UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PROTOTIPO DE UN AMBIENTE DE DESARROLLO PARA ANIMACIONES EN JAVA PARA CURSOS ENVASADOS

JENS HARDINGS PERL

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

SANTIAGO DE CHILE
ENERO DE 2000

Resumen

El presente trabajo tuvo como objetivo el diseño de una API para crear y manipular animaciones en *Java*, y la implementación de un prototipo para su validación. Este se enmarca dentro del proyecto global de una herramienta para cursos envasados, la cual se ha dividido en componentes. El proyecto pretende apoyar la docencia, haciendo uso de todos los sentidos del alumno para permitir la comprensión de conceptos, ideas e información. Es posible usar el prototipo construido para validar el diseño y permitir mejoras.

El diseño se realizó siguiendo un patrón de diseño aplicado a la programación orientada a objetos, propia del lenguaje *Java*. Este patrón de diseño, junto con bibliotecas de trabajos anteriores, presentan una abstracción que permite la programación de un sistema conceptualmente simple, pero al mismo tiempo poderoso, para crear animaciones.

La biblioteca creada constituye uno de los componentes de la herramienta para cursos envasados y sienta una base para continuar el desarrollo de otros componentes.

Su principal característica es presentar animaciones simples y complejas con la misma interfaz, permitiendo el uso de animaciones arbitrariamente complejas de igual manera que las simples. La ventaja de un diseño de este estilo es que se pueden crear bibliotecas de animaciones, las cuales se pueden componer arbitrariamente en animaciones más complejas, pero la reproducción de cualquier animación continúa siendo de la misma simpleza.

Si bien se cuenta con un prototipo que funciona correctamente, se detectaron aspectos que deben aún ser abordados para mejorar la robustez de la implementación de la herramienta. Se sugiere la creación de una capa adicional para manejar las diferencias entre dos mecanismos de programación usados.

En base al diseño realizado, es posible crear un marco de referencia por el cual debieran regirse los trabajos relacionados con los componentes faltantes para completar la herramienta para cursos envasados, en particular el editor gráfico que unirá todos los componentes.

Agradezco a mis padres, por su apoyo incondicional durante toda mi vida. Y a Carolina por acompañar e incentivarme en estos importantes años.

Índice General

Capítulo 1

Introducción

Con el aumento de las capacidades multimediales y de interconexión de los computadores personales, así como su masificación, se ha vuelto una posibilidad real la utilización de material de apoyo electrónico para complementar la docencia. Así, este material puede ser publicado en un servidor para ser visto a cualquier hora y en cualquier lugar por los alumnos de un curso, o por cualquier otro individuo interesado.

Este mismo material puede resultar muy útil además como una herramienta de trabajo cotidiana para el profesor, con la cual puede mejorar sus clases incluyendo animaciones, simulaciones y presentaciones en general al material de apoyo. También se considera útil la posibilidad de poder editar estas animaciones en forma cómoda y fácil, manteniéndose al día así con los cambios curriculares, cada día más dinámicos. Así, se puede evitar el trabajo tedioso y poco enriquecedor por parte del profesor de repetir una y otra vez la materia básica del curso, dándole mayor dedicación a guiar a sus alumnos en el aprendizaje, más que transmitirles la información básica, que los alumnos podrían obtener en forma fácil y eficiente a través de otros medios.

1.1 Motivación

Prácticamente todos los computadores estándares que se venden actualmente ya tienen capacidad de manejar audio y video y la suficiente memoria como para no tener problemas en reproducir presentaciones de alta calidad. Sin embargo, aunque es fácil reproducir este tipo de ayudas, su producción sigue requiriendo en general de software y en muchas ocasiones hardware especializado y aparatoso, con el cual no se cuenta en la mayoría de los casos. Esto causa que en la práctica, la preparación de material de este tipo implique más trabajo que el aporte que brinda. Sin embargo, lo que se necesita en la mayoría de las ocasiones no es una herramienta para producir presentaciones de calidad de estudio ni que aprovechen todas las características de la tecnología de punta, sino más bien algo que permita realizar el trabajo en forma cómoda y rápida. Esto no deteriora el resultado final, que es el aprendizaje del alumno.

Un punto adicional es la posibilidad de reusar objetos de presentaciones existentes, lo cual permitiría crear una biblioteca de objetos básicos, que se pueden usar como bloques en la construcción de nuevas presentaciones y animaciones. Teniendo esta posibilidad, se aumenta enormemente la posible productividad de un profesor o presentador, al poder construir en forma fácil, a partir de componentes disponibles, presentaciones o clases de calidad bastante aceptable y que permiten mejorar mucho la receptividad de las ideas, y disminuir el tiempo necesario para entender conceptos que pueden resultar complejos de expresar en palabras, pero triviales de transmitir a través de otras vías.

1.2 Relevancia y alcance

Al contar con un sistema que permita realizar en forma fácil, amigable y efectiva este tipo de presentaciones, se puede entonces aprovechar de mejor manera la capacidad de los docentes. No es necesario que los profesores usen demasiado tiempo en explicar conceptos, y pueden usar ese tiempo en mejorar la calidad de las presentaciones, motivar a los alumnos y estar disponibles para consultas. Los alumnos podrán realizar solos la labor de adquirir conocimientos, y contarán con el profesor para ahondar en asuntos específicos, resolver dudas que puedan haber quedado y sobre todo guiarlos en el aprendizaje.

A pesar de que el enfoque que se da al trabajo es para usos docentes, también es útil poder contar con una herramienta de este tipo en el área empresarial. Es posible usar presentaciones interactivas para mejorar la comprensión de conceptos por parte de potenciales clientes y aumentar así las ventas o apoyar la presentación de aspectos técnicos, tanto internos como externos, entre otros usos.

1.3 Estado de avance y logros

El proyecto global [1] de creación de la herramienta para crear presentaciones envasadas está dividido en varias partes. Algunas ya se encuentran terminadas y en período de prueba, como es el caso de los procesos *non-preemptive* y los elementos gráficos usados. Otras componentes de la herramienta son las animaciones, la captura y reproducción del sonido y el editor de animaciones.

El objetivo de este trabajo es diseñar una API para la creación, manipulación y uso de las animaciones. Este diseño se implementó a modo de prototipo usando los componentes ya creados, y esta implementación servirá para validar la API definida y será la base para diseñar los componentes faltantes.

Con el componente de las animaciones en funcionamiento, será posible construir una biblioteca de animaciones que podrá ser usada para crear animaciones complejas, y sobre todo para probar el funcionamiento en conjunto de los componentes existentes.

[1] El proyecto global es explicado en el capítulo 2

Capítulo 2

Descripción del proyecto global

Este trabajo forma parte de un proyecto general, el cual consiste en desarrollar una herramienta gráfica que permita crear *cursos envasados*. La herramienta trata de aprovechar todos los beneficios que brinda un aula de clases y evitar sus debilidades. Es así como se pretende involucrar todos los sentidos del alumno y permitir la repetición y la edición de las exposiciones. El uso de los sentidos, en conjunto con posibilitar la experimentación, permitirá al alumno seguir su propio ritmo a seguir, optimizando con ello la atención y la concentración necesaria.

A pesar de que actualmente existen programas que involucran todos los sentidos del alumno, éstos en general no son simples de usar, o no ofrecen la flexibilidad ni la capacidad deseable. Es por ello que se pretende crear una herramienta que incluya en un solo programa los medios que permitan generar y manipular presentaciones que incluyen animaciones, gráficos, narraciones y otras formas de interacción que aporten a la docencia. Esto permitirá generar apuntes activos de clases, que pueden incluir simulaciones, tests interactivos, y otras posibilidades de dejar al alumno experimentar y aumentar su interés, motivación y participación, con lo cual se logrará una mejor comprensión de la materia. Esta herramienta debe contar por lo tanto con diferentes componentes debidamente integrados en un editor que permita la manipulación y reproducción de animaciones en forma simple y útil.

2.1 Antecedentes

Actualmente, los ambientes de desarrollo de presentaciones tipo *Power Point* están más bien orientados como material de apoyo para la presentación en vivo de un expositor. El énfasis en este tipo de herramientas está en proveer material gráfico de apoyo durante una exposición. Estos ambientes no satisfacen los requerimientos para crear presentaciones envasadas, y nunca fueron pensados para ello.

También existen herramientas para el desarrollo de presentaciones envasadas, como por ejemplo *Director* o *Flash*. El énfasis en este tipo de herramientas está en proveer al expositor los mecanismos para mezclar gráficos, video, audio y mecanismos de interacción y publicarlos por medio de un CD-ROM o el WEB. Estas herramientas son bastante cercanas a lo que se necesita para poder realizar presentaciones envasadas, sobre todo desde el punto de vista del usuario. Pero tienen debilidades fundamentales:

1. La creación de presentaciones es en general compleja, y requiere de programas apartes para manipular la narración y las animaciones.
2. Son programas propietarios y por lo tanto no existen en una tan amplia gama de plataformas. En cambio, el proyecto al que se refiere este trabajo está siendo desarrollado en Java [1], lo cual le brinda portabilidad en forma inmediata.
3. No son suficientemente poderosos. Con el paquete desarrollado en este trabajo, se podría por ejemplo permitir al usuario crear objetos a su discreción, para usarlos dentro de la misma animación. Estos objetos pueden interactuar entre ellos, basados en el funcionamiento de la clase **Actor** [4], y cualquier **Actor** existente puede ser incluido en la animación con un mínimo de cambios en el código [1]. También es posible crear objetos sumamente complejos definiendo su comportamiento mediante programas en Java, y agregándolos a operaciones. En cambio, en los programas existentes, el usuario solamente puede realizar decisiones sobre la ejecución de la presentación.

Existen proyectos como los *Active Essays* [3] creados en el MIT que resaltan los beneficios de la interactividad y la experimentación para enseñar temas específicos. Ejemplos concretos de ensayos activos se encuentran publicados en el *Web* bajo el nombre de *Exploring Emergence* [6] y *Going in Circles* [5] que hacen uso de ***Applets*** y otras herramientas

[1] En el capítulo 3 se describe en detalle la clase **Actor**

para interiorizar al lector con conceptos no triviales de entender. Queda completamente claro con estos ensayos el potencial que tienen la experimentación y la interactividad, dado que explicar el mismo fenómeno con métodos tradicionales resultaría notablemente engorroso. Este tipo de proyectos se pueden ver beneficiados con una herramienta que permita crear animaciones para representar conceptos que sería difícil de expresar en palabras.

2.2 Justificación

En la actualidad, los alumnos basan sus aprendizajes en dos paradigmas:

1. clases expositivas
2. libros o apuntes

2.2.1 Sistema expositivo y Sistema de libros o apuntes

Las clases expositivas son mejores que los libros o apuntes, porque es sabido que cuando se involucra el sentido de la vista, el sentido auditivo y el sistema motor al mismo tiempo, el aprendizaje es mayor que cuando se involucra sólo el sentido de la vista (el caso de la lectura de apuntes o libros).

A pesar de ello, las preferencias de los alumnos están divididas. Algunos prefieren aprender de libros o de fotocopias de los apuntes de sus compañeros, porque esto les permite aprender según su propio ritmo. Si junto con leer los apuntes, se escribe un resumen de lo leído, se acerca a la efectividad del sistema expositivo en un aula de clases, ya que sólo se está dejando fuera del proceso al sentido de la audición.

2.2.2 Sistema interactivo

La interactividad es la respuesta por medio del lenguaje al lenguaje. Un medio se considera interactivo cuando tiene la capacidad de implicar al aprendiz activamente a la actividad que viene implícita en el diseño. Es por ello que un medio interactivo saca provecho de la inteligencia del receptor invocando una decisión continua, manteniendo además activos sus sentidos y concentración.

Partiendo de la base que el uso activo de los sentidos que tiene lugar en un aula de clases tradicional aumenta la calidad de las clases o exposiciones, es válido preguntarse si esta calidad es mejorable. Encontramos que en efecto la interactividad en un aula está limitada por la cantidad de expositores, que es generalmente uno por varias decenas de alumnos. Si se permite una interactividad más directa, en la que *cada alumno* más que el *grupo de alumnos* tiene un contacto directo con el expositor, aumenta el uso de los sentidos y por lo tanto mejora la calidad de la exposición.

Otra ventaja que tiene un medio interactivo como el que provee un computador es la capacidad de repetir una exposición. Esto tiene como consecuencia inmediata que el autor de la exposición puede mejorar las partes débiles de la presentación, y sacar buen provecho a las fortalezas de la misma.

De esta manera, se está mejorando incrementalmente la calidad de la presentación, y la facilidad de reproducción de un medio digital permite masificar la cantidad de las presentaciones. No existe por lo tanto un compromiso entre calidad y cantidad de presentaciones, como es el común en un medio tradicional como el aula de clases.

