atPut(0,-1);
        _hijos^.atPut(0,-1);
        _siguientes^.atPut(0,-1);

        self._activarBloque('MAIN');
end;

procedure _PUERTA._connect( cConx:string; anObject:hObject );
begin
        anObject^._makeConnWith( addr(self) )
end;

function _PUERTA._aceptaEvento(_event:_tEvent;sender:hObject;var _params:string):boolean;
var sigue : boolean;
begin
        sigue := true;
        self._funcionDebug( _event, sender, _params, 'PUERTA' );
        if _flags^.at( 0 ) then begin
            if sigue then begin
                sigue := false;
                if _rg3( _event, sender, _params ) then
                    else if _rg4( _event, sender, _params ) then
                        else if _rg5( _event, sender, _params ) then
```

```
                    else if _rg6( _event, sender, _params ) then
                        else sigue := true
            end
        end;
        if sigue then
          sigue := _LINKBASE._aceptaEvento(_event,sender,_params );
        _aceptaEvento := not sigue
    end;
```

El constructor _init inicializa el objeto, este método se ejecutará cuendo arranque la hiperhistoria, lo que hace es inicializar las estructuras internas del objeto y sus atributos. Cada objeto tendra definido este procedimiento.

El procedimiento _connect es utilizado cuando en la inicialización de un contexto se quiere conectar a esta entidad con un Channel o con otra entidad.

Por último la función _aceptaEvento es ejecutada cada vez que el objeto recibe un evento, en esta función está representada la estructura de bloques del comportamiento del objeto.

Lo último que se graba es la implementación de cada una de las reglas, un ejemplo de como se veria una de ellas es el siguiente :

```
function _PUERTA._rg3(_event:_tEvent;sender:hObject;var _params:string):boolean;
    var
        LASTCONTEXT : HOBJECT;
        _paramStr, _xxStr : string;
        ok : boolean;
begin
    LASTCONTEXT := NIL;
    _rg3 := false;
    if _event = 'IACTIVAR' then
        begin
            _xxStr := _params;
            if ( NOT ABIERTA ) then
                begin
                    ABIERTA := TRUE;

                    _paramStr := '';
                    ok := self._aceptaEvento( 'OBJFIRST', addr(self), _paramStr );

                    _paramStr := '';
                    _paramStr := _paramStr + _strHOBJECT( LASTCONTEXT);
                    ok := self._aceptaEvento( 'OBJCURR', addr(self), _paramStr );
                    LASTCONTEXT := _getHOBJECT( _paramStr );

                    _paramStr := '';
                    _paramStr := _paramStr + _strHOBJECT( LASTCONTEXT);
                    ok := Kernel^.sendToInter( _internalId, 'MEABREN', _paramStr, 1 );

                    _rg3 := true;
                    _params := '';
                end
                    else
                        _params := _xxStr
        end
end;
```

En cada regla se pregunta por el evento correspondiente (en el primero de los if's), luego se pregunta por la precondición de la regla y luego una traducción de cada una se las sentencias especificadas en el comportamiento del objeto en el lenguaje del modelo.

Por cada regla que se escriba en la implementación de la hiperhistoria se generará una función como la anterior, todas estas reglas son llamadas solamente desde la función *_aceptaEvento* de cada objeto y cada función de las anteriores corresponde a una sola regla de un solo objeto.

Por último el traductor genera un pequeño módulo que será el programa principal de la aplicación, este será generado en el directorio de la aplicación, es similar para todas las hiperhistorias y se ve como sigue :

