UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Modelo de Gestos considerando Contexto
Autores: Franco Agustín Musi Gentile
Director: Dra. Cecilia Challiol
Codirector: Dra. Silvia Gordillo
Carrera: Licenciatura en Informática

Resumen

En los últimos años hubo un gran avance en las tecnologías referentes a control de movimiento. Los más conocidos, si bien fueron pensados para el ámbito de los videojuegos, tienen sus aplicaciones en otros escenarios. Sin embargo, dentro de los proyectos que se han estudiado, ninguno muestra una separación clara entre el dominio o lógica de la aplicación y la recuperación de los datos a través de los sensores. Es por esto que se ve la falta de un modelo general capaz de integrar diferentes funcionalidades o medios de recuperación de datos y que, a su vez, los mismos se encuentren desacoplados, es decir, que la aplicación no sea dependiente de los sensores que utiliza.

En esta tesis se propone un modelo general de gestos desacoplado de su mecanismo de sensado. A su vez, este modelo se definirá de manera independiente a las acciones que desencadenan dichos gestos. Para lograr esto se utilizará el concepto de sensibilidad al contexto. Finalmente desarrollaremos un prototipo para probar el funcionamiento del modelo propuesto y el estudio del Kinect como sensor de movimiento particular.

Palabras Claves

Modelo de Gestos, Modelo de Contexto, Sensibilidad al Contexto, Sensores de Movimiento, Kinect.

Trabajos Realizados

En primera instancia se hizo un estudio de las aplicaciones que hacen uso del Kinect y de aplicaciones sensibles al contexto. En segundo lugar se propuso un primer modelo de gestos general y una capa de contexto independiente del dominio. Luego ambos modelos fueron combinados para obtener un modelo integrado final.
Se desarrolló un prototipo para poder probar el funcionamiento y viabilidad del modelo como así también detectar problemáticas en el uso de un sensor como el Kinect.

Conclusiones

El modelo de gestos general integrado con la capa de contexto está planteado de una manera flexible para ser extendido tanto sea con acciones puntuales, gestos particulares o aspectos específicos de contexto.
Este modelo integrado está planteado de manera independiente de los sensores de movimiento. Si bien fue probado usando el Kinect como sensor, el mismo puede ser utilizado con otros sensores de movimiento disponibles en el mercado.
El prototipo permitió descubrir problemáticas del uso del Kinect no detalladas en la documentación teórica.

Trabajos Futuros

- Extender el modelo con gestos y acciones que sean usados frecuentemente en diferentes dominios.
- Extender la capa de contexto para proveer comportamiento más específico.
- Analizar y diseñar gestos que requieran la interacción con múltiples sensores de movimiento.
- Analizar e implementar la captura de movimientos desde diferentes tipos de sensores.
- Implementar prototipos para detectar aspectos no modelados aún y, con esto, enriquecer la funcionalidad del modelo.

Fecha de la presentación: 12/11/2015

ÍNDICE

1. Introducción

1.1 Motivación

En los últimos diez años hubo un gran avance en las tecnologías referentes a control de movimiento. Los más conocidos son aquellos que fueron diseñados para videojuegos, cada una de las empresas más importantes del rubro lanzó su propia versión al mercado. Los primeros fueron la gente de Nintendo quienes propusieron el *Wiimote*, control que caracterizaba a su nueva consola *Wii* lanzada en el 2006. Dado su gran éxito en el mercado Sony y Microsoft comenzaron a desarrollar sus propios controles de movimiento para sus respectivas consolas. *PlayStation Move* es el control desarrollado por Sony presentado en la *E3* en el 2009, mientras que Microsoft presentó a *Kinect* en la misma expo bajo el nombre de *Project Natal*.

Todos estos controles de movimiento, si bien fueron pensados para el ámbito de los videojuegos, tienen sus aplicaciones en otros escenarios. El que más destaca de ellos es Kinect, dado que permite a los usuarios registrar sus movimientos sin la necesidad de un dispositivo en sus manos, sino que sus propios cuerpos sean todo lo necesario para interactuar con las aplicaciones que hacen uso de esta tecnología. Existen varios proyectos que hacen uso del Kinect por fuera de los videojuegos. Por ejemplo, en [Binbin Xu, 2014] se usa para desarrollar una aplicación de dibujo interactivo para niños pequeños, en [Bond and Curran, 2014] se utiliza para detectar artritis reumatoide en los usuarios, en [Gallo et al, 2011] se usa para controlar una base de datos de imágenes en un ambiente crítico, como es una sala de operaciones, sin que el usuario requiera ningún tipo de interacción física, mientras que en [Stowers et al, 2011] se usa para controlar el vuelo de un cuadricoptero.