Otro aspecto que es posible incluir en cierta medida es la experimentación del alumno. Cuando se le da la posibilidad de usar no sólo sus sentidos sino también su curiosidad para descubrir novedades relacionadas con el tema tratado, aumenta la atención del alumno y se facilita la comprensión de la materia.

2.3 Objetivos del proyecto

El material de estudio ideal sería entonces uno que ofreciera a los alumnos imágenes y animaciones de los conceptos en conjunto con una narrativa que los explica. La herramienta que se pretende desarrollar es un ambiente que ayude a confeccionar este tipo de material.

Se intenta con esto aprovechar mejor las posibilidades que se presentan al usar medios dinámicos e interactivos para comunicarnos, los cuales la mayoría de las veces son subutilizados, al plasmar en un medio digital y dinámico la misma estática y unidireccionalidad de los medios tradicionales como el papel. Con unidireccionalidad nos referimos a los métodos expositivos más que interactivos, en los cuales el alumno es sólo receptor, y el profesor sólo emisor de la información.

Es importante resaltar, en lo posible con ejemplos concretos, cómo se puede mejorar la capacidad de transmisión de ideas y conceptos al utilizar mejor los medios que ya se encuentran presentes en la mayor parte de los hogares y las organizaciones, no sólo educativas, sino también comerciales.

Concretamente, lo que se pretende con el desarrollo de esta herramienta es poder crear cursos envasados, que puedan publicarse en un servidor `WEB` o `FTP`, para que sea obtenido por los interesados y reproducido en sus computadores localmente, con el uso de algún programa que también formará parte de la herramienta.

El sonido, y en particular la narración, es un factor importante que puede complementar muy bien una animación. Puede servir tanto para acompañar la animación como efectos de sonido, o dar explicaciones relativas al desarrollo de una animación. En cada caso, es importante mantener un sincronismo entre la animación y el sonido. Por ello, se debe considerar la narración al diseñar la metáfora y las estructuras a usar.

No es trivial sincronizar una narración con una animación, sobre todo si se considera que ambos deben ser fácilmente modificables. Por ejemplo, si se está explicando verbalmente una animación, y se agrega un paso a la animación, se desearía que la narración no perdiera el sincronismo a partir de ese punto, pues significaría tener que volver a realizar todo el trabajo de creación de la narración. En ese caso, debiera ser posible editar la narración, para agregar sonido en la nueva parte de la animación, si se trata de una explicación. En cambio, para música de fondo o acompañamiento no necesariamente sincronizado con las acciones, debiera continuar sin interrupciones.

La edición del sonido es también útil al realizar modificaciones o depuraciones a algunas partes de la narración, y no se desea volver a realizar todo, sino repetir solamente una parte y acoplarla al resto de las secuencias de sonido.

2.4 Componentes

El ambiente está siendo desarrollado en la plataforma *Java* y se ha descompuesto en paquetes independientes. Estos paquetes serán explicados con mayor detalle en el capítulo 3, y son los siguientes:

- *Biblioteca de objetos gráficos* (paquete `anim.glyphs`)

  Esta biblioteca contiene la clase **Glyph** [7], que permite la creación y manipulación de objetos gráficos como polígonos, texto, imágenes, etc. El principal objetivo de esta clase es proveer una interfaz común para todos los componentes gráficos que se manejarán.

- *Procesos `non-preemptive` con capacidad de envío de mensajes* (paquete `anim.actors`)

  Los actores son objetos de una clase **Actor** [4] que provee procesos non-preemptive, con lo cual no es necesario que el programador se preocupe de secciones críticas sobre objetos compartidos entre distintos actores. Además, los actores se pueden comunicar entre ellos a través de mensajes tipados.

- *Mecanismos de animación* (paquete `anim.operations`)

  La clase **Operation** provee una interfaz común para todas las operaciones que se podrán realizar sobre los **Glyph**. Estas operaciones son por definición interruptibles y reversibles. Con estas características, es posible recrear cualquier estado dentro de una animación compleja, lo cual permite la edición gráfica de las animaciones. Para lograr la interrupción de animaciones y facilitar la programación, en la creación de estas clases se usa a **Actor** como superclase de la implementación de estas operaciones. Este es el paquete desarrollado durante el desarrollo de la presente memoria.

- *Mecanismos de captura y reproducción de la narración* (aún no implementado)

Es necesario crear los mecanismos de captura y reproducción de la narración, los cuales no corresponden a la biblioteca de **Glyph**. Una vez programados estos módulos, se podrán acoplar al paquete `anim.operations` para proveer sonido en las animaciones. Es importante tomar en cuenta al implementar estas operaciones el papel que juega la sincronización de sonido y animación y proveer alguna forma de alcanzar esta sincronización de manera fácil y sencilla.

- *Edición gráfica de animaciones* (aún no implementado)

  El autor de presentaciones necesitará un editor ***WYSIWYG*** para poder crear y editar las animaciones en forma cómoda y rápida. Las animaciones, como se verá en los capítulos 5 y 6, tienen una estructura de árbol. El principal desafío para este punto es crear una metáfora para presentar gráficamente las animaciones, considerando la estructura de árbol de las mismas. Además, es necesario considerar que los nodos (que se llaman clases ***compositoras***) del árbol mencionado ejercen un efecto distinto sobre el tiempo de la animación. Por ejemplo, en una animación secuencial el tiempo total es la suma de las partes, mientras que en una animación paralela el tiempo es la mayor de las partes.

Capítulo 3

Descripción de `anim`

El paquete `anim` es la biblioteca que provee todo lo necesario para poder realizar las animaciones que incluyen narración y despliegue gráfico. También deberá proveer las herramientas para editar estas animaciones, y obviamente habrá un programa que las reproduzca.

Para ello, se usa la metáfora de un teatro, en el cual se presentan actores, formando una animación. Los actores cuentan con elementos gráficos básicos, los cuales pueden ser modificados tanto en forma, posición y orientación. También pueden comunicarse entre sí mediante mensajes tipados con otros actores.

La importancia de este paquete está en que define, mediante la clase **Theater**, la estrategia de animación que se usa para lograr el efecto deseado.

A su vez, esta biblioteca está dividida en paquetes de *Java*, agrupando las funcionalidades requeridas en componentes específicos. Durante el desarrollo de este trabajo, fue necesario realizar algunas modificaciones a los paquetes existentes, para mejorar tanto la facilidad de uso como las capacidades de los paquetes.

- `anim.tools` implementa las clases necesarias para el paquete `anim.actors`. Entre ellas, la más importante es la clase **Agenda** que coordina a los `actores`, realizando la programación de eventos y administrando el tiempo de simulación.

- `anim.actors` tiene como principal clase la **Actor** [4], que provee procesos `non-preemptive` que pueden compartir objetos y se comunican entre sí mediante mensajes tipados. Con esta clase, se libera al programador que use esta biblioteca de preocuparse por secciones críticas en su código, y facilita la sincronización.
- `anim.glyphs` por un lado define la clase **Glyph** [7], que agrupa todos los elementos gráficos básicos que se podrán usar dentro de las animaciones. También provee una clase **Theater**, que representa al teatro donde se presentan los `actores`. Esta clase y provee el escenario y determina la estrategia de animación.
- `anim.operations` es el paquete que define las operaciones que se realizaron sobre los **Glyph** dentro de un **Theater**. Para ello, se definió la clase **ExecutionActor** que implementa estas operaciones, y la clase **Operation** que provee una interfaz para el programador. Este paquete es el que se desarrolló durante el trabajo, y es descrito con mayor detalle en los capítulos 5 y 6.

Los componentes considerados como parte del proyecto, pero aún no implementados son los siguientes:

- *narración*: debe permitir la creación y edición de sonido que acompañará las animaciones.
- *coordinación narración-animación*: como parte del componente de creación y edición, es necesario definir cuidadosamente la coordinación entre el narración y la animación.
- *editor gráfico de animaciones*: la herramienta que incorpora todas las funcionalidades y las presenta al usuario.

3.1 El paquete `anim.actors`

Este paquete está basado en la memoria de *Pablo Pozo* sobre **Herramientas de simulación basadas en threads de Java** [4]. Su función es proveer procesos `non-preemptive` que pueden compartir objetos entre ellos. En cualquier momento, solamente un actor está activo y tiene el control del procesador, mientras el resto de los actores se encuentran suspendidos. El actor en ejecución es denominado "current actor" y transfiere el control del procesador al enviar o esperar mensajes o retardar su ejecución.

Existe una **Agenda** que controla el tiempo de simulación y se encarga de despertar a los actores que corresponda a medida que avanza el tiempo de simulación. Se trata de que el tiempo de simulación y el tiempo real coincidan dentro de lo posible. Como por lo general el tiempo de simulación se ejecuta con mayor rapidez que el tiempo real, la **Agenda** introduce las pausas necesarias.

Los *actores* pueden comunicarse entre sí usando envío de mensajes tipados. Estos mensajes pueden ser síncronos o asíncronos, dependiendo del método usado para enviarlos, como se ve más adelante. Así, es posible coordinar un grupo de actores usando mensajes. Cada mensaje tiene definido su tipo de acuerdo a un objeto de la clase **Symbol**, y el receptor de los mensajes puede elegir qué mensajes recibe a partir de su tipo. Para ello, cada actor puede definir un conjunto de símbolos (**SymbolSet**) por defecto, que determina qué mensajes se reciben. Los mensajes de otro tipo permanecerán encolados hasta que el **Actor** decida recibirlos.

La *API* para el envío de mensajes comprende los siguientes métodos:

- `public void call(Msg msg)`: envía el mensaje `msg` al actor sobre el cual se invoca el método, quedando bloqueado hasta que ese actor invoque `reply(msg)`. No es necesario que el objeto que envía el mensaje sea un actor. Por ejemplo, para enviar un mensaje `msg` al **Actor** `a`, se invoca `a.call(msg);`

- `public void post(Msg msg)`: envía el mensaje `msg` al actor, pero no se bloquea.
- `public void reply(Msg msg)`: desbloquea al emisor del mensaje. Si el emisor no está bloqueado porque envió el mensaje usando
- `public void enable(Symbol s)`: agrega el **Symbol** `s` al **SymbolSet** por defecto.
- `public void disable(Symbol s)`: elimina el **Symbol** `s` del **SymbolSet** por defecto. `post()`, simplemente retorna.

- `public Msg accept(SymbolSet set, double delay)`: espera un mensaje cuyo tipo esté dentro del **SymbolSet** `set`. Si ha esperado un tiempo de `delay` milisegundos, retorna `null`. Si ya hay un mensaje encolado que corresponda al **SymbolSet**, retorna de inmediato el mensaje. Si `delay` es igual a `Double.MAX_VALUE`, permanece bloqueado hasta recibir un mensaje, independientemente del tiempo transcurrido. Si el parámetro `set` es `null`, se retorna cualquier mensaje, sin importar el tipo. El resto de los métodos `accept()` de esta clase invocan a este método con los parámetros adecuados.
- `public Msg accept()`: espera un mensaje cuyo tipo esté dentro del **SymbolSet** por defecto. Permanece bloqueado hasta poder retornar dicho mensaje.
- `public Msg accept(SymbolSet set)`: espera un mensaje cuyo tipo esté contenido en `set`, permaneciendo bloqueado.
- `public Msg accept(double delay)`: espera un mensaje cuyo tipo esté contenido en el **SymbolSet** por defecto, por un máximo de `delay` milisegundos.
- `public Msg acceptAny(double delay)`: espera cualquier mensaje por un máximo de `delay` milisegundos.
- `public Msg acceptAny()`: espera cualquier mensaje, quedándose bloqueado.
- `public Msg forward(Msg msg)`: el mensaje `msg`, se reenvía al actor sobre el cual se invoca este método. Es obligatorio que el mensaje haya sido recibido por el actor que invoca el método, o se generará un error. El actor que reenvía el mensaje se olvida del mismo, y el **Actor** que lo recibe deberá invocar `reply()` si corresponde.

3.2 El paquete `anim.glyphs`

3.2.1 La clase Glyph

Un **Glyph** [7] es un elemento gráfico que representa una figura geométrica, como un polígono, un texto o una imagen. Cada **Glyph** puede contener uno o más *`puntos de control`*, color de fondo y de primer plano. La clase **Glyph** tiene un sistema de coordenadas local para definir los *`puntos de control`* geométricos que necesite, pero no tiene referencia a su posición dentro de un canvas.

Para evitar recalcular un **Glyph** complejo cada vez que se repinta, se mantiene un cache del mismo. Cada vez que se realiza un cambio, es invocado el método `touch()`, el cual se encarga de señalar que el cache está "sucio". En la siguiente ejecución de `paint()` se recalcula el **Glyph**.