```
program SIMPLE;
    uses
        miWCrt, WinProcs, WinTypes, Objects, Strings, nlassdec, import, HHUnit;
begin
    KERNEL := new( ptKernel, _init );
    TIMER := new( ptTimer, _init );
    QUEUE := new( ptQueue, _init );
    NULL := new( ptObjetoNulo, _init( '','NULL', NIL ) );
    initExternalContext;
    if paramCount = 0
        then
            begin
                if KERNEL^.archConfig('SIMPLE.CFG')
                    then
                        KERNEL^.start
            end
        else
            KERNEL^.archConfig('SIMPLE.CFG');
    doneWinCrt
end.
```

Este código simplemente lo que hace es inicializar los componentes principales de la hiperhistoria, inicializar todos los objetos de la hiperhistoria mediante la ejecución del procedimiento *initExternalContext* el cual va inicializando hacia adentro todos los objetos y luego, si se lo ejecuta con algún parámetro en la línea de comandos, genera el archivo de nombres para la interface y si no inicia la hiperhistoria.

5.2.1.3 - Compilación de los módulos en Pascal.

Una vez generados los módulos que se mencionaron en la sección anteiror, se ejecuta el compilador Pascal, el cual compilará estos y luego los linkeará con unidades ya generadas y unidades ya definidas de Pascal, para generar el archivo ejecutable.

La ejecución del compilador Pascal la efectuará automáticamente el editor de hiperhistorias y usará para ello las opciones definidas en el diálogo de configuración Pascal.

En caso de que la compilación Pascal falle, se mostrará un cartel que indica que hay un error en dicho proceso y el ejecutable no fue generado.

5.2.1.4 - Generación del archivo de nombres para la interface.

Una vez que la compilación termino de manera exitosa, el editor ejecuta la nueva aplicación enviándole un valor como parámetro en la linea de comandos, con esto la aplicación, en lugar de iniciar la ejecución de la hiperhistoria, busca en el directorio de la aplicación si esta el archivo de configuración y si no está lo genera.

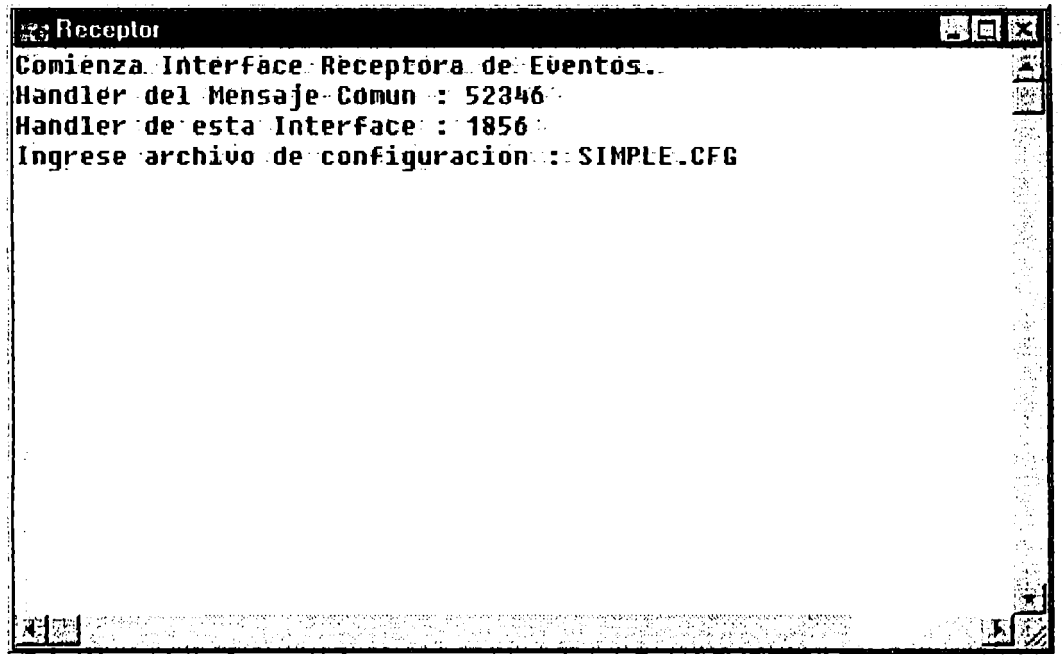
Si el archivo de configuración existe, lo lee, determina en el que nombres se les dio a cada objeto y genera uno nuevo en el cual si un objeto ya tenia nombre en el antiguo, se le asigna el mismo nombre en el nuevo, en caso de que sea un objeto nuevo, se deja sin nombre de interface, para que el usuario lo complete y en caso de que en la aplicacion haya desaparecido algún objeto, este no aparecerá en el nuevo archivo de configuración.

5.3 Interface genérica textual receptora de eventos.

Esta interface esta hecha con el objeto de realizar pruebas sobre la aplicación, o sea, el kernel de la hiperhistoria, sin tener que desarrollar una interface especialmente para esto.

Esta interface es una ventana de texto, al iniciar, pide que se le ingrese el nombre del archivo de configuración donde se encuentra la lista de objetos con sus correspondientes nombres de interface.

Asi se ve la pantalla al iniciar la ejecución.



El handler del mensaje común que muestra la ventana es el número que le asigna Windows al mensaje que usarán todas las interfaces y el kernel para comunicarse.

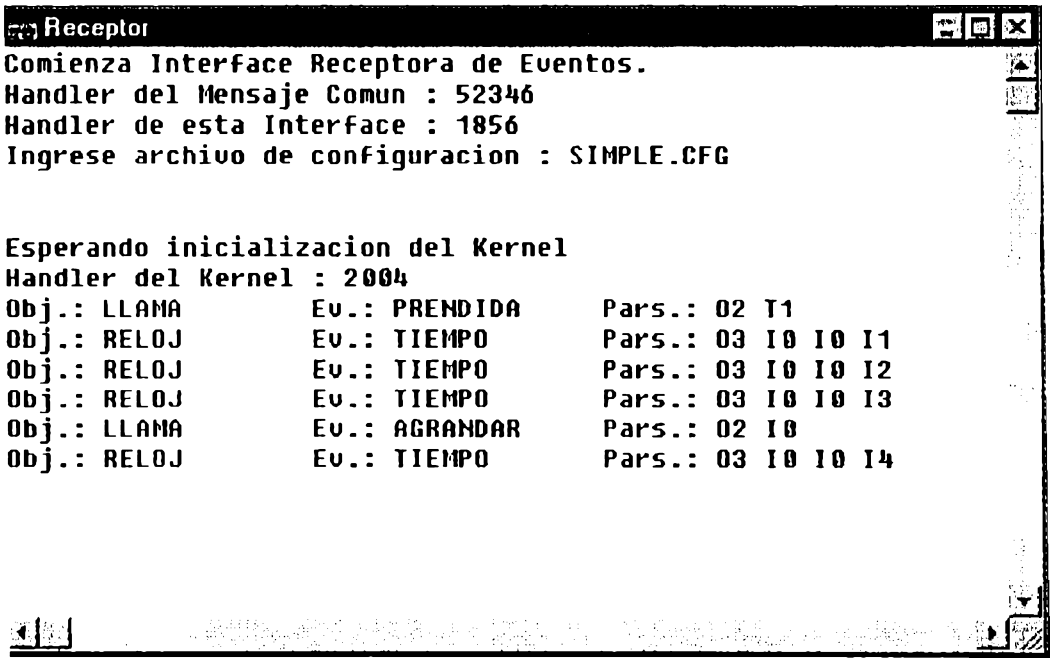
Luego muestra el handler de la ventana de esta interface y luego solicita el ingreso del nombre del archivo de configuración que debe usar para darle nombre a los objetos del kernel desde los cuales reciba eventos.

Una vez que el niombre del archivo de configuracion es ingresado, la interface se queda a la espera que arranque el kernel de la hiperhistoria o, si este ya esta activo, arranca directamente mostrando todos los eventos que recibe de este.

Lo primero que muestra es el handler del kernel y luego empieza a mostrar los eventos a medida que lleguen, los datos de estos que muestra son los siguientes:

- Nombre de interface del objeto en caso que se haya especificado uno o el número de identificación interna del objeto, en caso que el objeto no tenga definido el nombre de interface.
- Evento que el objeto de interface recibe.
- Lista de los parámetros que recibe, cada párametro esta formado por un cáracter que indica el tipo del párametro y luego el valor del mismo.

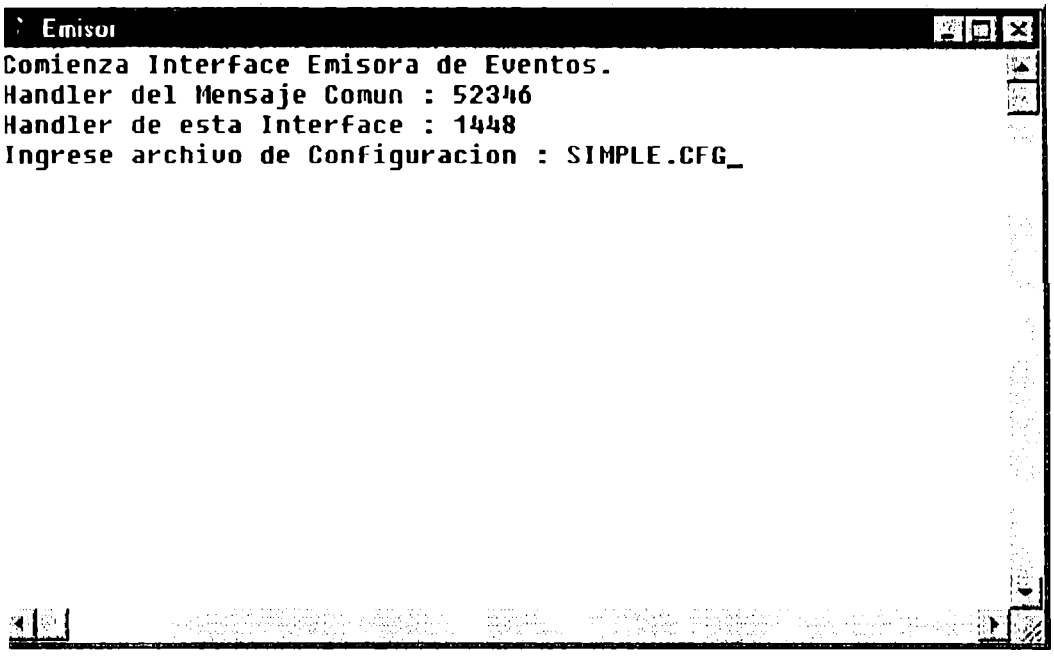
La ventana, una vez que arranco y empieza a recibir eventos se ve como sigue:



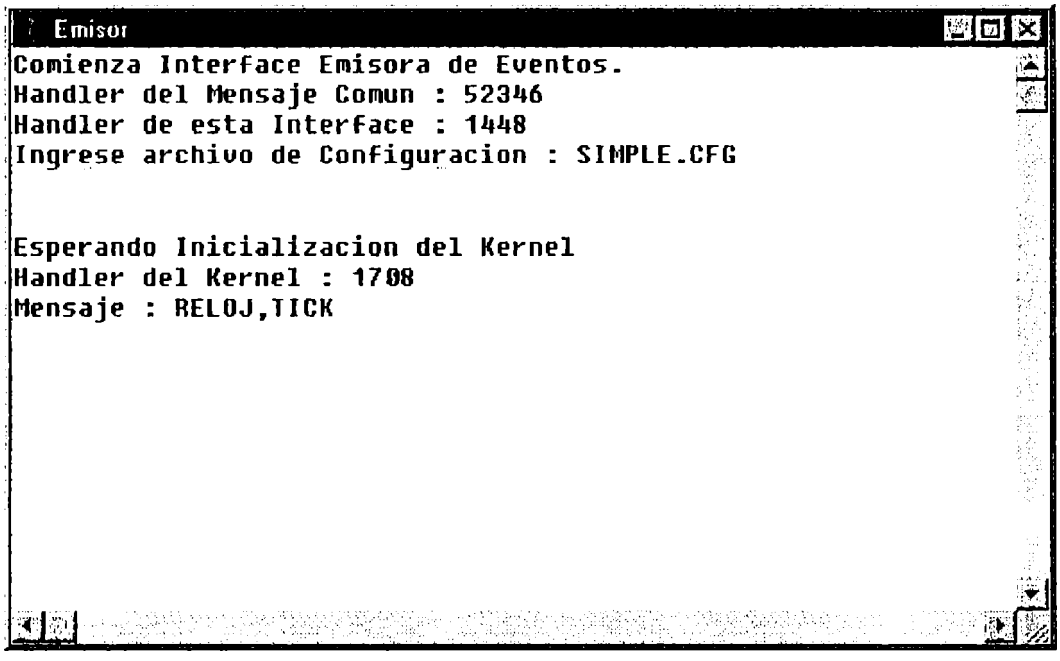
ace genérica textual emisora de eventos.

erface es similar a la anterior en lo que respecta a su apariencia, pero la diferencia es que esta sirve para
s objetos de la hiperhistoria que están en el kernel, de manera de poder actuar sobre ellos y realizar prue
nto.

sma manera que el anterior, lo primero que pide es el nombre del archivo de configuración de la interface, la v
ve como sigue :



que se ingresa el nombre del archivo de conifiguración, la interface queda lista para enviar eventos al kerne
para ello el usuario debe tipearlos a la derecha del prompt que indica "Mensaje :", la ventana en ese momento



El formato en el que se deben escribir los mensajes es el siguiente :

1. Nombre del objeto destino del evento.
2. Nombre del evento.
3. Nombre del otro objeto involucrado si lo hay .
4. Parametros, primero una letra que indica el tipo y luego el valor del parametro.

Este es la última de las herramientas desarrolladas para ayudar al desarrollo de hiperhistorias, estas son herramientas básicas y no proveen facilidades como para desarrollar aplicaciones demasiado complejas ya que no poseen capacidades de diseño de la hiperhistoria. Mas adelante veremos que herramientas sería deseable tener para el diseño y el desarrollo mas sencillo de hiperhistorias.

CAPITULO VI.

EJEMPLO DE IMPLEMENTACION.

6.1 - Introducción.

En este capítulo veremos un ejemplo de la implementación de una hiperhistoria, en este caso la interface esta realizada con la herramienta de autoría llamada Toolbook. En el ejemplo veremos como se escribió el código fuente, en que orden se fueron codificando los distintos componentes, como se fueron integrando para llegar a la implementación final y como se implementó la interface en la herramienta mencionada.

6.2 - Diseño de la hiperhistoria.

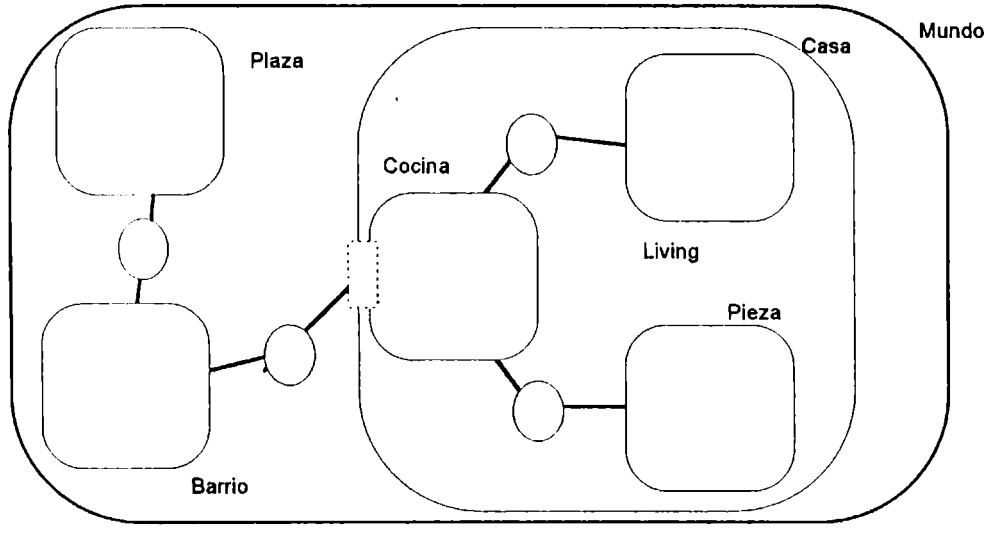
6.2.1 - Historia.

En este ejemplo se tomará como base una historia en la cual el personaje habita en una casa, que esta dentro de un barrio, la cual tiene una chimenea cuyo fuego se esta extendiendo, el objetivo del personaje es apagarlo antes de que este termine por incendiar la casa, para ello debe encontrar un vaso de agua con que apagarlo, una de las puertas del interior de la casa se encuentra cerrada con llave, por lo que, para poder acceder al interior de ese cuarto, el lector debera también buscar la llave correspondiente.

La urgencia de la situación a resolver hará que el lector, teniendo un objetivo claro, efectúe una determinada secuencia de acciones que lleven a la culminación del objetivo, con lo cual deberá aprender ciertos conceptos y usar ciertas habilidades, como ser la de orientarse en el espacio, al tener que navegar distintos ambientes en busca de los elementos necesarios, deberá ejercitar su percepción temporal, al darse cuenta de que dispone de un tiempo limitado, etc.

6.2.2 - Estructura de navegación.

El siguiente gráfico describe como sería la estructura de navegación en la que transcurrirán las acciones de la hiperhistoria, en el se ven los contextos en los cuales podrá habitar el personaje y su comunicación a través de los links que representarán a puertas, pasajes, etc.



6.2.3 - Entidades.

Las entidades principales de la hiperhistoria son las siguientes :

- El personaje, se encuentra inicialmente en el living, este podra recorrer toda la estructura de navegación guiado por el lector.
- El fuego, se encuentra en la chimenea del living, va creciendo conforme pasa el tiempo y solo puede ser apagado cuando sobre el se le arroja el vaso de agua.
- El vaso de agua, se encuentra inicialmente en la pieza, esta entidad puede ser transportada, al igual que algunas otras, por el protagonista, también puede ser usada para combinar con otras entidades, como por ejemplo el fuego, para conseguir algún efecto.
- La llave de la puerta de la pieza, se encuentra en la plaza, el lector debe hallarla para poder abrir la puerta de la pieza en la que se encuentra el vaso.

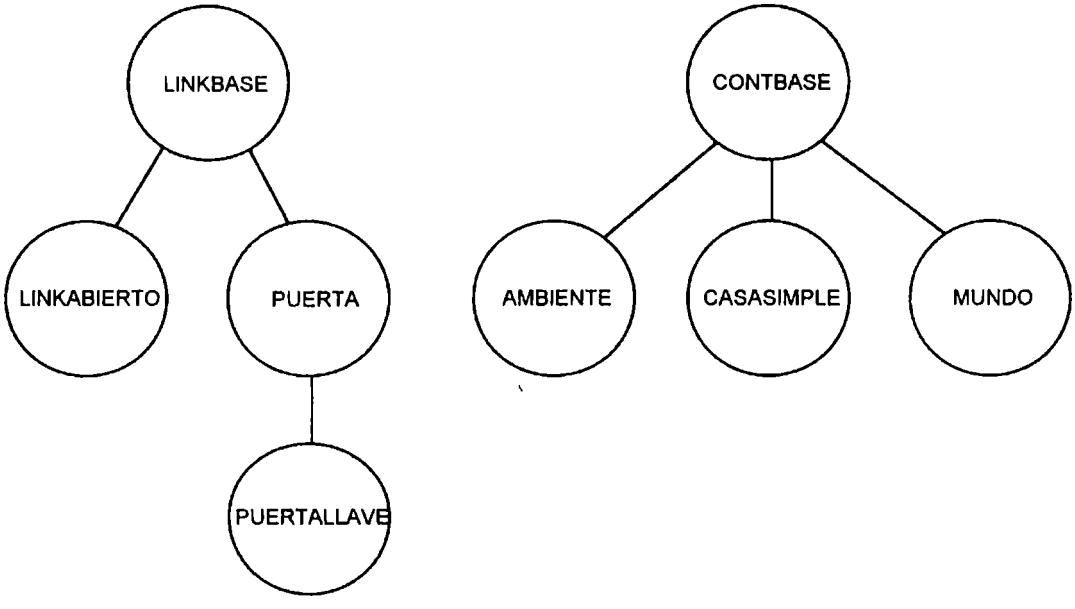
Luego existen otras entidades que el lector puede manipular, como ser una pelota, las llaves de luz, etc, que no hacen al nudo principal de la historia.

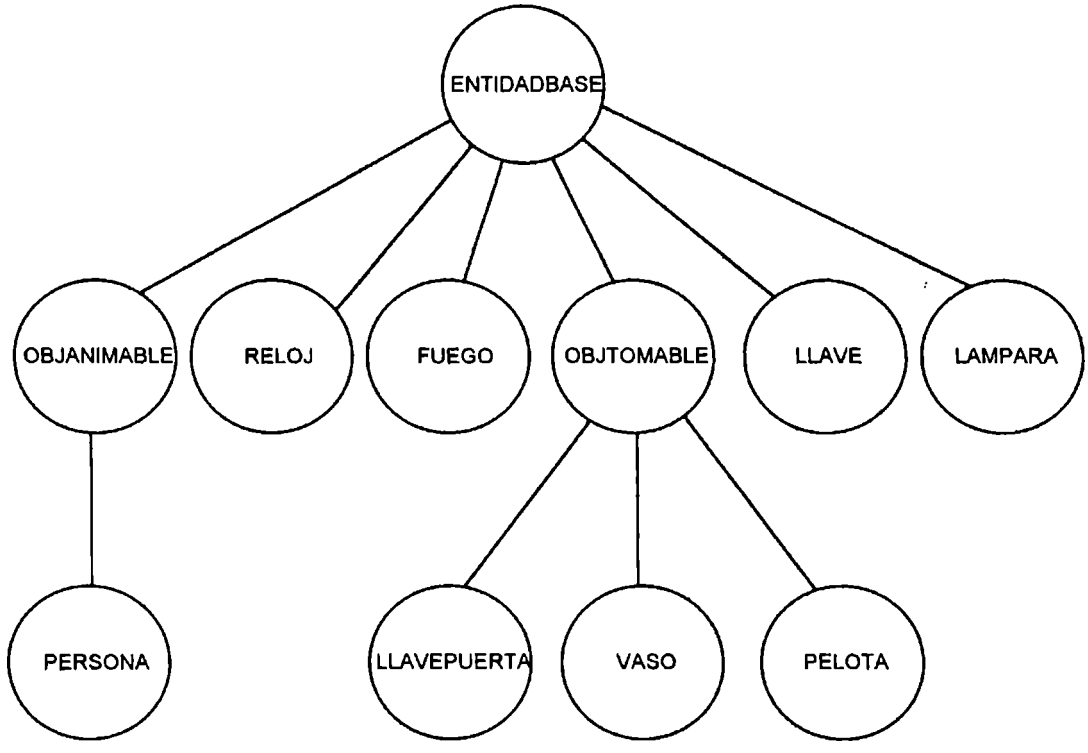
6.3 - Implementación en el lenguaje del modelo.

6.3.1 - Jerarquía de clases.

La siguiente es una jerarquía de las clases básicas que se definen, además de las ya vistas que son predefinidas en una hiperhistoria, para la implementación de la historia que describimos arriba.

Esta estructura incluye las clases que describen a todos los objetos que intervienen en la hiperhistoria, incluyendo contextos y links.





En los tres gráficos precedentes se observa que hay una clase superior para cada caso la cual sera subclase, según corresponda, de las clases de contextos, links y entidades predefinidas, esto es así porque cada uno de las subclases de cada tipo implementadas necesitan algún comportamiento común especifico de esta hiperhistoria el cual no se provee de manera predefinida.

6.3.2 - Implementación de los links.

El primer paso de la implementación de la hiperhistoria es la definición de la estructura de navegación, para esto se debe comenzar con la programación de los links que luego serán usados para la comunicación de entre los contextos de la estructura mencionada.

En este punto tenemos que hacer una aclaración, por la manera que tiene la herramienta que utilizamos para la implementación de la interface de nombrar los objetos, no podíamos referirnos a estos con un nombre único sino que debíamos especificar también el contexto en el cual se encontraban para que pudiese determinar unívocamente el objeto al cual nos referíamos cuando enviamos un mensaje hacia la interface, por lo tanto no utilizamos los atributos normales en la implementación de los objetos de la hiperhistoria, los cuales envían de manera implícita un evento a la interface cuando cambian de valor, sino que toda comunicación con la interface la haremos de manera explícita y los atributos que usaremos para la implementación de los objetos serán siempre internos.

La siguiente es la implementación de un link básico, este siempre deja cruzar al objeto que quiere hacerlo, es decir, siempre esta abierto.