Ninguno de estos proyectos mencionados muestra una separación clara entre el dominio o lógica de la aplicación y la recuperación de los datos a través del Kinect. Esto puede ser extendido a todo tipo de sensores; las aplicaciones no deberían ser totalmente dependientes de los dispositivos que se estén utilizando. Si en un futuro se pretende cambiar el mecanismo de sensado o surge una nueva tecnología que a los desarrolladores les interesa para su aplicación, es muy probable que para adaptarse a ese cambio sea necesario reescribir el código completo. A su vez, estas aplicaciones pueden verse como buenos ejemplos de aplicaciones sensibles al contexto, pero ninguno de los artículos menciona o determina claramente el contexto de su aplicación, lo que podría ser de gran ayuda para conseguir el desacoplamiento buscado.

Es por esto que se ve la falta de un modelo general capaz de integrar diferentes funcionalidades o medios de recuperación de datos y que, a su vez, los mismos se encuentren desacoplados. Finalizada esta tesis se espera contar con una aplicación reconocedora de gestos y desacoplada de su mecanismo de sensado en la cual cada usuario pueda tener un conjunto de gestos diferentes y poder configurar dichos gestos para realizar diferentes acciones dentro de la aplicación. A su vez, se espera contar con una capa de contexto capaz de brindar funcionalidades contextuales a diferentes aplicaciones y que la misma sea de un uso sencillo al programador.

1.2 Objetivo

La problemática que se busca resolver es contar con un modelo general de gestos desacoplado del mecanismo de sensado. Este modelo se definirá de manera independiente a las acciones que desencadenan dichos gestos. Es decir, se deberá tener la definición de los gestos y por otra parte las acciones que se pueden realizar cuando se detectan los mismos. Este modelo de gestos permitirá que cada usuario configure cómo cada gesto es interpretado, y qué acciones se dispararán cuando se detecta dicho gesto.

También se desarrollará una capa que permita a cualquier aplicación lograr el desacoplamiento requerido. Para esto se usará de base el modelo propuesto en [Fortier et al., 2007] el cual está centrado en el desarrollo de aplicaciones sensibles al contexto permitiendo un desacoplamiento del dominio de la aplicación de sus aspectos contextuales.

Se propondrá, en primera instancia, un modelo de gestos general. En segundo lugar, se presentará una capa de contexto capaz de desacoplar diferentes funcionalidades de una aplicación de manera sencilla y eficiente, dicha capa se presentará como un modelo en sí mismo. Luego, se integrarán ambos modelos para lograr tener una propuesta de una aplicación reconocedora de gestos y desacoplada de su mecanismo de sensado. Por último, se desarrollará un prototipo que implemente esta integración y plantee un ejemplo de una aplicación que pueda hacer uso del mismo.

1.3 Estructura de la Tesis

En el Capítulo 2 introduciremos al lector a los conceptos de contexto y aplicaciones sensibles al contexto. Mostraremos algunas clasificaciones sobre el contexto como también algunos modelos desarrollados por otros autores para implementar aplicaciones que hagan uso de este concepto. En la segunda parte de este mismo capítulo introduciremos el Kinect, su historia, de lo que es capaz y algunos usos que ciertos desarrolladores le dieron por fuera del ámbito de los videojuegos, que es el campo donde es más usado este sensor.

En el Capítulo 3 entraremos más en detalle sobre la problemática que buscamos resolver y propondremos un modelo inicial para el módulo de detección de gestos en el cual no se encuentra desacoplado su funcionamiento de los sensores que utiliza. Luego propondremos nuestra propia implementación de una capa de contexto basada en la descripta en [Fortier et al, 2007] y explicaremos en detalle su funcionamiento. Por último, combinaremos ambos modelos para obtener el modelo final de nuestra aplicación, el cual estará finalmente desacoplado de los sensores que se usen en él, que es lo que se buscaba obtener.

En el Capítulo 4 describiremos la idea, interfaz e implementación del prototipo haciendo uso del modelo propuesto en el Capítulo 3, como así también las problemáticas o cuestiones interesantes para mencionar que se encontraron durante el desarrollo del mismo.

En el Capítulo 5 mostraremos casos de uso del prototipo para asentar las funcionalidades del mismo. Primero veremos cómo un usuario realiza el gesto de saludar con la mano derecha o con la izquierda, y luego, veremos cómo dos usuarios interactúan al mismo tiempo con la aplicación. A su vez, para los tres casos, describiremos lo que sucede a nivel de software y mostraremos las acciones que desencadena cada uno de los gestos.