Algunos ejemplos de objetos **Glyph** implementados son los siguientes:

- **GImage** despliega una imagen. No tiene *puntos de control*.
- **GText** representa un texto. Tampoco tiene *puntos de control*, solamente los colores de primer plano y de fondo.
- **GPolygon** es un polígono cerrado representado por una serie de puntos. Las coordenadas usadas por esos puntos son locales, y corresponden a los *puntos de control* de este **Glyph**.
- **GArrow** es una subclase de **GPolygon**, que solamente tiene un punto de control (la dirección). No tiene posición, ya que esa coordenada la maneja el **Canvas** y no el **Glyph**.

3.2.2 La clase Theater

Un objeto **Theater** contiene objetos de clase **Glyph**, y los despliega dentro de un canvas que forma parte del **Theater**. Por cada **Glyph**, es el **Theater** que define la posición que ocupa dentro del **Canvas** en el cual se despliegan. Los **Glyph** son mantenidos en una lista doblemente enlazada, y el orden en el cual aparecen en la lista determina la posición relativa entre los **Glyph**, determinando qué **Glyph** estará sobre otro. La clase **Theater** es una subclase de **Actor** y por lo tanto permite el intercambio de mensajes con otros actores, y se garantiza la ejecución mutuamente exclusiva del teatro y los demás actores.

El elemento más importante de un **Theater** es el **AnimCanvas**, dentro del cual se despliegan los **Glyph** correspondientes y se controla la ejecución de la animación. Para evitar el efecto de parpadeo que se puede producir al existir una demora en la actualización de un cuadro, se usa la técnica de *double buffering* para desplegar los elementos gráficos.

3.3 Estrategias de animación

Antes de describir las posibles estrategias de animación, es necesario entender el concepto de tiempo de animación y tiempo real. El tiempo real es el tiempo tal y como se vive en el mundo real. Al simular un mundo virtual, es necesario que éste tenga su propio tiempo asociado. Este tiempo se puede manejar al antojo del programador, pudiendo avanzarlo, detenerlo, e incluso cambiar la velocidad con la cual avanza o retrocede. Al momento de reproducir una animación creada con la biblioteca desarrollada, se pretende que el tiempo de simulación del mundo virtual coincida en lo posible con el tiempo real.

Para realizar una animación, se apunta a tener una cierta cantidad fps de cuadros por segundos para lograr la ilusión de un movimiento continuo. Generalmente el número de cuadros por segundo está entre 20 y 30; por ejemplo, en el cine se usan 24 cuadros por segundo, mientras que en la televisión se utilizan 30.

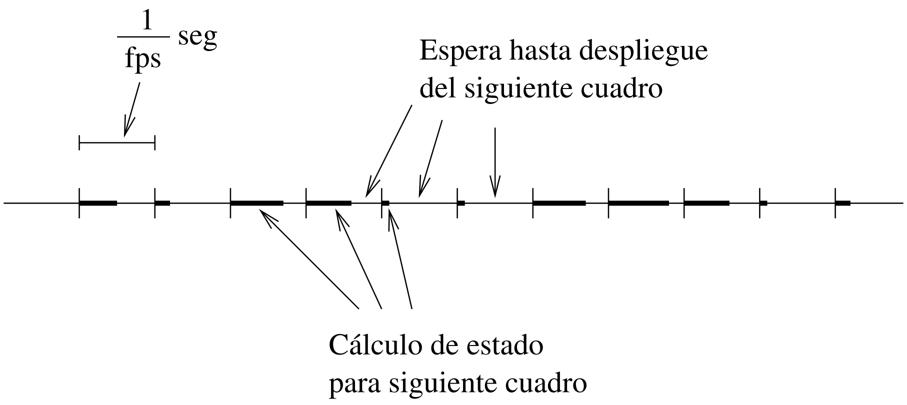


Figura 3.1: Discretización del tiempo

Como se necesita mostrar una secuencia de cuadros, se puede flexibilizar la condición de mantener iguales los tiempos es necesario hacer coincidir el tiempo de simulación y el real sólo en los instantes en que se muestra un cuadro. Así, se logra discretizar el tiempo, como se muestra en la figura 3.1. Se ve que el cálculo para el siguiente paso se realiza inmediatamente concluido el despliegue del cuadro actual. De esta manera, el tiempo de simulación se adelanta al tiempo real, y una vez alcanzado el tiempo de simulación para el siguiente cuadro, se detiene y espera hasta que haya transcurrido el tiempo real correspondiente. Es claro entonces que se tiene un tiempo de $\frac{1}{fps}$ segundo para realizar los cálculos entre cada paso.

En general, el tiempo que se demora el computador en realizar los cálculos para el siguiente paso debiera ser menor a $\frac{1}{fps}$, y se debe incluir una demora programada para que el tiempo de simulación coincida con el tiempo de ejecución (real). En el caso de que los cálculos se demoren más que eso, es necesario tomar una decisión sobre el camino a seguir:

1. Dibujar todos los cuadros por segundo (en tiempo de simulación), resultando en una de-sincronización de los tiempos. Se mantiene la cantidad de cuadros a mostrar, retrasando el tiempo de simulación. Así, la animación se verá excesivamente lenta pero tendrá la continuidad necesaria para poder entenderla al presentar todos los cuadros programados.

2. Disminuir la cantidad de cuadros por segundo a mostrar para mantener el tiempo de ejecución sincronizado con el tiempo real. En este caso, se redibujará cada vez que se pueda, y el tiempo de simulación avanzará más entre cuadros si el cálculo es largo. El resultado es que la animación se verá a saltos, y posiblemente no se entienda del todo, al carecer de la continuidad necesaria si transcurre demasiado tiempo.

En este proyecto se definió un tiempo máximo de cálculo para cada paso. Si el cálculo se demora más, se retrasa el tiempo de simulación con respecto del tiempo real, siguiendo el primero de los esquemas señalados. Si para cierta animación es recurrente llegar al extremo de demorar el tiempo de simulación por sobre el tiempo real, es posible modificar la cantidad de cuadros por segundo a mostrar. Esto es posible tanto en la construcción del **Theater** como durante la ejecución. Incluso es posible crear un **Theater** que ajuste automáticamente la cantidad de cuadros por segundo a deplegar en base a la cantidad de saltos en el tiempo de simulación que se hacen necesarios.

Un posible problema que surge de este enfoque es cómo adaptar la narración a este método. Es claro que no se puede modificar la velocidad de una narración o sonido sin producir efectos aberrantes. Posiblemente sea necesario usar un método distinto para sincronizar el tiempo de la narración con el tiempo de simulación.

Una vez definido el manejo de las demoras excesivas en los cálculos, la forma más intuitiva de realizar una animación consiste en los siguientes pasos definidos por el pseudocódigo:

```
  while(true) {
    PintarCanvas();
    calcularProximoPaso();
    esperarRefresco();
  }
```

Si el tiempo de ejecución de `PintarCanvas()` sumado al de `calcularProximoPaso()` no superan al tiempo real, es necesario esperar hasta que este tiempo se cumpla. En cambio, si el tiempo de ejecución supera al tiempo real, no se espera y se continúa de inmediato.

En Java, no se invocan directamente los métodos que pintan el canvas. En cambio, cada objeto a ser pintado debe proveer un método `paint(`**Graphics g**`)`, el cual es invocado por el thread del `AWT`[1]. Por esto, el código en Java sería más parecido al siguiente:

```
class AnimCanvas {

  ...

  paint() {
    PintarCanvas();
    calcularProximoPaso();
    repaint(delay)
  }
}
```

El problema de este código es que el método `repaint(int delay)` no provoca que se repinte en el tiempo exacto, sino en forma aproximada. No existe otra forma de controlar cuándo se repinta una componente. Por esto, la forma de animar debiera tratar de mantener un refresco de `delay` milisegundos, en cada paso revisar cuánto realmente avanzó el tiempo y según eso repintar el **Canvas**. Algo como sigue:

[1] `AWT` es el kit gráfico de Java

```
class AnimCanvas {

  ...

  paint() {
    int step = cuantoTiempoPaso();
    calcularEstePaso(step);
    PintarCanvas();
    repaint(delay)
  }
}
```

En la implementación, el método `calcularEstePaso(int step)` simplemente entrega el control del procesador al siguiente actor y programa la **Agenda** para volver a obtenerlo en `step` milisegundos. Esto es posible, ya que el **Theater** es a su vez objeto de la clase **Actor**. Después de un tiempo de animación de `step` milisegundos, este actor retoma control de la ejecución y ejecuta `PintarCanvas()`. Se ve que los tiempos de animación y real no están exactamente sincronizados, sino que el tiempo de animación siempre va unos instantes atrasado, según lo que demore el cálculo del cuadro actual. Es por eso que el `delay` no debe considerarse un valor exacto sino un mínimo. El tiempo exacto transcurrido entre cada cuadro dibujado en la pantalla será entonces la suma entre `delay` y lo que se demore el cálculo del cuadro.

3.4 Modificaciones realizadas

Durante la implementación de un programa de prueba, descrito en el capítulo 4, fue necesario incorporar ciertos cambios a la clase **Theater**.

3.4.1 Capacidad de eliminar mensajes encolados aún no procesados (`flush()`)

Dentro del trabajo, se programó una aplicación con interfaz gráfica. Esta interfaz gráfica usa un mecanismo de programación orientado a eventos, mientras que el mecanismo usado por los actores es orientado a mensajes. Fue necesario entonces realizar una conversión entre eventos a mensajes. Esta conversión no es del todo transparente, ya que es necesario suscribir actores a eventos, y luego de-suscribirlos. En algunos casos, pueden quedar mensajes encolados que no han sido procesados. Sería un error mantenerlos encolados, pues serían procesados una vez que se vuelva a suscribir al evento en cuestión. El método `flush()` implementado permite eliminar esos mensajes.

3.4.2 Conversión entre coordenadas de dispositivo (Theater) y coordenadas de Glyph

Una de las tareas esenciales de un editor gráfico es modificar los *puntos de control* de un **Glyph**. Estos *puntos de control* están en coordenadas locales del **Glyph** y el puntero con el cual se modifican los puntos está en coordenadas del **Canvas** u otro dispositivo. Por ello, se hace necesario un mecanismo simple que traduzca entre estas coordenadas.

3.4.3 Selección de Glyph mediante `hit()`

Para poder seleccionar un **Glyph** dentro de un canvas en forma gráfica, se necesita contar con un método que entregue un **Glyph** que intersecte cierto rectángulo. Si existe más de un **Glyph** con esta característica, se retorna el que esté sobre los demás. Si no existe ninguno, se retorna `null`.

La implementación de este método incluyó la creación de un *boundBox* de clase **Shape**. Este *boundBox* incluye todo el **Glyph** y en base a él se determina si un punto intersecta a un **Glyph** particular.

3.4.4 Serialización

Si se pretende implementar una biblioteca de operaciones, lo esencial es poder guardar animaciones y operaciones en general, para luego poder volver a cargarlas, para su edición o reproducción. El paquete `anim` no tuvo muchos cambios para lograr que sus objetos fueran *serializables*.

Los cambios hechos corresponden más que nada a mejorar la eficiencia, marcando variables temporales y de cache como *boundBoxes* para no guardar su estado.

Capítulo 4

Uso de `anim`: Implementación de Editor

Uno de los objetivos que quedan aún pendientes para el proyecto general de cursos envasados es la creación de un editor gráfico de animaciones. Para probar la factibilidad y las dificultades del paradigma usado, se implementó un editor simple que tuviera la capacidad de crear, seleccionar y mover elementos de tipo **Glyph**, sin usar aún el paquete `anim.operations`, que es el desarrollado en este trabajo. Este editor rudimentario puede ser usado como base para crear el editor gráfico que aún falta por implementar, reusando código y evitando cometer errores conocidos.

La principal dificultad para la programación de este editor resultó ser la incompatibilidad existente entre los métodos de programación orientado a eventos y orientado a mensajes. Por otro lado, como ya se mencionó en el capítulo 3, para el correcto funcionamiento de esta clase fue necesario realizar algunas modificaciones al paquete `anim`. En este capítulo se explicarán los detalles de los cambios y su razón de ser.

4.1 Traducción de Eventos a Mensajes tipados

En la programación de este editor fue necesario usar en un mismo programa dos mecanismos de programación: orientado a eventos y orientado a mensajes. Esta mezcla de mecanismos tiene que existir con la implementación actual, puesto que la AWT de *Java*, que maneja todos los aspectos gráficos, es naturalmente orientada a eventos, y los procesos definidos por la clase **Actor** están orientados a mensajes tipados.