```
CLASS LINKBASE  is a LINK
BEHAVIOR
    BLOCK main
        ( asoc, true, local clase : string; sender<--getClass( clase ); self<--intAsoc( clase );, NULL )
    END;
```

```
CLASS LINKABIERTO is a LINKBASE
BEHAVIOR
    BLOCK main
        ( cross, true, self<--crossby( sender );, NULL )
    END;
```

La implementación de este link básico consta de dos clases, la primera implementa a una clase de link básica que no sirve para conectar dos contextos pero tiene la implementación de la regla que recibe el mensaje de asociación y resuelve la clase del objeto con el que se quiere asociar y luego se envía a si mismo un mensaje de asociación para que lo resuelva alguna de sus subclases.

La segunda clase es el link que, al conectar dos contextos, permite que el personaje pase de uno a otro siempre, lo único que implementa es una regla que efectúa esta acción. Como ninguna asociación tiene efecto sobre este link, este no responde al evento *intAsoc*, en caso de recibirlo, lo ignora.

Luego las dos clases que siguen son el código de, primero, una puerta, la cual puede estar abierta o cerrada y según sea como esté, dejará que un personaje o entidad que quiera cruzarla lo haga o no, la puerta del segundo lugar permita además la asociación con una llave, la cual cambiará el estado de cerrada o no cerrada, no permitiendo abrirla si esta en el primero de los estados.

```
CLASS PUERTA is a LINKBASE
    INTERNAL ATTRIBUTES
        abierta : boolean;
    INITIALIZATION
        abierta := false;
BEHAVIOR
    BLOCK main
        ( iActivar, not abierta,    LOCAL lastContext : hObject;
                                    abierta := true;
                                    self<--objFirst;
                                    self<--objCurr( lastContext );
                                    INTERFACE<--meAbren( lastContext );, NULL )
        ( iActivar, abierta,    LOCAL lastContext : hObject;
                                abierta := false;
                                self<--objFirst;
                                self<--objCurr( lastContext );
                                INTERFACE<--meCierran( lastContext );, NULL )
        ( cross, abierta, self<--crossby( sender );, NULL )
    END;

CLASS PUERTALLAVE is a PUERTA
    INTERNAL ATTRIBUTES
        cerrada : boolean;
    INITIALIZATION
        cerrada := true;
BEHAVIOR
    BLOCK main
        ( intAsoc( c:string ), c='LLAVEPUERTA', if not abierta then ( cerrada := not cerrada; ) ;, NULL )
        ( iActivar, cerrada, NULL, NULL )
    END;
```

En la primera clase se define un atributo que indica el estado de la puerta, si esta abierta o cerrada, y tres reglas en el bloque principal, el cual estará siempre activo.

Estas reglas hacen lo siguiente, la primera maneja el evento de interface *iActivar* para cuando la puerta no esta abierta y lo que hace es poner en true el atributo que indica si esta abierta o no y luego averigua uno de los contextos que comunica la puerta y envía el evento de notificación a la interface con el nombre de uno de los contextos en donde está la puerta como parámetro. Luego veremos como se debe implementar la representación del link en la interface Toolbook.

La segunda regla es similar a la primera con la diferencia que se acepta sólo cuando la puerta está abierta y pone el atributo que lo indica en false.

La última regla hace que un objeto cruce el link, observar que este objeto no se puede poner como subclase de *LINKABIERTO* ya que si lo hiciéramos así, al recibir la puerta el evento *cross* estando cerrada, por la manera en que se atienden los eventos, seguirla buscando el evento en la superclase y como lo encuentra con la precondición en true, lo hace cruzar.

La segunda de las clases implementa la puerta que se abre con la llave, tiene dos reglas en el bloque *main* las cuales hacen lo siguiente: la primera implementa la asociación con la llave, al hacerlo, cambia el estado de cerrada a no cerrada.

La segunda de las reglas acepta el evento *iActivar* cuando la puerta esta cerrada con llave, con esto, se interrumpe la búsqueda del evento en la superclase y no ejecuta la regla que abre la puerta.

Con estas clases definimos los links que se usarán para el armado de la estructura de navegación, lo cual haremos mas adelante.

6.3.3 - Implementación de las entidades.

Aquí veremos como serán programadas las clases de las entidades que habitarán los distintos contextos de la hiperhistoria.