La AWT de *Java* genera eventos a los cuales los componentes gráficos se pueden suscribir. Estos eventos incluyen movimientos del puntero, cambios en el estado del componente, etc. La suscripción consiste en incluir un método (handler) en una lista, el cual será ejecutado cada vez que ocurra el evento respectivo. Es en ese método donde se debe realizar la traducción del evento, enviando un mensaje de cierto tipo dirigido a un **Actor** determinado.

Un primer problema se presenta justamente al enviar el mensaje. En general, los actores que envían un mensaje quedan bloqueados hasta que el mensaje es recibido y respondido con un `reply()` por parte del receptor. Si se hiciera esto mismo al traducir un evento, se estaría bloqueando el thread de la AWT hasta que el actor correspondiente invoque el método `reply(msg)`, como se puede ver en la figura 4.1. Esto tiene como consecuencia que no se puede generar ningún otro evento y se bloquea toda operación gráfica, como repintado, actualización, etc. durante el período de procesamiento del mensaje.

Como no se tiene de antemano el control sobre el tiempo que se bloquea el thread, y este tiempo puede ser arbitrariamente largo, es necesario evitar el bloqueo. Para ello, se puede encolar un mensaje sin esperar el respectivo `reply()`, usando el método `post()` en vez de `call()`. Sin embargo, con esta solución se generan otros problemas, como por ejemplo:

- pueden encolarse varios mensajes del mismo tipo, cosa que en muchas ocasiones no tiene sentido. Por ejemplo, cuando se encolan dos mensajes de movimiento del puntero, posiblemente sería coherente ignorar el primer mensaje y tomar solamente el último, que es la posición más cercana a la real.
- al desuscribir el método de un evento, pueden permanecer mensajes no procesados aún encolados. En general, el **Actor** que se ha desuscrito de dicho evento no cuenta con recibir esos mensajes. Si el mismo **Actor** después se vuelve a suscribir a ese evento,

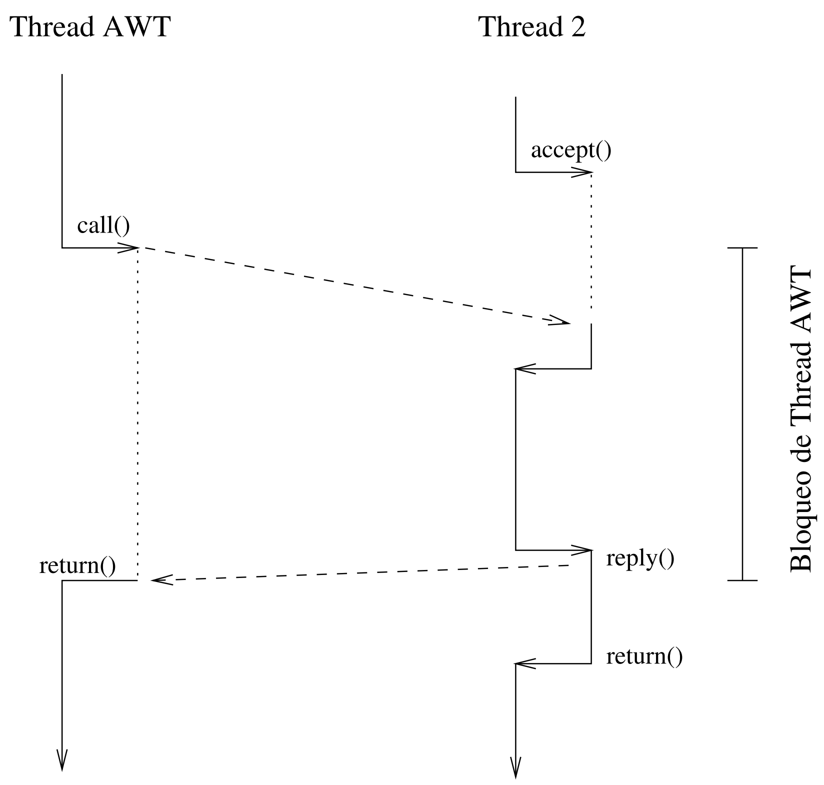


Figura 4.1: Bloqueo del thread de AWT

tomará los mensajes encolados anteriormente como si fueran nuevos, lo cual puede producir resultados inesperados y debiera ser considerado un error.

Debido a estos problemas, se implementó un método que permite eliminar mensajes encolados aún no procesados. Usando este método, es posible solucionar ambos problemas mencionados. Si corresponde, antes de encolar un nuevo mensaje se pueden eliminar todos los anteriores del mismo tipo dirigidos al mismo actor. Esto es útil, por ejemplo para el caso de movimientos del puntero. Además, al desuscribir a un actor de un evento también se eliminarían todos los mensajes del tipo correspondiente, solucionando el segundo problema enunciado.

Sin embargo, la solución presentada es bastante frágil, ya que es necesario que el programador esté preocupado de todos los aspectos relacionados con los eventos y con los mensajes. Así, no se ha solucionado el problema de estar usando dos mecanismos de programación simultáneamente. Solamente se ha provisto una solución temporal para poder programar el editor.

Una buena solución consiste en crear una capa intermedia que haga la función de adaptar las respectivas ***API***. Para realizar esto correctamente, es necesario estudiar el problema con más detalle. Las preguntas que surgen al tratar de diseñar una capa de este tipo son por ejemplo:

- ¿Cuántos eventos deben generar un mensaje? Puede ser una buena aproximación que cada vez que se suscriba a un evento, se recibe a lo más un mensaje. Para recibir otro, sería necesario volver a suscribirse a ese evento.
- ¿Qué significado tiene el orden de los mensajes? Los eventos tienen un orden FIFO, en cambio los mensajes se reciben en distinto orden, dependiendo del tipo de mensaje. ¿Tiene sentido juntar todos los mensajes generados por eventos en un solo tipo?
- ¿Qué pasa si se pierde un evento de tipo MOUSEUP y llegan dos mensajes de tipo MOUSEDOWN? ¿Es eso un error, o se puede considerar correcto? ¿Es necesario inventar un MOUSEUP entre dos MOUSEDOWN consecutivos?
- ¿Cómo garantizar que la solución sea robusta?

4.2 Diseño gráfico

Gráficamente, el editor programado consta de un *panel de control*, ubicado arriba como se aprecia en la figura 4.2. Este panel contiene checkboxes para crear distintos tipos de **Glyph**, y un textbox para ingresar nombres de archivo o texto. En el centro se encuentra el **Canvas** dentro del cual se despliegan los **Glyph** creados. En la figura 4.2 se han creado diferentes **Glyph**, de tipo **GText**, **GImage** y **GPolygon**.

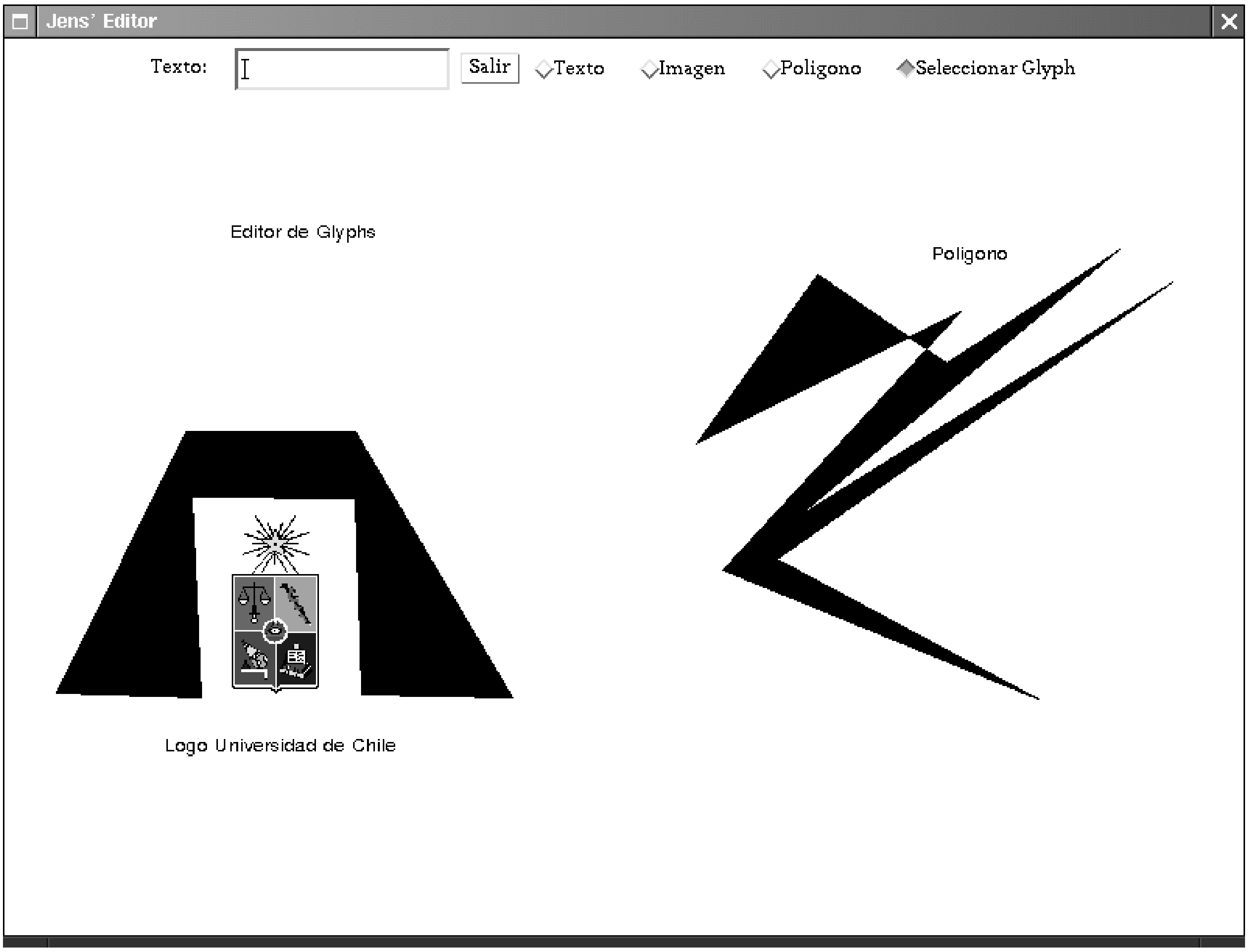


Figura 4.2: Snapshot del editor

4.3 Creación de Glyph

Cada **Glyph** maneja sus propios puntos de control y variables específicas, las cuales se deben tener en cuenta a la hora de crearlos. Por cada tipo de glyph a crear, se ha agregado un checkbox que determina el modo de creación. Solamente un checkbox está activo en cada momento. Al cambiar el estado de los checkboxes, se envía un mensaje a través de un listener al **Theater** que contiene al **Canvas**. Esto causa la invocación del método respectivo para crear el tipo de **Glyph** seleccionado gráficamente. Este método esperará los mensajes adecuados para proceder con la creación de los **Glyph**.

4.3.1 GImage

Para desplegar una imagen, se crea un glyph de tipo **GImage**. Este objeto necesita el nombre de un archivo, el cual es ingresado con anterioridad en el *textbox* del panel de control. Luego, se selecciona el checkbox de creación de un **GImage** y se presiona el botón del puntero en el lugar donde se desea ubicar la imagen dentro del **Canvas**. El código que crea el **Glyph** es básicamente el siguiente:

```
private void addImagenActor() {
  Msg mensaje;
  for (;;) {
    mensaje = accept(MOUSEDOWN);
    actores.addElement(new ImageActor(miControlPanel.miTextf.getText(),
                       ((MsgPos)mensaje).pos, this));
  }
}
```

Sin embargo, es necesario poder recibir además otros mensajes. Entre ellos, los mensajes referentes al cambio de modo de creación de **Glyph**. Lamentablemente, la forma de hacerlo es recibir todo tipo de mensaje y si no corresponde al esperado, volver a encolarlo y salir del método. Esto requiere que el programador esté consciente de que es necesario recibir todo tipo de mensajes, y por lo tanto resulta poco robusto. Además, la orientación a mensajes fue elegida justamente para que el programador de un actor no deba preocuparse de los mensajes que no le interesen. El código quedaría como sigue:

```
private void addImagenActor() {
  Msg mensaje;
  for (;;) {
    mensaje = acceptAny();
    if (mensaje.getSym() != MOUSEDOWN) {
      requeue(mensaje);
      return;
    }
    actores.addElement(new ImageActor(miControlPanel.miTextf.getText(),
                       ((MsgPos)mensaje).pos, this));
```

```
    }
  }
```

Se podría implementar una solución en la cual exista un actor por cada tipo de **Glyph** posible de crear, y en vez de invocar un método dentro del **Theater**, simplemente enviar un mensaje (a través de `post()`, para evitar el bloqueo del **Theater**) al actor correspondiente para que cree el **Glyph**. Pero también surge el mismo problema: si se modifica el modo de creación usado en el editor, sería necesario avisar de ello a todos los actores que crean **Glyph**, y el código deberá considerar mensajes que aborten la creación, tal como se hace con la solución implementada. A pesar de que el problema se puede solucionar para **Glyph** que necesitan de un solo evento (y por lo tanto de un solo mensaje) para ser creados, persiste la necesidad de abortar la creación de los más complejos, como por ejemplo los polígonos.

4.3.2 GText

La creación de un **GText** (**Glyph** de texto) funciona en forma casi idéntica al de una imagen: se ingresa el texto a desplegar en vez del nombre de archivo en el textbox, y se presiona el botón del puntero en el lugar del **Canvas** donde se desea que aparezca.

4.3.3 GPolygon

Un polígono tiene más puntos de control y es por lo tanto algo más complejo de crear. Lo que se hace es seleccionar el checkbox de creación de polígonos, y se va presionando el botón en los lugares donde se agregarán puntos de control al polígono. Después de haber terminado con todos los puntos, se debe hacer un *doble click* para terminar el polígono que está en proceso de ser creado. Cada punto de control creado representa un vértice del polígono.

4.4 Selección y movimiento de un Glyph

Al seleccionar el checkbox de selección de **Glyph**, se pueden mover glyphs mediante "drag and drop". En este caso, por cada evento MOUSEDOWN recibido dentro del **Canvas** se ejecuta el método `hit()` del **Theater**.

El método `hit()` del **Theater** intenta intersectar cada **Glyph** que contiene con un cuadrado que contiene al punto sobre el cual se presionó el botón del puntero. Si este método retorna `true`, se ha encontrado un **Glyph** que debe ser seleccionado. De lo contrario, se sigue buscando. Si al terminar de recorrer todos los **Glyph** no se ha dado con ninguno que intersecte el cuadrado, no se selecciona ningún **Glyph**.

Para el caso de una imagen o un texto, el ***boundBox*** es un rectángulo fijo y por lo tanto el método `hit()` es ejecutado con la rapidez necesaria. Pero en el caso de un polígono medianamente complejo, junto al uso de una implementación de *Java* no demasiado eficiente, es posible que la invocación del método `hit()` incorpore una demora inaceptable. Un ejemplo de una implementación de *Java* que genera una demora así es el *JDK 1.2 preV2* para linux, con el cual no es raro encontrar demoras de unos 14 segundos para la ejecución del método `hit()`. En otras implementaciones, no se aprecia una demora sustancial, por lo cual se decidió mantener el esquema.

Capítulo 5

Diseño de la clase Operation

En este capítulo se revisarán los criterios y las decisiones de diseño utilizados al definir la API de las operaciones. Esta API deberá cumplir con los objetivos propuestos, que incluyen la simplicidad en el manejo y al mismo tiempo flexibilidad y capacidad en la creación de animaciones. Para permitir la edición de animaciones, las operaciones deben ser interruptibles y reversibles.

En el diseño de la API se usó el patrón de diseño denominado "Composite" por los autores del libro "Design Patterns" [2]. Este patrón de diseño sirve de guía para determinar las decisiones de diseño y su importancia. La idea básica del patrón de diseño es crear un modelo en el cual existen clases de tipo *componente* y clases que agrupan *componentes*, llamadas *compositoras (Composite)*. Estas clases pueden compartir una interfaz común, representada por una superclase. Así, no es necesario diferenciar en general entre clases *componentes* y clases *compositoras*.

5.1 Objetivos que debe cumplir la API

Si se construye una clase en la cual una operación puede componerse junto a otras para producir una nueva operación de mayor complejidad, es posible construir una biblioteca de operaciones básicas. Estas operaciones básicas, y otras de mayor complejidad podrán ser usadas como si fueran una operación simple. A partir de esta capacidad de componer operaciones se puede generar un árbol arbitrariamente complejo de composiciones de operaciones básicas.

La idea del proyecto es poder desarrollar un editor gráfico para producir y modificar operaciones. Es necesario para un correcto funcionamiento que las operaciones sean reversibles, para permitir al usuario navegar por el árbol que genera una operación compleja.

Además, las operaciones deberán ser interruptibles, ya que no es deseable que una operación deba finalizar para poder continuar con otra. Por otro lado, pueden existir operaciones cíclicas o sin límites de tiempo, que deben ser interrumpidas para finalizar.

5.2 El patrón de diseño "Composite"

El objetivo de este patrón de diseño[2] es componer objetos dentro de una estructura de árbol para representar jerarquías parte-todo. De esta manera, se pueden tratar de igual forma objetos simples y objetos arbitrariamente complejos creados mediante estas composiciones.

El uso de este patrón de diseño se asocia generalmente a aplicaciones gráficas, en las cuales es usual tener componentes que se pueden agrupar dentro de otros componentes más complejos. A su vez, estos componentes más complejos se pueden volver a agrupar, formando nuevos componentes aún más complejos. Una primera solución sería definir dos clases de objetos; por un lado los componentes básicos, y por otro los objetos *`compositores`*. El problema de esta aproximación es que el código que use estas clases deberá tratarlas de manera distinta, a pesar de que la mayoría de las veces el usuario los usaría de manera idéntica. Así se vuelve más complejo el uso de las clases, como se observa en el siguiente código:

```
while (operationStack.hasElements()) {
  Object op = operationStack.pop();
  if (op instanceof Operation) {

    ... // Código para operaciones básicas

  } else {

    ... // Código para operaciones compuestas

  }
```

Para evitar este problema, se unifica la API de las clases creando una superclase común, que representa ambas clases: las básicas y los *compositores*. De esta manera, tanto el usuario como los *compositores* mismos no necesitan hacer distinción entre clases básicas y clases de *compositores*, puesto que los pueden tratar como una instancia de la superclase común.

5.3 Aplicación del patrón "Composite" al proyecto

Al aplicar el patrón de diseño al problema específico, se define la clase **Operation**, que es la raíz de la jerarquía. Dentro de esta clase, se crean las operaciones básicas o primitivas como **Movement**, **Creation** y **Deletion**. Por otro lado, están las clases de *compositores* como es el caso de **Sequential** y **Parallel**, agrupadas bajo la clase **Animation**. En este caso, las dos clases componen operaciones, pero lo hacen de distinta manera. La estructura de estas clases se representa en el diagrama de clases simplificado de la figura 5.1. Un diagrama completo se encuentra en la página 55.

5.4 Decisiones de diseño

5.4.1 Referencias explícitas a padres

En el diseño de la jerarquía de operaciones se decidió crear referencias explícitas a los padres. Esto permite la navegación fácil en el árbol de operaciones, y lo más importante es que permite que cada operación esté contenida en un solo ***compositor***.

Esta referencia se hace en la clase **Operation** y por lo tanto toda operación tiene una referencia a un posible padre. En caso de la raíz de una animación, esta referencia es `null`.

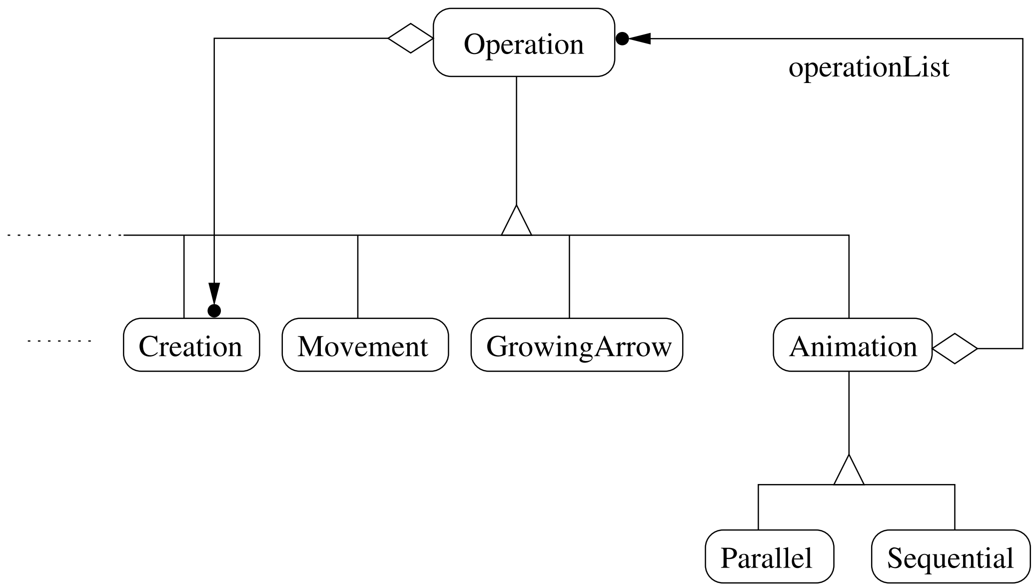


Figura 5.1: Diagrama de Clases (simplificado) de `anim.operations`

5.4.2 Compartir componentes

Existen situaciones en las que es conveniente compartir componentes, por ejemplo para ahorrar espacio en memoria. En este caso, sin embargo, como un componente tiene un solo padre, no viene al caso compartir componentes.

5.4.3 Maximizar la interfaz de Operation

Uno de los objetivos de este patrón de diseño es minimizar las diferencias entre operaciones, para que los usuarios de estas clases no necesiten saber acerca de las particularidades de cada operación. Para cumplir con este objetivo, la superclase **Operation** debiera implementar el máximo de métodos que proveerán luego las distintas subclases. Así, la interfaz resulta más completa, a pesar que haya métodos no utilizados para algunas subclases. La clase **Operation** proveerá un comportamiento por defecto para cada método, y las subclases redefinirán los métodos necesarios para su uso específico.

Sin embargo, existe otro principio de diseño orientado a objetos que sugiere definir solamente los métodos que correspondan a un denominador común para todas las subclases que se definan. Si se maximiza la interfaz de **Operation**, se daría el caso de que un método no tenga sentido en alguna de sus subclases, y por lo tanto se estaría violando este principio.

Un ejemplo concreto son los métodos de administración de hijos, que se estudiarán como caso aparte más adelante. Otro ejemplo son los métodos `playOne()`, `advanceOne()` [1] y similares, que no tienen sentido en general para operaciones básicas, pero sí son útiles para las subclases de **Animation**. Como estas clases están directamente relacionadas con la reproducción de animaciones, se consideró mejor incluirlas en la interfaz de **Operation**, ya que además es trivial encontrar un sentido en cada operación para estos métodos. Por ello, en la mayoría de las operaciones, el método `playOne()` va a tener exactamente el mismo efecto que el `play()`.

[1] Estos métodos y su uso son explicados en el capítulo 6

5.4.4 Declarar métodos de administración de hijos

Este es un caso particular del punto anterior. Es obvio que en las operaciones básicas no tiene sentido implementar métodos que sirvan para administrar los hijos (agregar, quitar y listar). Existe un compromiso entre robustez y transparencia al decidir si implementar estos métodos en la clase **Operation** y hacerlos existir en las operaciones básicas o solamente proveerlos en las clases *compositoras*.

1. si se definen estos métodos en la raíz (en **Operation**), se tiene transparencia ya que se pueden tratar a todos los componentes por igual. Sin embargo, esta implementación resta robustez porque pueden existir usuarios que traten de hacer cosas sin sentido como agregar o quitar hijos a operaciones básicas.
2. por otro lado, si se definen los métodos en las clases *compositoras*, se tiene robustez porque cualquier intento de agregar o quitar hijos a operaciones básicas podrá ser detectado, pero se pierde la transparencia de tratar igual a todas las subclases de **Operation**.

En el caso de esta implementación, el énfasis del diseño radica en que pueda existir una biblioteca de animaciones, y que el reproducir estas animaciones sea fácil. Para la creación y manipulación de estas animaciones, se proveerá un programa como parte del paquete que podrá encargarse de manejar la falta de transparencia que aparece al definir los métodos para administrar hijos solamente en las clases *compositoras*. Cualquier otro usuario usará estas clases en general solamente para reproducir o componer las animaciones.

Es por esta razón que se implementaron los métodos de administración de hijos en la clase **Animation**, y por lo tanto existen solamente en las clases *compositoras*.

5.4.5 ¿Debiera implementarse un listado de operaciones en Operation?

Si existen muy pocos hijos en una estructura de **Animation**, puede ser provechoso usar una lista de operaciones en **Operation**, que existirá también en cada operación básica. De lo contrario, el espacio gastado por mantener una lista en cada hoja es excesivo.

En este caso, como los métodos de administración de los hijos están ya en las clases *compositoras* y no en las operaciones básicas, es consistente definir esta lista en la clase **Animation**.