```
CLASS ENTIDADBASE is an ENTITY
BEHAVIOR
    BLOCK main
        ( sys_init, true,self<--myInit;, NULL )
        ( asoc, true, local clase : string; sender<--getClass( clase ); self<--intAsoc( clase );, NULL )
    END;
```

Esta clase implementa una entidad genérica que contiene todas las características que debe tener todas las entidades dentro de la hiperhistoria, implementa tres reglas, la primera que acepta el evento *sys_init*, esto se hace así para que las entidades ejecuten el evento *myInit* en lugar del *sys_init* y este no se propague a clases superiores.

El segundo evento implementa la determinación por parte del objeto de la clase del objeto que se quiere asociar a este cuando recibe el evento de asociación, y envía el evento *intAsoc* para que sea resuelto por la subclase que lo necesite.

```
CLASS OBJANIMABLE is an ENTIDADBASE
    INTERNAL ATTRIBUTES
        destino : INTEGER;
    INITIALIZATION
        destino := 0;
BEHAVIOR
    BLOCK main
        ( iMover( d : integer ), true,   TIMER<--setEvent( 'oMover', 1, true );
                    destino := d; ,NULL )
        ( iDetener, true,      TIMER<--killEvent( 'oMover' );, NULL )
        ( oMover, true,  INTERFACE<--mover( myContext, destino );, NULL )
    END;
```

Esta clase implementa un tipo de entidad que tiene autonomía, es decir, efectúa alguna actividad a lo largo del tiempo, tiene un atributo interno que puede ser usado, según la subclase, como dirección del movimiento, etc.

Las reglas que implementa son las siguientes, *iMover*, con la cual el objeto comienza su actividad, *iDetener*, con la cual cesa su actividad, y *oMover*, que es el evento que recibe del timer, al hacerlo, envía a la interface una notificación de esto.

```
CLASS PERSONA is an OBJANIMABLE
BEHAVIOR
    BLOCK main
        ( activar, true,   sender<--IActivar;, NULL )
        ( iCruzar, true,   sender<--cross; ,NULL )
        ( tomar, true, sender<--iTomar; , NULL )
        ( soltar, true, sender<--iSoltar; , NULL )
    END;
```

Esta clase implementa al personaje de la hiperhistoria, no tiene atributos y su comportamiento hace que ante cada evento recibido de la interface correspondiente a una acción a realizar, efectúe dicha acción con el objeto correspondiente, el cual viene informado en la variable *sender*.

```
CLASS LLAVE is an ENTIDADBASE
    CONNECTIONS
        output;
BEHAVIOR
    BLOCK main
        ( iActivar, true, output<==pulso;, NULL )
    END;
```

Esta es la implementación de una llave eléctrica, se define una conexión llamada *output*, a través de la cual se enviará el evento que la llave genere al objeto o canal que se conecte a ésta. El único comportamiento que tiene es que al recibir el evento *iActivar*, se envía a través de la conexión el evento pulso, todo objeto que se desee conectar a una llave y que esta deba controlar, debe responder a este evento.

Notar que el evento es enviado de manera asincrónica, con lo que se escalonará con otros eventos asincrónicos en la hiperhistoria.

```
CLASS LAMPARA is an ENTIDADBASE
    INTERNAL ATTRIBUTES
        encendida : BOOLEAN;
    INITIALIZATION
        encendida := false;
BEHAVIOR
    BLOCK main
        ( pulso, true,    encendida := not encendida;
                          INTERFACE<--encendida( myContext, encendida );
                          if encendida    then( myContext<--iluminar; )
                                          else( myContext<--oscurecer; ); ,NULL )
    END;
```

Aquí se Implementa una lámpara que puede ser conectada a la llave anteriormente descripta, tiene un atributo que indica si está encendida o no y una regla que cambia el estado de este atributo y notifica a la interface en caso de que esto pase, además notifica al contexto en el que está incluida que se ilumine u oscurezca según corresponda.

```
CLASS OBJTOMABLE is an  ENTIDADBASE
BEHAVIOR
    BLOCK main
        ( iTomar, true, self<--takeBy( sender ); , NULL )
        ( iSoltar, true, self<--leaveBy( sender );, NULL )
    END;
```

Esta es la implementación básica de cualquier objeto que puede ser transportado por el personaje y asociado a otro objeto, las dos reglas que tiene como comportamiento hacen que el objeto pueda ser tomado o soltado.

```
CLASS LLAVEPUERTA is an OBJTOMABLE
BEHAVIOR
    BLOCK main
    END;

CLASS VASO is an OBJTOMABLE
BEHAVIOR
    BLOCK main
    END;

CLASS PELOTA is an OBJTOMABLE
BEHAVIOR
    BLOCK main
    END;
```

Estas tres clases de acá arriba definen algunas de las entidades que compondrán la hiperhistoria, observar que no tienen atributos ni comportamiento, la única diferencia entre ellas es que son distintas clases y su asociación, por este motivo, con otro objeto tendrá diferentes consecuencias según de que clase se trate.

```
CLASS FUEGO is an ENTIDADBASE
    INTERNAL ATTRIBUTES
        enc : boolean;
        cont : integer;
    INITIALIZATION
        enc := false;
        cont := 0;
BEHAVIOR
    BLOCK main
        ( myInit, true, self<--encender;, NULL )
        ( intAsoc( c:string ), c='ENCENDEDOR', self<--encender; , NULL )
        ( intAsoc( c:string ), c='VASO', self<--aApagar; , NULL )
        ( encender, not enc,    enc:=true;
                                INTERFACE<--prendida(myContext, enc );
                                TIMER<--setEvent( 'Agrandar', 20, true );
                                cont := 0; , NULL )
        ( aApagar , enc,    enc:=false;
                            INTERFACE<--prendida(myContext,enc );
                            TIMER<--killEvent( 'Agrandar' ); , NULL )
         ( agrandar, enc,   if cont < 40 then (  INTERFACE<--agrandar( myContext, cont );
                                                 cont := cont + 1; )
                                         else (  INTERFACE<--incendiar( myContext );
                                                 TIMER<--killEvent( 'Agrandar' ); );  , NULL )
    END;
```

Esta es la clase que implementa al fuego fuera de control, tiene dos atributos, uno que indica si esta o no encendido y otro un contador que ira aumentando a medida que pasa el tiempo, cuando llegue a cierto límite, incendiará la casa.

El comportamiento que tiene es el siguiente, la primera regla acepta el evento de inicialización encendiendo el fuego cuando arranca la hiperhistoria, la segunda y tercera son las asociaciones con el vaso y el encendedor que hacen que el fuego se pueda apagar y volver a encender.

Las dos reglas que siguen implementan el encendido y apagado del fuego, en la primera de las dos, además de notificar a la interface, programa el timer para que comience a correr agrandando el fuego, en la segunda, también notifica a la interface pero además detiene el timer.

La última de las reglas se ejecuta cada vez que el timer envía el evento *agrandar* al objeto, si el contador es menor que el límite preestablecido, lo aumenta y notifica a la interface de ello, en caso que llegue al límite, se informa a la interface de ello y se detiene el timer.

```
CLASS RELOJ is an ENTIDADBASE
    INTERNAL ATTRIBUTES
        horas, minutos, segundos : integer;
    INITIALIZATION
        horas := 0;
        minutos := 0;
        segundos := 0;
BEHAVIOR
    BLOCK main
        ( myInit, true, TIMER<--setEvent( 'elTick', 6, true );, NULL )
        ( elTick, true,     segundos := segundos + 1;
                if ( segundos > 59 ) then ( segundos := 0; minutos := minutos + 1; );
                if ( minutos > 59 ) then ( minutos := 0; horas := horas + 1; );
                if ( horas > 23 ) then ( horas := 0; );
                INTERFACE<--tiempo( myContext, horas, minutos, segundos );, NULL )
        ( iActivar, true, horas := horas + 1;, NULL )
        ( setear( ph, pm, ps : integer ), true, horas := ph; minutos := pm; segundos := ps; , NULL )
     END;
```

Esta última clase implementa un reloj, tiene tres atributos para las horas, los minutos y los segundos y arranca cuando llega la inicialización del sistema, programando el timer para que le envíe el evento *tick* , luego, la implementación de este evento modifica los atributos como corresponde, avanzando un segundo por cada vez que lo reciba, e informa estos atributos a la interface.

El reloj responde al evento *iActivar* adelantando una hora, también responde al evento *setear* que lo pone en hora.

6.3.4 - Definición de los conjuntos de objetos.

Luego, el paso que sigue es la definición de los distintos conjuntos de objetos, los cuales serán los que luego se incluirán en los diferentes contextos según corresponda, veremos ahora estas definiciones.

```
SET loDelLivIng
    personaje : PERSONA;
    lape : PELOTA;
    elHogar : FUEGO;
END;

SET llaveLamp
    laLlave : LLAVE;
    elCable : CHANNEL;
    laLampara : LAMPARA;
END;

SET cCocina
    elReloj : RELOJ;
END;

SET cPieza
    elVaso : VASO;
END;

SET cPlaza
    laLlavePuerta LLAVEPUERTA;
END;
```

En estos conjuntos se especifican objetos los cuales luego serán incluidos dentro de los contextos que se definirán luego.

6.3.5 - Implementación de la estructura de navegación e inclusión de objetos.

Por último, para completar la implementación de la hiperhistoria se deben definir los contextos que se usarán y como se comunicarán para conformar la estructura de navegación y que objetos estarán incluidos en cada uno de los contextos.

Primero veremos como definimos las clases con las cuales se hará la implementación de la estructura de navegación, estas son las siguientes:

```
CLASS CONTBASE is a CONTEXT
BEHAVIOR
BLOCK main
    ( take, true, NULL, NULL )
    ( leave, true, NULL, NULL )
    ( oTaked(o:hObject), true, INTERFACE<--oTomado(self, o );, NULL )
    ( oLeaved(o:hObject), true, INTERFACE<--oSoltado(self, o );, NULL )
END;
```

Esta clase define el comportamiento básico que deben tener todos los ambientes dentro de la hiperhistoria, esta declaración contexto no tiene atributos, las dos primeras reglas que implementa hacen que el contexto no pueda ser tomado, es decir acepta estos eventos para que no siga la búsqueda pero no hace nada.

Los dos siguientes informan a la interface cuando un objeto es soltado o tomado dentro del contexto, este evento se envía automáticamente cuando un objeto es tomado o soltado y lo único que hacen es informar a la interface.