5.4.6 Orden de los hijos

El diseño de **Sequential** compone operaciones que serán ejecutadas una después de la otra, y claramente necesita un orden predeterminado de los hijos. Así, un recorrido correcto de un árbol que contenga solamente *compositores* del tipo **Sequential** será en preorden. Sin embargo, en el caso de la clase compositora **Parallel**, el orden no importa, puesto que todos sus hijos se ejecutan al mismo tiempo.

Como a la clase **Parallel** no le afecta que exista un orden, y puede ignorarlo si existe, se han definido los métodos de manera que mantengan el orden de los hijos. Además, se proveen distintos métodos para agregar hijos en el orden deseado, y para manejar el orden de los mismos.

5.4.7 Caching para aumentar el performance

En caso de necesitar realizar frecuentes búsquedas o accesos a través de composiciones complejas, puede resultar útil mantener información sobre los hijos en un cache. Este cache deberá ser invalidado al cambiar la información sobre los hijos, lo cual agrega cierta complejidad al código.

En el diseño de `anim.operations` no es necesario realizar este tipo de caches, puesto que en general las operaciones que se realizan son recorridos en preorden, ya sea de un sub-árbol completo o de un hijo a la vez.

5.4.8 Determinar la mejor estructura para almacenar componentes

Aunque la estructura intuitiva es la de un árbol, es necesario considerar si existen mejores alternativas. Por ejemplo, una lista enlazada u otra estructura puede mejorar el rendimiento de los programas que usen estas clases. Sin embargo, la estructura de árbol es lo suficientemente eficiente, y un cambio dificultaría demasiado el manejo de esta estructura. Por ello, se decidió implementar una estructura de árbol.

5.5 Conclusiones y asuntos pendientes

Con este patrón de diseño, se crea una jerarquía de clases que consiste en objetos primitivos (las operaciones básicas) y *compositores* (**Sequential** y **Parallel**) que componen nuevas operaciones a partir de operaciones básicas. En cualquier parte del código donde un programa espera encontrar un objeto de operaciones básicas, puede tomar de igual manera un *compositor* complejo en forma transparente. Por lo tanto, un programa que usa estas clases se mantiene simple, puesto que no necesita diferenciar objetos ni tratarlos distinto.

Se facilita la creación de nuevos *compositores*, como es el caso de la clase **Parallel**, que fue creada en forma simple y rápida. En cualquier otro diseño, hubiese sido bastante complejo integrar una clase que agrupa operaciones como lo hace esta clase.

Capítulo 6

Implementación de `anim.operations`

En la implementación del paquete `anim.operations` se crearon las clases abstractas **Operation** y **Animation**, usadas para dos fines. El primero es establecer una interfaz única para accesar sus subclases, y el segundo es incorporar métodos comunes para reutilizar el código en sus subclases. Existen por lo tanto dos APIs de importancia: una definida por **Operation** y otra por **Animation**. En base a esto, se crearon algunas operaciones básicas que permitieron probar y validar el diseño.

Dentro de los objetivos de este paquete [1] está que las operaciones sean interruptibles. La clase **Actor**, descrita en el capítulo 3 facilita proveer dicha facultad, pero no es deseable que el usuario de `anim.operations` necesite estar al tanto de dicha clase. Es por ello que se ha implementado la clase **ExecutionActor** como subclase de **Actor**, y la clase **Operation** provee una interfaz simple para acceder a **ExecutionActor**, encapsulando así el uso de la clase **Actor**.

[1] Ver capítulo 5 para los objetivos del paquete

6.1 API de Operation

La interfaz de la clase **Operation**, como ya se señaló en el capítulo 5, deberá incluir al menos los métodos básicos que permitan reproducir y editar la operación. La manipulación de las clases ***compositoras*** no se incluye dentro de la interfaz, ya que esos métodos sólo tienen sentido para subclases de **Animation**, y es allí donde se definen.

Los métodos de reproducción que presenta la clase **Operation** son los siguientes, que están definidos para toda operación, tanto básicas como compuestas:

- `play()`: ejecuta la operación, tomando el tiempo de simulación programado.
- `stop()`: interrumpe la ejecución de la operación y vuelve la operación al estado inicial.
- `pause()`: interrumpe la ejecución, dejando la operación en un estado intermedio, pudiendo ser retomada mediante `play()`, `playRev()`, `playOne()` o `playRevOne()`.
- `advance()`: ejecuta la operación, pero tomando el mínimo tiempo posible.
- `back()`: retrocede la operación al estado inicial.
- `playRev()`: ejecuta la operación en sentido contrario, es decir, desde el estado final hasta el estado inicial.
- `playOne()`: tiene sentido solamente para subclases de **Animation**, y ejecuta una sola operación.
- `advanceOne()`: avanza una sola operación.
- `backOne()`: retrocede una sola operación.
- `playRevOne()`: ejecuta en sentido contrario una sola operación.

Para la edición de las operaciones, se han definido los siguientes métodos, que aún resta por implementar en las subclases, ya que Este método está fuertemente ligado a la interfaz que se usará para manipular las operaciones, y por ello se ha implementado. En un ambiente gráfico, lo más probable es que se use un **Actor** que despliegue objetos de la clase **Glyph**, y que el usuario puede modificar gráficamente usando el mecanismo de *drag and drop*, al estilo del editor descrito en el capítulo 4. Los métodos mencionados son:

- `showCPoints()`: es el método encargado de desplegar los puntos de control y permitir al usuario modificarlos.
- `hideCPoints()`: este método termina el despliegue de los puntos de control y guarda las modificaciones realizadas.

6.2 API de Animation

La clase **Animation** es una subclase de **Operation**, que agrupa todas las clases *compositoras*. Por lo tanto, hereda todos los métodos de **Operation**, y se le agregan los relacionados con el manejo de hijos. También en esta clase se define la lista de los hijos, en una variable de tipo **List** llamada `operationList`. Por el momento, existen dos clases *compositoras*: **Sequential** y **Parallel**. Es deseable que tengan una interfaz común para facilitar el uso y la manipulación de animaciones creadas con el ambiente `anim`.

- `void add(Operation op)`: agrega la **Operation** `op` al final de la lista.
- `void put(Operation op)`: idéntica a `add(Operation op)`.
- `void push(Operation op)`: agrega `op` al comienzo de la lista.
- `void remove(Operation op)`: quita `op` de la lista de operaciones.
- `Operation set(int i, Operation op)`: reemplaza la operación en la posición `i` de la lista por `op`. Retorna la operación reemplazada o null, según corresponda.
- `List getList()`: retorna una nueva lista de operaciones.
- `ListIterator listIterator()`: retorna un iterador de la lista de operaciones.

6.3 ExecutionActor

El uso del paquete `anim.actors` permitió implementar las operaciones sin necesidad de preocuparse de secciones críticas, e introducir la interruptibilidad de las animaciones. Esto último, a través del envío de un mensaje desde el thread del kit gráfico *AWT* de *Java* a los actores correspondientes. Sin embargo, no es necesario que el usuario del paquete `anim.operations` necesite conocer ni saber de la existencia de los *actores*. Para ello existen dos razones fundamentales:

1. sería necesario que el usuario se familiarice con los *actores*, siendo que solamente pretende usar operaciones existentes y no crear nuevas ni aprovechar la flexibilidad o poder que entregan estas clases.

2. como el sistema aún no es robusto, podría ser que las operaciones dejen de funcionar por un error de programación del usuario. Este tipo de errores, como por ejemplo *deadlocks*, son difíciles de depurar, y esto crearía una dificultad innecesaria en el uso de la biblioteca.

Por ello, se decidió presentar una interfaz simple al usuario, que encapsule el uso de los *actores*. Las clases que implementan las operaciones son subclases de **ExecutionActor**, y las subclases de **Operation** sirven meramente de interfaz para lograr el encapsulamiento. Así, por cada operación definida, incluyendo las operaciones **Animation**, **Sequential** y **Parallel**, existe una subclase de **ExecutionActor** que implementa realmente la operación.

6.4 Serialización

Un aspecto importante de una animación es la capacidad de poder almacenar y luego volver a construir en memoria los objetos que la componen. Esto se logra en *Java* implementando la interfaz **Serializable**. Una vez que se definido como *serializable* la clase **Operation**, es posible almacenar una operación en un **Stream**, sin importar lo compleja que ésta sea. Para el caso de operaciones que son subclases de **Animation**, se guardará todo el árbol en un **Stream**. Sin embargo, existen ciertas consideraciones que se deben tomar en cuenta para que estas operaciones funcionen correctamente:

1. *Información pierde sentido*: en ciertos casos, existe información (generalmente referencias a objetos) que pierde sentido al almacenar una operación en un **Stream**. Por

ejemplo, para la operación **Creation**, la variable `theater` que almacena el teatro dentro del cual se crea al **Glyph** no tiene sentido ser guardada.

En una implementación directa, se almacenaría el **Theater** referenciado como parte de la operación en el mismo **Stream**, pero esto no tiene sentido. El **Theater** en general contiene una o más operaciones en alguna lista, y guarda estas operaciones en un stream. Luego, algún **Theater** creará una operación a partir del **Stream**. Entonces, es responsabilidad del **Theater** que crea esta nueva copia de la operación asignar el correspondiente valor a la variable `theater` de la operación recién creada, mediante el método `setTheater(Theater t)` definido en **Operation**.

2. *Múltiples referencias a objetos*: se da el caso de que varias operaciones tengan referencia a la misma operación de tipo **Creation**. Es importante que los objetos se refieran al mismo objeto en memoria, cosa que no ocurre si se usa la implementación por defecto que ofrece *Java* para la serialización. Para garantizar que toda referencia a un mismo objeto de clase **Creation** dentro de una animación se reconstruya correctamente, se define un identificador *único* para cada objeto de tipo **Creation** en memoria dentro de un programa.

   Al escribir una operación a un **Stream**, se escribe entonces el identificador en vez de una copia serializada del objeto **Creation**. Para reconstruir una animación, se guarda el identificador de cada **Creation** leído en una tabla, asociado al nuevo objeto existente en memoria. Luego, al reconstruir una operación con referencia a una operación de tipo **Creation** por medio del identificador, se recupera la referencia del objeto reconstruido en memoria correspondiente a dicho identificador.

   El problema de este mecanismo es que no queda claro mediante la información de que se dispone de cuándo es posible eliminar la entrada creada en la tabla al leer una operación **Creation**. Para una versión definitiva del paquete `anim.operations` sería necesario solucionar este problema, para evitar que la tabla usada comience a ocupar excesiva memoria a medida que se crean operaciones a partir de streams.

3. *Información redundante (caches por ejemplo)*: Existe información contenida en los objetos que no es necesario o incluso contraproducente almacenar. Esta información simplemente no se incluye en el **Stream** generado, y al reconstruir el objeto se asigna un valor inicial definido.

6.5 Asuntos pendientes

6.5.1 Consistencia de animaciones

En este trabajo no se ha tratado un tema importante, referente a modificaciones hechas sobre las operaciones. En particular, pueden existir modificaciones a operaciones que dañen la consistencia de una animación. Por ejemplo, sería crítico que se elimine una operación de tipo **Creation** que está siendo referenciada por otras operaciones más adelante en la animación.

6.5.2 Aspectos relacionados con animaciones

Al hacer una animación, una diferencia esencial con las películas es que las imágenes son demasiado nítidas. Eso tiene como efecto que el movimiento se vea saltado, a pesar de contar con la suficiente cantidad de cuadros por segundo. Ej: películas grabadas con muy alta velocidad. Es un aspecto interesante, aunque no se trate en esta memoria.

Sería interesante estudiar mecanismos que logren aminorar este tipo de efectos, por ejemplo mediante la deformación o uso de efectos gráficos sobre los **Glyph** para dar la ilusión de movimiento.

Capítulo 7

Uso de la biblioteca `anim.operations`

En este capítulo se describirá la biblioteca contenida en el paquete `anim.operations` que fue desarrollada durante este trabajo. La biblioteca permite generar animaciones componiendo operaciones básicas. Dentro de las operaciones básicas existentes, encontramos las siguientes:

- **Creation**: es la operación que crea un **Glyph**, lo despliega en pantalla y guarda su referencia.
- **Movement**: mueve un **Glyph** entre dos puntos en un determinado período de tiempo.
- **Deletion**: hace desaparecer un **Glyph** después de haber sido desplegado por una operación **Creation**.
- **GrowingArrow**: modifica la dirección y longitud de una flecha dentro de un determinado período de tiempo.
- **ColorChange**: modifica el color de fondo y primer plano.
- **Pause**: introduce una pausa por un determinado período de tiempo.

Para poder componer las operaciones básicas, existen operaciones, llamadas *compositoras*. La clase base es **Animation**, y por el momento existen las subclases **Sequential** y **Parallel**. La primera ejecuta una serie de operaciones en secuencia, y la segunda ejecuta dos operaciones en paralelo y finaliza cuando la última haya terminado su ejecución. La mejor forma de presentar el uso de la biblioteca es a través de un ejemplo, por lo tanto se incluye a continuación un método que crea una animación a partir de operaciones básicas:

```
public static Operation HolaMundo(GPoint comienzo, GPoint fin) {
  Animation animacion = new Sequential();
  Creation texto = new Creation(new GText("Hola, Mundo"), null, comienzo);
  animacion.add(texto);
  animacion.add(new Movement(texto, 5, comienzo, fin));
  return animacion;
}
```

Este código crea una animación que consiste en desplegar el string "Hola, Mundo", y moverlo desde la posición `comienzo` hasta la posición `fin` en un intervalo de 5 segundos. Una vez construida esta animación, es posible ocuparla dentro de otra como si fuera una operación básica. Por ejemplo, el siguiente código crea una animación en la cual se despliegan dos strings "Hola, Mundo", y se mueven en paralelo:

```
public static Operation HolaParalelo() {
  GPoint p1 = new GPoint(100,0);
  GPoint p2 = new GPoint(50,100);
  Animation paralelo = new Parallel();
  paralelo.add(HolaMundo(p1,p2));
  paralelo.add(HolaMundo(p2,p1));
  return paralelo;
}
```

El código completo de dos ejemplos se pueden encontrar en los anexos de este trabajo. Se incluye una implementación completa del ejemplo *Hola Mundo*, que escribe la operación creada en un archivo (en la página 56), y otro ejemplo que lee un archivo conteniendo un objeto de clase **Operation** y permite ejecutarlo (en la página 58).

Como parte del trabajo, se desarrolló una clase llamada **test.CreaAnim**, que genera una animación para luego escribirla en un archivo. Esto se hace usando la propiedad de

Serializable que tiene la clase **Operation**. También se creó una clase que permite reproducir operaciones guardadas en un archivo, llamada **test.Test**. Esta clase está basada en la clase **Editor** descrita en el capítulo 4, pero no tiene la capacidad de crear ni seleccionar o mover los **Glyph**. Su función es simplemente leer un archivo que contiene una operación, y permitir al usuario ejecutarla presentando la API de **Operation** por medio de botones.

Capítulo 8

Discusión y Conclusiones

Como resultado del trabajo, se cuenta con un sistema que puede crear animaciones y reproducirlas. Es necesario evaluar este sistema mediante su uso, para luego implementar una versión definitiva. Además, con el diseño de las operaciones, se cuenta con una base para continuar el desarrollo de los restantes componentes de la herramienta.

8.1 Evaluación de `anim.actors`

Durante el proyecto, se hizo uso del paquete `anim.actors`, que forma parte de la herramienta que está siendo desarrollada. Durante éste uso se puso a prueba su diseño, y es posible realizar una evaluación parcial.

Gracias a `anim.actors` se facilitó la programación del paquete `anim.operations`, ya que termina con la necesidad de identificar secciones críticas de los programas, al garantizar que los threads de Java se ejecuten en forma ***non-preemptive***. Este hecho, junto con la posibilidad de sincronización y la abstracción que permite el envío de mensajes, facilitó enormemente la tarea de programar las clases que forman este trabajo.

A pesar de que en este caso particular no se aprovechó de la mejor manera el uso de los mensajes tipados, resulta útil considerar esta capacidad para integrar operaciones complejas dentro de animaciones. En estas operaciones complejas, pueden existir una serie de actores interactuando entre ellos, sincronizándose a través de mensajes. Con este sistema, es posible crear bibliotecas de operaciones específicas para determinado tipo de animación, aprovechando todo el potencial del lenguaje de programación *Java*, pero encapsulando esta operación en un esquema de fácil uso y manipulación. Como el paquete `anim.actors` fue desarrollado justamente para crear simulaciones, es posible incluir simulaciones existentes en forma rápida en animaciones, permitiendo tanto al autor de la animación como al usuario que la reproduce modificar ciertos parámetros para experimentar con la simulación.

Sin embargo, la parte que aún necesita trabajo para mejorar su fragilidad es la interacción entre el usuario y las animaciones. Esto se debe a que el usuario utiliza una interfaz gráfica, que está basada en eventos, mientras que las animaciones usan mensajes para comunicarse. Al mezclar estos dos métodos de programación, se está agregando cierta fragilidad a la herramienta, puesto que es fácil que el programador de una operación pueda afectar el funcionamiento de toda la animación al no manejar correctamente alguna interacción con el usuario.

8.2 Aspectos pendientes

8.2.1 Traducción de eventos a mensajes tipados

Es necesario crear una capa intermedia que presente los eventos como mensajes a los *actores*, ya que esa es la forma en que reciben la información. Esta capa necesita ser lo suficientemente robusta como para evitar que un error de programación de una operación afecte el correcto funcionamiento de la herramienta, y de las demás operaciones. Un error como por ejemplo dejar bloqueado un *thread* o perder un evento, llevan actualmente a un posible congelamiento de la herramienta, dando una sensación de fragilidad.

Por lo general, es deseable que las operaciones interactúen con la interfaz gráfica presentada al usuario. Esto permite por ejemplo construir la forma de los **Glyph** a usar en una animación al momento de reproducir la misma. También se puede alterar el comportamiento de una animación mediante la intervención del usuario. Gracias a esta interacción se permitiría la experimentación de la cual se habló en el capítulo 2. Sin esta cualidad, el efecto de la herramienta podría perder parte de su ventaja frente a otros sistemas. También es necesario contar con esta interacción para permitir la edición gráfica de las operaciones, a través de *drag and drop* y otros mecanismos propios de un ambiente gráfico.

En el diseño de una capa así es necesario considerar cómo adecuar la API de la mensajería de la clase **Actor** a la API de programación orientada a eventos que ofrece el kit gráfico de Java, AWT. Es necesario considerar que por ejemplo no tiene sentido encolar más de un evento de movimiento o arrastre del puntero. Pero hay otros eventos que sí es necesario considerar por separado, como por ejemplo dos *clicks* del puntero.

Una clase que adapte estas metodologías deberá considerar casos excepcionales, como por ejemplo el efecto de un mensaje perdido. Es necesario cuestionarse diversas condiciones, como por ejemplo:

- ¿Qué es lo que pasa si se reciben dos mensajes de MOUSEDOWN y ningún MOUSEUP? Es necesario poder manejar este tipo de situaciones, que aunque no tienen sentido, pueden darse. Es parte de la robustez exigida que el programa pueda manejar este tipo de situaciones.

- ¿Cuántos eventos de un tipo deberán enviarse a un **Actor** que se suscribe? En algunos casos, es natural enviar solamente un evento, para así evitar por ejemplo que se encolen varios mensajes para un mismo evento. Pero en otras ocasiones, es necesario no perder ningún evento.

- ¿Sirven los mensajes de distinto tipo según el evento, o es necesario agrupar todos los mensajes originados por evento en un mismo tipo? Si se usan distintos tipos, los mensajes podrían llegar en un orden distinto al FIFO, que es el que existe por definición en los sistemas orientados a eventos. Esto podría producir situaciones conflictivas, y difíciles de depurar.

8.2.2 Narración

La narración, a pesar de no ser compleja de implementar usando las bibliotecas existentes en Java, presenta una dificultad especial. Es necesario integrar la narración a las operaciones, y esto amerita una serie de consideraciones. La más compleja de ellas es la coordinación entre la narración (y sonido en general) y la presentación.

Es necesario considerar que el tiempo de simulación, según el cual se rige la presentación que se reproduce, no necesariamente es igual al tiempo real, según se explica en el la parte 3.3 de este documento. Por otro lado, la narración debe ser reproducida en tiempo real, porque de lo contrario no es percibida correctamente y pierde el sentido. Es por esto que la animación y la narración tienen una tendencia normal a de-sincronizarse, efecto que debe controlarse de alguna manera.

- *duración de la narración*: en general se intuye que la narración tiene una duración definida. Sin embargo, existen ocasiones en las que el sonido que acompaña una animación es más bien de fondo. En ese caso, el sonido debiera durar lo mismo que la animación, repitiéndose de ser necesario.
- *sincronización de eventos dentro de la narración*: puede resultar útil tener alguna forma de sincronizar instantes de tiempo de la narración con estados definidos de la animación. Por ejemplo, se podría agregar un sonido cada vez que dos elementos gráficos pasan cerca el uno del otro. También es útil poder coordinar una voz que explica un fenómeno con la simulación de ese fenómeno.
- *representación gráfica de la coordinación*: para incluir la narración dentro de la herramienta, que es gráfica, es necesario definir un esquema para presentar la coordinación en forma gráfica.

8.2.3 Editor gráfico

Para construir el editor gráfico, ya se han agregado los métodos necesarios para el manejo de los **Glyph**. Con ello, la construcción de un editor para manipular animaciones puede usar el editor de **Glyph** desarrollado en este trabajo como base. Es deseable usar una capa intermedia, como se explicó en el punto 8.2.1, para traducir eventos a mensajes, ya que el editor de este trabajo no cuenta con dicha capa y resulta necesaria para un correcto funcionamiento del programa. También es necesario contar con un diseño del esquema que se usará para la narración. Esto, ya que la narración formará parte de las presentaciones y debiera ser posible tanto la edición de la narración a través de este editor como la coordinación entre la narración y las operaciones.

Como parte del editor gráfico se considera también una funcionalidad aún no implementada en las operaciones. Esta funcionalidad es la edición de los parámetros de las operaciones, a través de menúes y despliegue de los puntos de control. Para ello, es necesario que cada operación permita desplegar los puntos de control en el editor, y estos puntos de control debieran poder modificarse mediante *drag and drop* (arrastre por medio del puntero) o a través de un menú.

El Editor gráfico debiera ser el último elemento en construirse, para que pueda integrar todos los componentes de la herramienta en forma armónica en un solo elemento simple de usar. Deberá considerar todos los aspectos que surjan al desarrollar un tipo de narración compatible con las animaciones, y presentar al usuario una interfaz simple y entendible.

Para presentar el esquema de árbol que presentan las operaciones definidas en este trabajo, se ha pensado que una buena forma de presentar gráficamente las operaciones es usando un esquema como el explorador de archivos de *Windows 95*. Estos esquemas permiten visualizar la estructura del árbol, y a su vez determinar el nivel de detalle con el que se ve cada rama. Es posible comprimir todo un (sub-)árbol a un solo nivel, con lo cual el usuario puede concentrar su atención a la parte específica de la animación que necesite revisar.

Sin embargo, en los sistemas de archivo se tiene un solo tipo de nodos. Para el caso de las operaciones, existen al menos dos tipos de nodos: las animaciones secuenciales y las paralelas. Además, es necesario incluir la narración, tomando en cuenta cualquier consideración especial para ello, según se determine durante el desarrollo del componente de narración.

Bibliografía

[1] Varios autores. **JavaTM 2 Platform, Standard Edition, v1.2.2 API Specification**, 1999. disponible a través de `http://java.sun.com/products/jdk/1.2/docs/api/index.html`.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. **Design Patterns**. Addison-Wesley, 1994. Elements of Reusable Object-Oriented Software.

[3] Alan Kay. **Active Essays**, 1996. disponible a través de `http://lcs.www.media.mit.edu/groups/el/projects/emergence/active-essay.html`.

[4] Pablo José Pozo Palma. **Herramientas de simulación basadas en Threads de Java**. Memoria para optar al título de Ingeniero Civil en Computación, Universidad de Chile, 1998.

[5] Mitchel Resnick and Brian Silverman. **Going in Circles**. disponible a través de `http://el.www.media.mit.edu/groups/el/projects/circles/`.

[6] Mitchel Resnick and Brian Silverman. **Exploring Emergence**, 1996. disponible a través de `http://el.www.media.mit.edu/groups/el/projects/emergence/`.

[7] Luis Igor Ávila Prado. **Animation2D**. Memoria para optar al título de Ingeniero Civil en Computación, Universidad de Chile, 1999.