```
CLASS AMBIENTE is a CONTBASE
INTERNAL ATTRIBUTES
    iluminado : boolean;
INITIALIZATION
    iluminado := false;
BEHAVIOR
    BLOCK main
        ( EntityIn( Ink : hObject ), true, INTERFACE<--entraObj( self, Ink, sender );, NULL )
        ( EntityOut( Ink : hObject), true,  INTERFACE<--saleObj( self, Ink, sender );, NULL )
        ( iluminar, true,  iluminado := true;
                           INTERFACE<--clluminar( id ) ;
                           self<--olluminar;, NULL )
        ( oscurecer, true,    iluminado := false;
                              INTERFACE<--cOscurecer( id );
                              self<--oOscurecer;, NULL )
END;
```

Esta es la clase que implementa un ambiente en la hiperhistoria en el cual pueden habitar objetos, tiene un solo atributo que indica si el ambiente esta o no iluminado y el comportamiento que hace lo siguiente: en las dos primeras reglas se informa a la interface cuando un objeto entra o sale del contexto, las dos siguientes informan a la interface la modificación del estado de iluminado del contexto e informan a todos los objetos dentro del contexto de ese cambio enviándose a si mismo el evento *olluminar* u *oOscurecer* según corresponda, como el contexto no responde a estos eventos, los reenviará a todos los objetos que habiten en él.

```
CLASS CASASIMPLE is a CONTBASE
ENTRYS
    aLaCocina;
CONTEXTS
    Living,Cocina,Pieza IS AN AMBIENTE;
LINKS
    livcoc IS A PUERTA;
    livpie IS A PUERTALLAVE;
INITIALIZATION
    living LINKED WITH livpie, livcoc;
    cocina LINKED WITH livcoc, aLaCocina;
    pieza LINKED WITH livpie;

    living INCLUDES loDelLiving + llaveLamp;
    cocina INCLUDES cCocina + llaveLamp;
    pieza INCLUDES cPieza + llaveLamp;

    living.laLlave CONNECT output WITH living.elCable;
    living.laLampara CONNECT WITH living.elCable;

    cocina.laLlave CONNECT output WITH cocina.elCable;
    cocina.laLampara CONNECT WITH cocina.elCable;

    pieza.laLlave CONNECT output WITH pieza.elCable;
    pieza.laLampara CONNECT WITH pieza.elCable;
END;
```

Esta última declaración define una casa en la hiperhistoria, como el contexto casa no será un ambiente en el cual los objetos habiten, sino que lo harán en los contextos internos a ella, la casa es subclase de *CONTBASE* directamente.

Este contexto no tiene atributos ni comportamiento pero define primero, una entrada a través de la cual el personaje podrá entrar o salir de ella, luego los contextos internos a ella, los links con los que se comunicarán dichos contextos y por último la sección de inicialización, en la cual se conectan los contextos (cláusula *LINKED WITH*), luego se incluyen los objetos dentro de cada contexto (cláusula *INCLUDES*) y por último se realizan las conexiones entre objetos y canales (cláusula *CONNECT WITH*).

Ahora veremos como se integran todas las declaraciones precedentes con el objeto de completar la implementación de la hiperhistoria, esto se lleva a cabo definiendo el contexto más externo de la misma

```
CLASS MUNDO is a CONTBASE
CONTEXTS
    unaCasa IS AN CASASIMPLE;
    barrio, plaza IS AN AMBIENTE;
LINKS
    barcoc IS A PUERTA;
    barpla IS A LINKABIERTO;
INITIALIZATION
    unaCasa LINKED WITH barcoc;
    barrio LINKED WITH barcoc, barpla;
    plaza LINKED WITH barpla;

    plaza INCLUDES cPlaza;
END;
```

Esta implementación es similar a la casa vista anteriormente, en ella se definen contextos, links y se ligan los contextos e incluyen los objetos como en la declaración anterior. Notar que en esta declaración se incluyó a la casa como un contexto más. Este contexto, al ser el último declarado, no debe ser instanciado explícitamente sino que al arrancar la hiperhistoria tendrá una instancia implícita.

Todos los pasos descriptos anteriormente llevan a tener uno o varios fuentes en el lenguaje del modelo los cuales compilaremos con la herramienta descripta en el capítulo anterior, esta generara un ejecutable, el cual es el módulo del kernel de la hiperhistoria y el archivo de nombres el cual se deberá completar luego cuando se implemente la interface.

6.4 - Diseño de la interface.

6.4.1 - Descripción de la herramienta Toolbook.

Toolbook es una herramienta de autoría para realizar sistemas de multimedia, utilizaremos este herramienta por la facilidad que provee para generar interfaces de usuario, las cuales se pueden desarrollar de manera visual con las ventajas que ello implica.

6.4.1.1 - Libro, páginas y background.

La metáfora utilizada para la construcción de aplicaciones en esta herramienta es el libro, este libro en sí es un objeto y estará compuesto por páginas, cada página es como un contenedor de objetos. Existe también el concepto de background, que es otro objeto toolbook que se sitúa por detrás de las páginas, puede haber desde un sólo background compartido por todas las páginas en el libro hasta uno por cada página del libro.

El background también puede contener objetos, generalmente se usa este para colocar en el todos los dibujos que cada página en un libro, si varias páginas tienen el mismo fondo, entonces esas páginas pueden compartir el background, con lo cual este dibujo está una sola vez.

6.4.1.2 - Objetos y grupos.

Además del libro, las páginas y los backgrounds, en toolbook están los objetos, que pueden ser ubicados en las páginas o en los backgrounds para dibujar la interface. Estos objetos pueden ser botones, campos de texto, imágenes, líneas, etc.

Los grupos se forman asociando dos o mas de los elementos mencionados anteriormente.

6.4.1.3 - Propiedades y scripts.

Cada uno de los objetos mencionados anteriormente (incluyendo al libro, las páginas y los backgrounds) tienen asociados propiedades, algunas de las cuales son predefinidas, como pueden ser el nombre, el numero de página (este solo para las páginas), la posición dentro de una página (solo para los objetos que se pueden incluir en una página o background), si el objeto es visible o no, etc., y también el que desarrolle el sistema puede definir propiedades para los objetos que crea necesario hacerlo.

Una propiedad especial que tienen todos los objetos es el script, el cual es una secuencia de handlers a los cuales responde el objeto que lo implementa, en cada uno de estos handlers se programa un código que se ejecutará cuando se invoque ese handler en ese objeto. Cada handler puede recibir parámetros y puede retornar algún valor.

6.4.1.4 - Jerarquía para el manejo de handlers.

Existe una jerarquía de objetos en toolbook para que si uno de ellos no puede responder a un handler porque no lo tiene implementado, se continúe buscando el handler que responda en dicha jerarquía, antes de que se produzca un error.

La mencionada jerarquía es la siguiente, un evento dirigido a un objeto lo recibe este y si en su script no hay un handler que responda a este evento, es la página la que recibe el evento y luego el background y por último el libro.

6.4.2 - Representación de los contextos en la interface.

La forma en que se implementará la interface para la estructura de navegación es la siguiente : cada contexto definido en la hiperhistoria será una página toolbook en la interface, esta manera de definirlo la adoptamos en este ejemplo, pero no tiene por

que ser necesariamente así, por ejemplo, si implementamos un mapa en el cual se ve la ubicación de todos los personajes de la hiperhistoria, se verán todos los contextos en una sola pagina (suponiendo que quepan), y los contextos estarían representados por objetos (cuadrados, etc.).

Por la manera que tiene toolbook de nombrar a los objetos, es necesario, al definir así la estructura de navegación, llevar de alguna manera la ubicación de cada objeto, es decir, llevar en que pagina esta cada uno en cada momento, para que cuando se refieran a este, pueda identificárselo de manera unívoca.

6.4.3 - Forma en que representaremos las entidades y personajes.

Por la forma de referirse que tiene toolbook con los objetos, que se debe indicar el tipo de objeto que es además del nombre, como por ejemplo, picture imagen, rectangle uno ; implementaremos todos los objetos de interface como grupos, y estos serán los receptores finales de los eventos que el kernel envíe.

Como se vio anteriormente los grupos también tienen un script y en el se deben implementar las acciones de interface que se deben ejecutar por cada evento que recibirá cada objeto por parte del kernel.

6.5 - Implementación de la Interface.

Ahora veremos cómo será la implementación de la interface, como se implementará la comunicación con el kernel y como se manejarán los eventos recibidos por el kernel para que los reciba el objeto correspondiente de la interface.

También veremos como se deben implementar los grupos de la interface que responderán a dichos eventos y como se deben capturar las acciones de usuario y generar eventos para el kernel.

6.5.1 - Libro de sistema para la implementación de rutinas.

Existe un conjunto de handlers que serán fijos no importa de que hiperhistoria se trate, son aquellos que implementarán la comunicación con el kernel, los que ejecutarán los pasos, que ya describimos anteriormente, para conectarse con el kernel, para recibir un evento que este envíe y reenviárselo a su vez al objeto correspondiente y para enviar un evento hacia el kernel.

Todas estas acciones serán implementadas en un libro del sistema de toolbook el cual deberá usar cada una de las interfaces que se implementen, este contendrá en el script del libro los handlers para llevar a cabo las acciones descriptas anteriormente.

6.5.2 - Rutina de inicialización de la interface.

Este handler se debe ser invocado por la interface cada vez que esta arranque, en el se implementa la conexión con el kernel usando las funciones implementadas en la librería común, descripta anteriormente en el Capitulo IV.

En este ejemplo se implemento también otra DLL usada para que toolbook determine dado un nombre, la identificación interna del objeto y dada la identificación interna de un objeto el nombre de este para toolbook. Esta DLL lee el archivo de nombres mencionado en el Capitulo IV y llena unas estructuras internas a la DLL con estos datos, los cuales permanecen accesibles a toolbook el resto del tiempo que dure la ejecución de la interface.

Esta DLL posee las siguientes rutinas que debe invocar la interface que la use :

function buildTable(word) : word ; export ;

Esta función carga la tabla de nombres en las estructuras internas de la DLL, el parámetro que recibe es el handler al área de memoria donde se encuentra el nombre del archivo, en forma de un string terminado en el carácter nulo.

Esta función retorna uno si la operación fue exitosa y cero en caso contrario.

function lenTable() : word ; export ;

Esta función retorna la cantidad de elementos que tiene la tabla que se arma con la función previa, si esta no fue armada o hubo un error en el armado, retorna cero.

function hayNameAt(word) : word ; export ;

Determina si, dada una identificación interna de un objeto de la hiperhistoria, hay un nombre de interface asociado a ella. Si es así esta función retorna uno, en caso contrario retorna cero.

function nameAt(word) : word ; export ;

Dada una identificación interna, esta función retorna el nombre de interface que corresponda, en realidad, retorna el handler al área de memoria donde se encuentra este nombre en forma de un string terminado en carácter cero.

function idOf(word) : word ; export ;

Esta función determina, dado el nombre de interface del objeto, la identificación interna de este, el parámetro que se pasa debe ser el handler de un área de memoria donde estará el nombre como un string terminado en cero. Esta función retorna la identificación interna del objeto en caso que este exista, en caso contrario retorna FFFF hex.

Ahora veremos la implementación del handler utilizado para la inicialización de la interface y su conexión con el Kernel :