Apéndice A

Definiciones

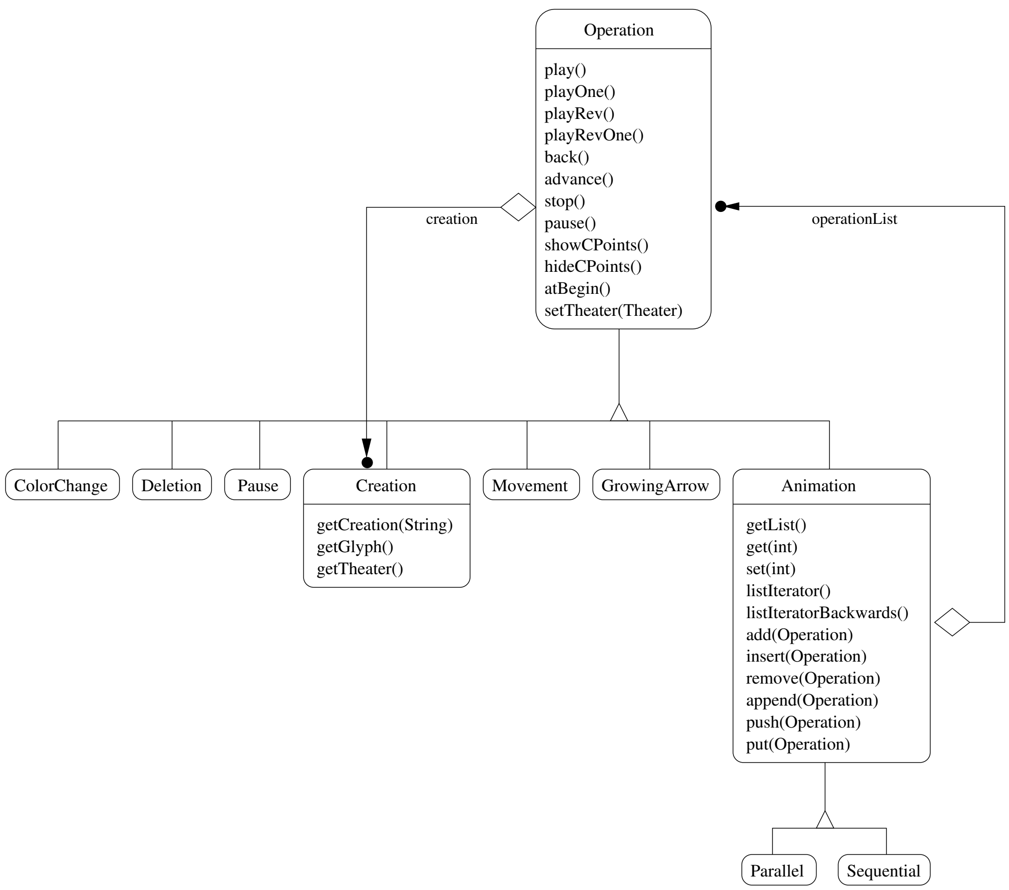
- ***interactividad, interactivo***: es la respuesta por medio del lenguaje al lenguaje.
- ***puntos de control***: son variables que determinan el cómo va a actuar una operación o cómo va a dibujarse un Glyph. No necesariamente corresponden a coordenadas, el sentido común de "punto", sino pueden ser variables como tiempo, distancia (ej: radio de un círculo).
- ***WYSIWYG***: "What You See Is What You Get"
- ***double buffering***: proceso de generar primero una imagen en memoria, antes de mostrarla en pantalla. Permite evitar el parpadeo que se produce al tener una demora en el dibujado de un gráfico, aunque se ocupe un poco más de memoria y el tiempo sea marginalmente superior, al tener más operaciones asociadas.
- ***applet***: programa en Java, que corre bajo ciertas restricciones. La clase que implementa este programa también se llama Applet. Este esquema se implementa en Browsers o navegadores, que permiten así correr programas en páginas web, sin comprometer la seguridad.
- ***bound box***: elemento gráfico, generalmente un rectángulo que incluye completamente otro elemento gráfico, como un Glyph en el caso de este trabajo. Para efectos del paquete anim, el bound box es un objeto que implementa la interfaz java.awt.Shape. Con esto, se mejora la calidad de la aproximación, asumiendo un costo en eficiencia.
- ***actor(es)***: dentro de este documento, se refiere como ***actores*** a las subclases de la clase `anim.actors.`**Actor**

- *`compositores, clases compositoras`*: son las clases del paquete `anim.operations` que componen operaciones. Están agrupadas bajo la superclase **Animation**, y también se denominan animaciones.

- *`Serializable`*: en *Java*, existe una interfaz llamada **Serializable**. Cualquier clase que implemente esta interfaz adquiere la capacidad de que sus objetos pueden ser escritos en forma serial a un **Stream**, pudiendo escribir dicho **Stream** a un archivo o enviarlo a través de una conexión de red. Luego, es posible reconstruir el estado de ese objeto a partir del **Stream**.

Apéndice B

Diagrama de clases de Operation



Apéndice C

Ejemplos Completos

C.1 Hola Mundo

El código siguiente crea una animación simple, la cual es finalmente guardada en un archivo para su posterior reproducción.

```
package test;

import anim.operations.*;
import anim.actors.*;
import anim.glyphs.*;
import java.io.*;

public class HolaMundo {
  public static Operation HolaMundo(GPoint comienzo, GPoint fin) {
    Animation animacion = new Sequential();
    Creation texto = new Creation(new GText("Hola, Mundo"), null, comienzo);
    animacion.add(texto);
    animacion.add(new Movement(texto, 5, comienzo, fin));
```

```
    return animacion;
  }
  public static Operation HolaParalelo() {
    GPoint p1 = new GPoint(100,20);
    GPoint p2 = new GPoint(50,100);
    Animation paralelo = new Parallel();
    paralelo.add(HolaMundo(p1,p2));
    paralelo.add(HolaMundo(p2,p1));
    return paralelo;
  }
  public static void main(String argv[]) {
    String archivo = null;
    if (argv.length < 1) {
      System.out.println("Uso: CreaAnim <archivo_animacion>");
      System.exit(0);
    } else
      archivo = argv[0];

    Operation animacion = HolaParalelo();

    System.out.print("Writing file... ");
    try {
      FileOutputStream ostream = new FileOutputStream(archivo);
      ObjectOutputStream p = new ObjectOutputStream(ostream);
      p.writeObject(animacion);
    } catch (FileNotFoundException fnf) {
      System.out.print("\n\nFile not found: " + archivo);
    } catch (IOException io) {
      System.out.print("\n\n" + io);
    } finally {
      System.out.println();
    }
    System.exit(0);
  }
}
```

C.2 Ejecución de Operation desde archivo

Este ejemplo lee una animación desde un archivo, la cual podrá ser reproducida usando la API standard, por medio de botones en un ambiente gráfico.

```
package test;

import anim.operations.*;
import anim.actors.*;
import anim.glyphs.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class Test {

  Frame ventana;
  Label statusbar;
  Theater testTheater;
  ControlPanel miControlPanel;

  static Symbol PLAY = Symbol.make("PLAY");
  static Symbol PLAYREV = Symbol.make("PLAYREV");
  static Symbol ADVANCE = Symbol.make("ADVANCE");
  static Symbol BACK = Symbol.make("BACK");
  static Symbol BACKONE = Symbol.make("BACKONE");
  static Symbol ADVANCEONE = Symbol.make("ADVANCEONE");
  static Symbol PLAYONE = Symbol.make("PLAYONE");
  static Symbol PLAYREVONE = Symbol.make("PLAYREVONE");
  Operation animacion;

  public Test(String archivo) {
    System.out.print("Leyendo archivo... ");
    try {
      FileInputStream istream = new FileInputStream(archivo);
      ObjectInputStream p = new ObjectInputStream(istream);
      animacion = (Operation)p.readObject();
```

```
    if (animacion == null) {
      System.out.println("animacion == null?");
      System.exit(1);
    } else {
      int size;
      if (animacion instanceof Animation) {
        java.util.List l = ((Animation)animacion).getList();
        size = l.size();
      } else
        size = 1;
        System.out.println("Leido:");
        System.out.println(size + " operaciones");
    }
  } catch (FileNotFoundException fnf) {
    System.out.print("\n\nFile not found: " + archivo);
  } catch (IOException io) {
    System.out.print("\n\n" + io);
  } catch (ClassNotFoundException cnf) {
    System.out.print("\n\n" + cnf.getMessage());
  } finally {
  }
  testTheater = new TestTheater();
  ventana = new Frame("Operation Testing");
  ventana.setSize(new Dimension(800,600));
  ventana.addWindowListener(new miWindowListener());
  statusbar = new Label();
  Panel miPanel = new Panel(new BorderLayout());
  miPanel.add(testTheater.canvas(), "Center");
  ventana.add(miPanel);
  miControlPanel = new ControlPanel(testTheater);
  miPanel.add(miControlPanel, "North");
  miPanel.add(statusbar, "South");
  miPanel.setVisible(true);
  ventana.show();
}

public static void main(String[] argv) {
  if (argv.length < 1) {
    System.out.println("Uso: Test <archivo_animacion>");
    System.exit(0);
  }
  Test miTest = new Test(argv[0]);
```

```
}

public class miWindowListener extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}

class ControlPanel extends Panel {

  Button BotonSalir;
  Button BotonPlay, BotonPlayRev, BotonAdvance, BotonBack, BotonStop,
      BotonPause, BotonPlayOne, BotonPlayRevOne, BotonAdvanceOne,
      BotonBackOne;
  Theater testTheater;

  ControlPanel(Theater t) {
    testTheater = t;
    if (testTheater == null) {
      throw new NullPointerException();
    }
    BotonBack = new Button("Back");
    BotonBack.addActionListener(new MiActionListener(BACK));
    this.add(BotonBack);
    BotonPlayRev = new Button("PlayRev");
    BotonPlayRev.addActionListener(new MiActionListener(PLAYREV));
    this.add(BotonPlayRev);
    BotonPlay = new Button("Play");
    BotonPlay.addActionListener(new MiActionListener(PLAY));
    this.add(BotonPlay);
    BotonAdvance = new Button("Advance");
    BotonAdvance.addActionListener(new MiActionListener(ADVANCE));
    this.add(BotonAdvance);
    BotonPause = new Button("Pause");
    BotonPause.addActionListener(new PauseListener());
    this.add(BotonPause);
    BotonStop = new Button("Stop");
    BotonStop.addActionListener(new StopListener());
    this.add(BotonStop);
    BotonSalir = new Button("Salir");
    BotonSalir.addActionListener(new SalirActionListener());
    this.add(BotonSalir);
```

```
    BotonPlayOne = new Button("PlayOne");
    BotonPlayOne.addActionListener(new MiActionListener(PLAYONE));
    this.add(BotonPlayOne);
    BotonPlayRevOne = new Button("PlayRevOne");
    BotonPlayRevOne.addActionListener(new MiActionListener(PLAYREVONE));
    this.add(BotonPlayRevOne);
    BotonAdvanceOne = new Button("AdvanceOne");
    BotonAdvanceOne.addActionListener(new MiActionListener(ADVANCEONE));
    this.add(BotonAdvanceOne);
    BotonBackOne = new Button("BackOne");
    BotonBackOne.addActionListener(new MiActionListener(BACKONE));
    this.add(BotonBackOne);
  }
}

class StopListener implements ActionListener {
  public void actionPerformed(java.awt.event.ActionEvent e) {
    System.out.println("Stopping animation...");
    ((TestTheater)testTheater).stopAnimation();
  }
}

class PauseListener implements ActionListener {
  public void actionPerformed(java.awt.event.ActionEvent e) {
    //System.out.println("Pausing animation...");
    ((TestTheater)testTheater).pauseAnimation();
  }
}

class MiActionListener implements ActionListener {
  Symbol mi_simbolo;
  MiActionListener(Symbol s) {
    super();
    mi_simbolo = s;
  }

  public void actionPerformed(java.awt.event.ActionEvent e) {
    testTheater.post(new Msg(mi_simbolo));
  }
}

class SalirActionListener implements java.awt.event.ActionListener {
```

```
  public void actionPerformed(java.awt.event.ActionEvent e) {
    if (e.getActionCommand().equals("Salir")) {
      System.exit(0);
    }
  }
}

class TestTheater extends Theater {

  public TestTheater() {
    super();
  }
  public void body() {
    System.out.println("TestTheater.body()");
    enable(PLAY);
    enable(PLAYREV);
    enable(ADVANCE);
    enable(BACK);
    enable(ADVANCEONE);
    enable(BACKONE);
    enable(PLAYONE);
    enable(PLAYREVONE);
    animacion.setTheater(this);
    for (;;) {
      Msg msg = accept();
      if (msg.getSym() == PLAY) {
        animacion.play();
      } else if (msg.getSym() == PLAYREV) {
        animacion.playRev();
      } else if (msg.getSym() == ADVANCE) {
        animacion.advance();
      } else if (msg.getSym() == BACK) {
        animacion.back();
      } else if (msg.getSym() == PLAYONE) {
        animacion.playOne();
      } else if (msg.getSym() == PLAYREVONE) {
        animacion.playRevOne();
      } else if (msg.getSym() == ADVANCEONE) {
        animacion.advanceOne();
      } else if (msg.getSym() == BACKONE) {
        animacion.backOne();
      }
```

```
      }
    }
    public void stopAnimation() {
      animacion.stop();
    }
    public void pauseAnimation() {
      animacion.pause();
    }
  }
}
```