```
to handle initialize aConfigFile
    system _a_Pascal, _a_Message        Handler del kernel y mensaje común que se usará para la comunicación.

    linkDLL "NCOMMON.DLL"               Se liga con la librería común para usar sus funciones.
        word initlib()                  Inicializa la librería y determina el mensaje común.
        word registrarTool( word )      Función para registrar la interface en la librería
        word pasHandle()                Determina el handler de la aplicación ( Kernel ).
        word tPostMessage( word, word, word, dword )    Implementa la función postMessage de Windows.
    end linkDLL

    linkDLL kernel                      Librería de Windows para el manejo de la memoria.
        word globalAlloc( word, dword )
        pointer globalLock( word )
        word globalUnlock( word )
        word globalFree( word )
    end linkDLL

    linkDLL "FORINTER.DLL"              Librería especifica para la interface toolbook. Manejo de nombres
        word buildTable( word )
        word lenTable()
        word hayNameAt( word )
        word nameAt( word )
        word idOf( word )
    end linkDLL

    linkDLL "TB30DOS.DLL"               Función para determinar si un archivo existe.
        int fileExists( string )
    end linkDLL

    if aConfigFile=NULL then            Si no se especifica un archivo de configuración usar config.pas
        set aConfigFile to "config.pas"
    end
```

```
if fileExists( aConfigFile ) = 1          Si existe el archivo de configuración, seguir con la inicialización.

    get putName( aConfigFile )            Guarda el nombre en un área de memoria y obtiene su handler.

    get buildTable( it )                  Arma la tabla de nombres interna con el archivo que se indica.
    if it = 0 then                        Si no pudo armar la tabla entonces cancela la inicialización.
        send errorEnLaTabla
    end

    set _a_Message to initLib()           Inicializa NCOMMON.DLL y obtiene el mensaje de Windows común.
    get registrarTool( SYSWINDOWHANDLE )  Registra en NCOMMON.DLL el handler de la instancia de la
                                          interface que esta ejecutándose.
    if it<>0                              Si ya habla una interface activa ( registrarTool retornó 1 ).
        send iniciaConn to this book      Comienza el contacto con la interface.
    else                                  En caso contrario.
        translateWindowMessage            Hace que toolbook cuando reciba el mensaje común envíe el evento
                                          IniciaConn al libro ( espera la activación del kernel )
            before (_a_Message) send iniciaConn to this book
        end
    end if
else                                      Si no encuentra el archivo de nombres cancela la inicialización.
    send fileNotFound aConfigFile to this book
end if
end initialize
```

La función *putName* que aparece en el código precedente lo que hace es reservar un área de memoria y almacenar el string que se pasa como parámetro en ella, retornando el handler a dicha área.

El siguiente handler se ejecutará una vez que el kernel este activo, para redireccionar los mensajes de Windows para que cuando toolbook los reciba invoque el handler *msgPascal*.

```
to handle iniciaConn hwnd, winmsg, wp, lplo, lphi
    system _a_Pascal, _a_Message          Handler del kernel y mensaje común que se usará para la comunicación.
    set _a_Pascal to pasHandle()          Obtiene el handler del kernel.
    unTranslateWindowMessage (_a_Message) Desactiva el reenvío anterior del mensaje común de Windows.
    translateWindowMessage                Hace que toolbook cuando reciba el mensaje común envíe el evento
                                          msgPascal al libro ( espera evento del Kernel )
        before (_a_Message) send msgPascal to this book
    end
end iniciaConn
```

Luego, a partir de que este último handler se ejecute, la comunicación con la interface queda establecida, a partir de este momento el Kernel esta funcionando y envía eventos hacia la interface y esta puede hacer lo propio según las acciones del usuario.

6.5.3 - Manejador para recibir un evento desde el kernel.

El siguiente handler que veremos es el que se invoca cuando toolbook recibe el mensaje Windows común a las interfaces y el kernel, en este handler lo que se hace, básicamente, es extraer del área de memoria, cuyo handler se recibe como parámetro, los datos del evento y hacer que se ejecute el correspondiente handler en el objeto de interface indicado.

```
to handle msgPascal hwnd, winmsg, wp, wplo, wphi
    local nObj, pObj, msgAttr, auxobj

    set addrData to globalLock( wp )                    Obtiene la dirección de los datos que envió el Kernel.
    set msgOid to pointerWORD( 0, addrData )            Obtiene la identificación interna del objeto destino del evento.
    set msgAttr to pointerSTRING( 2, addrData )         Obtiene el nombre del evento enviado.
    set desp to ( charCount( msgAttr ) + 3 )            Apunta a la posición de memoria que sigue al nombre.
    set pCount to pointerWORD( desp, addrData )         Obtiene la cantidad de parámetros que trae el evento.
    increment desp by 2                                 Apunta al primer parámetro
    set p1 to null          Inicializa variables para almacenar los parámetros que se recibirán.
    set p2 to null
    set p3 to null
    set p4 to null
    set pasig to 1          Inicializa un Indice para referirse a los parámetros.
    while pCount>0          Mientras haya parámetros.
        set btipo to pointerBYTE( desp, addrData )      Obtiene el tipo del parámetro.
        set tipo to ansiToChar( btipo )                 Lo interpreta como carácter.
        increment desp                                  Apunta al valor del parámetro corriente.
        conditions          Según el tipo del parámetro obtiene la cantidad de bytes correspondientes a su valor.
            when tipo = "B"
                set elParam to pointerBYTE( desp, addrData )    Lee el valor del parámetro.
                set nextDesp to 1                               Asigna el desplazamiento del próximo parámetro.
            when tipo = "W"
                set elParam to pointerWORD( desp, addrData )
                set nextDesp to 2
            when tipo = "I"
                set elParam to pointerINT( desp, addrData )
                set nextDesp to 2
            when tipo = "L"
                set elParam to pointerLONG( desp, addrData )
                set nextDesp to 4
            when tipo = "S"
                set elParam to pointerSTRING( desp, addrData )
                set nextDesp to ( charCount( elParam ) + 1 )
            when tipo = "T"
                set elParam to ( pointerWORD( desp, addrData ) = 1 )
                set nextDesp to 2
            when tipo = "O"
                set auxObj to pointerWORD( desp, addrData )
                set nextDesp to 2
                get hayNameAt( auxObj )
                if it<>0            En caso de ser un objeto, obtiene el nombre de interface de este.
                    set hObj to nameAt( auxObj )
                    set pObj to globalLock( hObj )
                    set elParam to pointerSTRING( 0, pObj )
                    get globalUnlock( hObj )
                    get globalFree( hObj )
                end if
        end conditions
```

```
        conditions              Asigna el valor del parámetro a la variable correspondiente.
            when pasig=1
                set p1 to elParam
            when pasig=2
                set p2 to elParam
            when pasig=3
                set p3 to elParam
            when pasig=4
                set p4 to elParam
        end conditions

        increment pasig                 Incrementa el Indice para referirse al siguiente parámetro.
        increment desp by nextDesp      Apunta a la posición siguiente al parámetro que fue leído.
        decrement pCount                Decrementa la cantidad de parámetros leídos.
    end while

    get globalUnlock( wp )          Libera la memoria usada en el área de intercambio con la inteface.
    get globalFree( wp )

    get hayNameAt( msgOid )         Determina si el destino tiene un nombre de interface.
    if it<>0                        Y si es así...
        set hObj to nameAt( msgOid )            Obtiene el nombre del objeto destinatario del evento.
        set pObj to globalLock( hObj )
        set nObj to pointerSTRING( 0, pObj )
        get globalUnlock( hObj )
        get globalFree( hObj )

        En las dos sentencias siguientes se envía el evento al objeto destino con sus correspondientes parámetros.

        put "send"&&msgAttr&&"p2, p3, p4 to group"&&nObj&&"of page p1" into auxi
        execute auxi
    end
end msgPasc
```

En esta implementación, por las razones expuestas mas arriba acerca del nombrado de objetos en toolbook, se interpreta el primer parámetro que llega del kernel como el contexto en el cual esta el objeto que recibe el evento, de esta manera, como en la implementación de esta interface se hace que cada contexto en el kernel corresponda a una página toolbook, es posible identificar unívocamente el objeto de interface receptor del evento.

Esta forma de implementarlo hace que, en el kernel, se deba escribir código de mas para que, cada vez que se envíe el un evento a la interface, se mande como primer parámetro el contexto que contiene al objeto del kernel. Además, no es posible utilizar los atributos comunes, ya que éstos envían un evento implícito a la interface, el cual no lleva como primer parámetro el contexto en el cual está el objeto que genera el evento.

Se debe notar que los objetos de Toolbook receptores de los eventos serán siempre *grupos*, de esta manera, es sencillo referirse a los objetos que recibirán eventos del kernel. Si no se hiciese de esta manera, habría que llevar el control para determinar, dado un nombre de objeto Toolbook, si este es un dibujo, un botón, un polígono, etc.

6.5.4 - Manejador para enviar un evento hacia el kernel.

Al siguiente handler hay que invocarlo cuando se quiere enviar un evento hacia alguno de los objetos de la hiperhistoria en el kernel, los parámetros que hay que especificarle son el nombre de interface del destino, el nombre del evento y, si existen, el nombre de otro objeto involucrado en el evento y los tipos y valores de los parámetros.

```
to handle msgInter nomDe, evento, nomOt, t1,p1, t2,p2, t3,p3, t4,p4
    system _a_Pascal, _a_Message        Handler del kernel y mensaje común que se usará para la comunicación.
    local idDe, idOt                    Variables para almacenar la identificación del destino y del otro objeto.
    local canParam                      Cantidad de parámetros.

    set canParam to argCount            Determina el número de parámetros recibidos.
    if canParam < 3 then
        set canParam to 3
    end if

    get putName( nomDe )                Coloca el nombre del destino en un área de memoria.
    set idDe to idOf( it )              Determina el identificador interno del objeto destino del evento.
    if idDe<>65535 then                 Si el nombre del destino esta en la tabla.
        get msgLong( evento, t1,p2,t2,p2,t3,p3,t4,p4 )   Obtiene la longitud total de la información a pasar al Kernel..
        set hMemory to globalAlloc( 8192, it )           Aloca un área de memoria para almacenar estos datos.
        set addrData to globalLock( hMemory )
        get pointerWORD( 0, addrData, idDe )             Escribe en dicha área la identificación interna del destino.
        get pointerSTRING( 2, addrData, evento )         Escribe el nombre del evento.
        set ndesp to ( 2+charCount( evento ) )
        get pointerBYTE( ndesp, addrData, 0 )            Escribe el carácter cero para terminar el nombre del evento..
        increment ndesp by 1
        if nomOt<>null then                              Si existe otro objeto involucrado.
            get putName( nomOt )
            set idOt to idOf( it )                       Obtiene la identificación interna de este.
            if idOt = 65535 then
                set idOt to idDe                         En caso de ser inválida la identificación del destino, se pone en
            end if                                       su lugar la identificación del origen.
        else
            set idOt to idDe
        end if
        get pointerWORD( nDesp, addrData, idOt )         Escribe la identificación del otro objeto en el área de memoria.
        set cParam to ( ( canParam - 3 ) / 2 )           Obtiene la cantidad de parámetros que se enviarán.
        get pointerWORD( nDesp+2, addrData, cParam )     Escribe la cantidad de parámetros que se enviarán.
        get pasaParam( addrData, cParam, nDesp+4, t1,p1,t2,p2,t3,p3,t4,p4 )   Escribe los parámetros.
        get tPostMessage( _a_Pascal, _a_Message, hMemory, 0 )   Envía el mensaje a la interface.
        get globalUnlock( hMemory )      Deslockea el área de memoria pero no la libera, esto lo hace el kernel.
    end if
end msgInter
```

En este manejador se almacena en el área de memoria común los datos del evento que se quiere enviar y se genera un mensaje de Windows, cuyo código es el común que se acordó entre el kernel y las interfaces, y se envía como parámetro word de este mensaje el handler del área de memoria usada.

Todos los handlers precedentes que vimos son los mismos para la implementación de cualquier interface en Toolbook en la cual cada contexto corresponda a una página, siempre y cuando también se mantenga la forma de programar el Kernel, por lo tanto, en el ejemplo, se incluyeron todos estos handlers en un libro de sistema de Toolbook, el cual debe ser utilizado por cada interface que se implemente.

6.5.5 - Implementación de los contextos, links y entidades en la interface.

Cada contexto, como se dijo, será representado, en este ejemplo, por una página de Toolbook, como para el kernel los contextos son objetos y son considerados, en lo que hace al aspecto de la comunicación, como cualquier otro objeto de la hiperhistoria, los eventos que envíe el kernel hacia la interface, dirigidos a los contextos serán similares al resto de los eventos enviados, así que, debido a la forma que tiene Toolbook de nombrar objetos, las páginas en si nunca recibirán estos eventos, por lo tanto en cada página se debe incluir un grupo, el cual no sea otro objeto de la hiperhistoria, el cual represente a la página y que recibirá los eventos dirigidos a esta desde el kernel.

Cada vez que el personaje central, esto es, el protagonista de la historia, aquel que el lector sigue, cambie de contexto en la representación de la estructura de navegación implementada en el kernel, se originarán mensajes hacia la los contextos involucrados, esto es, hacia el contexto del cual sale el objeto y aquel al cual llega, y estos a su vez enviarán sendos eventos a la interface informando acerca del objeto que cambia de contexto, por lo cual, ante esto, se debe realizar un cambio de página en la interface, borrar el dibujo del personaje del contexto viejo y dibujar este en el contexto nuevo.

Veamos como ejemplo los handlers que se deben implementar para la representación en la interface del contexto *LIVING*, solo pondremos los encabezados y una explicación de lo que deberían hacer:

```
to handle entraObj lugar, objeto
    ...
end entraObj
```

Este handler recibe como primer parametro el nombre de interface del link por el cual entra el objeto al contexto y el nombre de interface del objeto que entra, este handler debe mostrar, en la posición que corresponda según la ubicación del link, el objeto que entra al contexto (el objeto será un personaje u objeto animado el cual pueda trasladarse por si mismo o a petición del lector)

```
to handle saleObj lugar, objeto
    ...
end saleObj
```

En este caso lo que hara la implementación de este handler es ocultar el objeto que salió del contexto que recibe el evento, de modo que, para el lector, aparezca como que no está más en ese contexto. Por razones de implementación, en cada contexto hay una copia de la representación visual del personaje, quien es el único en este ejemplo que puede trasladarse por si mismo, los restantes objetos lo haran transportados por este.

Por último, los contextos deben responder a los eventos *cIluminar* y *cOscurecer* generando un cambio en la visualización del contexto de manera acorde a la acción que se quiere representar, y también a los eventos *oTomado y oSoltado* haciendo desaparecer y aparecer el objeto tomado o soltado en la interface.

Los link y las entidades también serán representados por grupos en la interface y estos deberán tambien responder a mensajes desde el kernel, los mensajes que tipicamente deberán responder los links para este ejemplo son *meAbren* y *meCierran* para que cambie el aspecto según el link este abierto o no (los links que son puertas, ya que aquellos que son pasajes no harán ningún cambio en su aspecto)

Las entidades que reciben mensajes son aquellas que cambian el aspecto con el paso del tiempo o con alguna acción del usuario, en el ejemplo, la entidad *fuego* recibirá el mensaje *agrandar* a medida que pase el tiempo, con lo que en la interface su tamaño deberá ir creciendo, y de la misma manera, cada entidad que reciba eventos de esta naturaleza debera implementar handlers de toolbook que hagan que su aspecto varíe según corresponda.

Por otra parte, ante las acciones del usuario, la interface debe detectar sobre que objetos se realiza la acción y informar de esto, el tipo de acción y los objetos involucrados, al kernel de la hiperhistoria..

6.5.6 - Aspecto de la interface y mecanismos de interacción.

Ahora vamos a explicar como el lector debe usar la interface para realizar las acciones necesarias para la hiperhistoria, toda esta interacción se realiza utilizando sólo el mouse; las acciones básicas que se pueden ejecutar en la interface son las siguentes:

- Activar : sirve para encender luces, abrir puertas, etc. o sea, actuar sobre un objeto directamente, para llevarla a cabo hacer drag desde el cuerpo del personaje hacia el objeto a activar.

- Tomar : para tomar los objetos con la finalidad de llevarlos de un lado a otro o usarlos para hacer algo. Para tomar, hacer un drag desde el objeto tomar hacia el cuerpo del personaje.

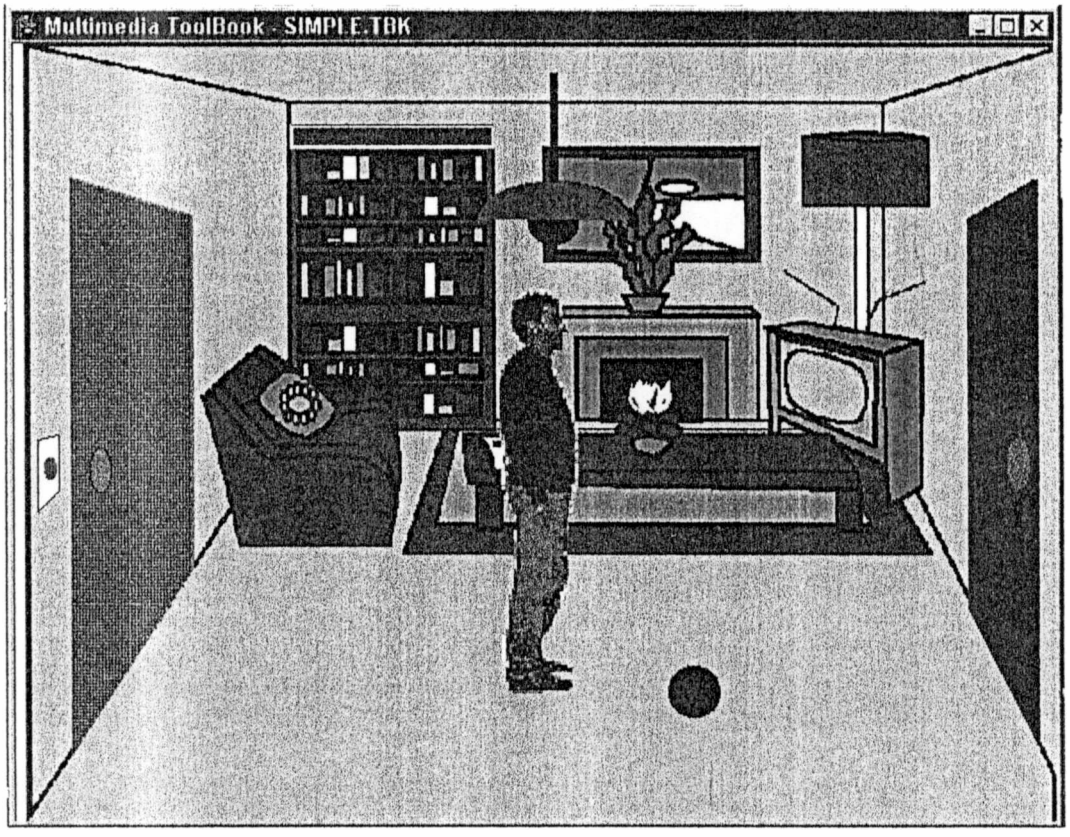
- Soltar : para dejar el objeto en el lugar en que esta el personaje, para hacer esto, hacer doble click sobre el cuerpo del personaje (torso), con lo cual aparecerá la bolsa del personaje con los objetos que en ese momento tiene tomados, luego hacer drag con el objeto a soltar (arrastrarlo hacia el fondo de la bolsa) (hasta que aparezca la mano sin el signo de prohibido) y luego hacer click en algún lugar del contexto que no haya ningún objeto. El objeto será soltado a los pies del personaje.

- Asociar : para usar un objeto en combinación con otro para hacer algo. Hacer igual que al soltar pero en lugar de clickear, en el final, en cualquier lugar, hacerlo sobre el objeto sobre el cual se quiere actuar. Por ejemplo, para destrabar la puerta del dormitorio se debe primero tomar la llave; luego seguir los pasos como para soltarla, pero al final clickear sobre la puerta.

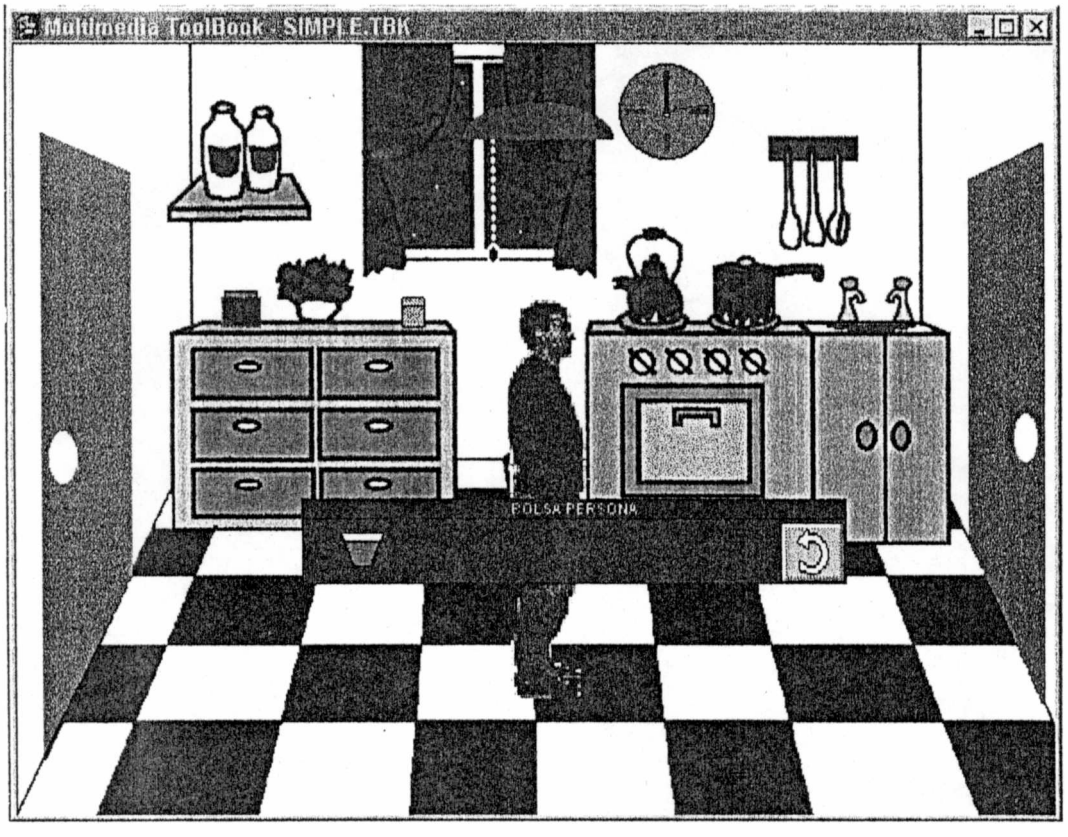
- Para caminar simplemente clickear hacia el lugar donde se quiere ir y para cruzar de un contexto a otro clickear sobre las puertas cuando estas estén abiertas o sobre las flechas, en caso de querer pasar hacia la plaza o desde la plaza, ya que ahí no hay una puerta, este link esta siempre abierto.

Por último, veremos como se ve la interface, cada pantalla que se ve es un contexto de la hiperhistoria, el personaje esta representado por la foto de la persona, y esta puede desplazarse a requerimiento del letor hacia uno u otro lado, ya que para actuar sobre algún objeto o link en el ambiente en que se encuentre, deberá estar próximo a este.

imer pantalla que vemos es el living, que es la pantalla inicial de la hiperhistoria:



pantalla se ve la bolsa del personaje, la cual contiene los objetos que el fue tomando:



CAPITULO VII.

TRABAJOS FUTUROS.

7.1 - Introducción.

Hasta aquí vimos la generación de hiperhistorias solo a través del editor hecho para tal fin, y generando la interface sin ayuda alguna. Este trabajo es bastante complejo en el caso de querer desarrollar una aplicación de tamaño considerable, por lo tanto sería deseable tener una manera de generar en forma automática parte o todo el código de la aplicación y su interface o una interface para una hiperhistoria ya realizada. En este capítulo veremos cuales serían los requerimientos que tales herramientas tendrían.

Por otra parte, el uso de hiperhistorias en forma cooperativa, es decir, en un ambiente donde varios usuarios o lectores puedan acceder a la misma hiperhistoria y que las acciones de uno de ellos sobre la historia influya sobre los otros es un campo interesante para su estudio, presentaremos algunas ideas de los módulos que se deberían desarrollar para utilizar las hiperhistorias generadas para este fin.

7.2 - Requerimientos para un generador de hiperhistorias.

Una aplicación destinada al desarrollo de hiperhistorias debería proveer un ambiente en el cual el diseño e implementación de la misma se realice en forma mas intuitiva, debería poseer una interface gráfica en la cual puedan representarse los contextos, objetos y personajes.

Seria deseable que la aplicación para desarrollar la hiperhistoria permita hacer la interface de manera directa, es decir, que pueda verse en el momento de la implementación como se vera esta.

Esta aplicación debe permitir en una primera instancia diseñar la estructura de navegación, para lo cual una interface que permita dibujar un grafo seria suficiente, en esta se pueden definir cuales serán los contextos, cuales los links, y definir sus propiedades.

Luego se debe tener alguna manera de definir de manera gráfica que objetos contendrá cada contexto y definir sus propiedades.

Como la definición de las hiperhistorias se hace a partir de clases y no de instancias de los objetos, la aplicación deberá proveer alguna forma de explorador de clases el cual permita su definición, luego, gráficamente, se instanciarían los objetos ya sobre la representación final de la interface, pero siempre asignándole a cada uno de ellos una clase a la cual pertenecen.

Basándose en esta definición gráfica la herramienta de desarrollo debe ser capaz de generar los archivos de definición del kernel y la estructura e implementación de la interface en el grado mas avanzado posible.

Seria deseable que también el generador permita la definición de una interface a partir de una hiperhistoria, o sea un kernel, ya creado, con el objeto de generar nuevas interfaces con métodos de interacción diferentes, siendo todas estas representación de la misma hiperhistoria pudiendo de esta manera ejecutarse de manera conjunta.

7.3 - Ejecución de hiperhistorias de manera cooperativa.

Al tener la posibilidad de desarrollar interfaces disímiles para una misma hiperhistoria, las cuales pueden poseer diferentes estilos de interacción, de modo que se adecúen a diferentes necesidades especiales que los lectores puedan llegar a necesitar, como ser lectores ciegos, con problemas de aprendizaje, etc. seria interesante que no solo interactúen con la hiperhistoria en si utilizando una interface adecuada, sino que sería posible que varios lectores interactúen entre ellos a través de la hiperhistoria, por ejemplo, teniendo el control cada uno de ellos de un personaje con el cual realizan acciones que afectan a los demás lectores.

Cada uno de los lectores interactuará con la hiperhistoria utilizando la interface mas adecuada a su necesidad, pero las acciones que realice que afecten a otros lectores harán que las interfaces de cada uno de ellos vean reflejadas estas acciones de un modo accesible a su necesidad, por lo tanto, la interacción se haría mas confortable para cada uno de ellos.

Para llevar a cabo esto es necesario poder ejecutar las hiperhistorias en un ambiente de red en el cual la hiperhistoria se ubicará en una máquina que hará las veces de un server y luego cada lector podrá usar la hiperhistoria a través de interfaces diferentes en máquinas distintas.

La implementación de las hiperhistorias tal cual como esta permitiría esta característica, pero sería necesario desarrollar módulos especiales para colocar en cada máquina, es decir, en la máquina donde se ejecute el kernel habría un módulo que el kernel vería como una interface el cual al recibir un evento de esta lo enviaría a través de la red y en caso contrario, al recibir por la red un mensaje hacia el kernel lo transmitiría a este.

En el caso de las terminales o máquinas donde se ejecutarán las interfaces deberá haber un módulo del mismo estilo del anterior pero que para la interface hará las veces de kernel, con lo cual pasará a la interface los mensajes que le lleguen a través de la red y enviará, a través de esta, hacia la interface los eventos correspondientes.

Bibliografía

Bates J., Abbe D., Strickland R. : Interface and narrative arts: contribution from narrative, drama and film.

Bork A., Learning in the twenty-first century Interactive multimedia technology, Proceedings of the NATO Advanced Research Workshop on Interactive Multimedia Learning Environments, Quebec, June 17-20, 1991, pages 2 to 18.

Giardina M., Interactivity and intelligent advisory strategies in a multimedia learning environment: human factors, desing issues and technical considerations, Proceedings of the NATO Advanced Research Workshop on Interactive Multimedia Learning Environments, Quebec, June 17-20, 1991, pages 48 to 66.

Goetzfried, L. And Hannafin, M.J.; The effects of embedded CAI instructional control strategies on the learning and applicactions of mathematics rules. American Educational Research Journal, Vol. 22, pages. 273 to 278 (1985).

Granieri, J.P., Becket, W., Reich, B.D., Crabtree, J., Badler, N.I. : Behavioral control for real-time simulated human agents, Symposium on Interactive 3D Graphics, Monterey CA USA, 1995

Gronbaek, K., Hem, J.E., Madsen, O.L., Sloth, L. : Hipermedia systems: a Dexter-based architecture, Communications of the ACM, February 1994, Vol. 37, No. 2, pages 64 to 74.

Gronbaek, K., Trigg, R.H. : Issues for a Dexter-based hipermedia system, Communications of the ACM, February 1994, Vol. 37, No. 2, pages 40 to 49.

Halasz F., Schwartz M. : The Dexter hipertext reference model, Communications of the ACM, February 1994, Vol. 37, No. 2, pages 31 to 39.

Hardman, L., Bulterman, D.C.A., Van Rossum, G. : The amsterdam hipermedia model: adding time and context to the Dexter model, Communications of the ACM, February 1994, Vol. 37, No. 2, pages 50 to 62.

Haugen H., Multimedia learning environment : an educational challenge, Proceedings of the NATO Advanced Research Workshop on Interactive Multimedia Learning Environments, Quebec, June 17-20, 1991, pages 39 to 45.

Lange, D. : An object-oriented design method for hypermedia information systems, presented at the Twenty-seventh Annual Hawaii International Conference on System Sciences in Hawaii, January 1994.

Lange, D. : Developing hipermedia information systems: an object-oriented approach, Submitted for the Journal of Organizational Computing.

Leggett, J.J., Schnase, J.L. : Dexter with open eyes, Communications of the ACM, February 1994, Vol. 37, No. 2, pages 76 to 86.

Lewis, J.B., Koved, L., Ling, D.T. : Dialogue structures for virtual worlds, ACM Communications, 1991.

Ross, S.M., et al. Matching the lesson to the student: Alternative adaptive design for individualized learning systems. Journal of Computer-Based Instruction, Vol. 32, No. 1, pages 41 to 49 (1984)

Sánchez, J. Mallegas, A., Cernuzzi, L., Rossi, G., Lumbreras, M., Diaz, A., Bibbó, L.M. Valdeni de Lima, J. : A conceptual framework for building hiperhistories.

Sánchez, J., Cernuzzi, L. : Guidelines para la contrucción de hiperhistorias que ayudan a resolver problemas de relaciones espacio-temporales y lateralidad.

Snowdon, D.N., West, A.J., Howard T.L.J. : Towards the next generation of human-computer interface, Presented at "Informatique '93 : Interface to Real & Virtual Worlds", pages 399 to 408, 24-26th March 1993, Montpellier, France.

Tennyson R.D. and Bultrey, T.: Advisement and management strategies as design variables in computed assisted instruction. Educational Communications and Technology Journal, Vol. 28, pages 169 to 176 (1980)