Finalmente en el Capítulo 6 discutiremos sobre lo que se logró en el desarrollo de esta tesis y posibles trabajos futuros en donde el modelo propuesto sea útil o extendido para proveer funcionalidades no presentes.

2. Background

2.1 Sensibilidad al Contexto

En esta sección se describirán los conceptos básicos de contexto, como así también diferentes caracterizaciones o taxonomías existentes. Se presentan diferentes enfoques que utilizan modelos para representar contexto.

2.1.1 Introducción al Contexto

La historia del nacimiento de las aplicaciones sensibles al contexto es descripta por [Schmidt, 2013], donde cuenta que en los primeros años de la computación el contexto del usuario estaba definido por la ubicación en donde se encontrara la computadora debido a que las mismas ocupaban grandes espacios. Pero con el surgimiento de la computación móvil, el contexto del usuario se convirtió en un tema importante a tratar y la preocupación residía en ser capaces de proveer la misma funcionalidad al usuario ante los cambios de su contexto sin que sea notable.

Por los años 90, con la investigación sobre computación ubicua, donde la computación aparenta estar en cualquier lugar y situación desde computadoras y laptops hasta heladeras y anteojos, se introdujo un nuevo interés. Además de proveer transparencia en la funcionalidad, se empezó a pensar en utilizar al contexto para adaptar las aplicaciones mismas. Dado que la ubicación era el dato contextual más fácil de obtener y por ser la movilidad el nuevo aporte de la computación móvil, las primeras aplicaciones sensibles al contexto se centraban en servicios referentes al lugar en donde se encontrara el usuario. De aquí la común confusión de que estas aplicaciones sólo tratan con el posicionamiento.

La idea de aplicaciones sensibles al contexto está relacionada generalmente con la computación móvil y la capacidad de la misma de determinar la posición del usuario (como usando el GPS en un smartphone). Pero el término no se refiere sólo a eso. Veremos en esta sección que el contexto depende de la aplicación que se esté estudiando o desarrollando, y que cada una considera contexto a diferentes elementos.

El término es utilizado por primera vez por [Schilit and Theimer, 1994] donde lo definen como la ubicación del usuario, identificación de las personas y objetos alrededor, y la interacción con las mismas. Por su parte, [Brown, 1997] ve al contexto como la ubicación, identidad de las personas alrededor del usuario, la hora, temperatura, estación del año, etc. [Dey, 1998] lo define como el estado emocional del usuario, su foco de atención, su ubicación y orientación, la fecha, hora y los objetos que se encuentran dentro del ambiente del usuario. Estas definiciones presentan el problema que, al ser por extensión, no son capaces de especificar qué es contexto para diferentes aplicaciones, especialmente cuando es algo fuera de lo nombrado.

Por otra parte, algunos autores intentaron encapsular lo que significa contexto dando una definición más cercana a la de un diccionario. [Brown, 1995] lo define como todo lo que se encuentra en el entorno del usuario que puede ser eventualmente de interés para otro usuario o

una aplicación. Para [Ward at al, 1997] significa el estado de las cosas alrededor de la aplicación. Por último, [Franklin and Flachsbart, 1998] lo ven como la situación del usuario.

En [Dey, 2000] se definió al contexto de la siguiente manera:

"*Contexto es cualquier información que puede caracterizar la situación de una entidad. Siendo una entidad una persona, lugar u objeto que es relevante a la interacción entre el usuario y la aplicación, incluyendo al usuario y la aplicación mismas".*

Dey argumenta que esta descripción ayuda a los desarrolladores a determinar qué es contexto para una aplicación determinada. Dado que la misma fusiona las definiciones por extensión con aquellas cercanas a las del diccionario, la utilizaremos durante el transcurso de esta tesis cuando hablemos de contexto.

Para terminar esta sección daremos un resumen de lo que implica ser sensible al contexto:

Una aplicación sensible al contexto es aquella que utiliza el contexto para proveer al usuario información o servicios relevantes a la situación en la que se encuentra. Por lo tanto, estas aplicaciones se centran en obtener el contexto, abstraer esa información y entenderla (ej. Detectar la temperatura del ambiente del usuario y decidir si hace frío o calor), y reaccionar ante el contexto detectado.

2.1.2 Clasificaciones de Contexto

A lo largo de los años diferentes autores han intentado caracterizar a las aplicaciones sensibles al contexto dando una clasificación de las mismas. [Schilit at al, 1994] determina una clasificación que depende de si la tarea que está realizando el usuario es manual o automática, y si la aplicación está obteniendo información o ejecutando un comando:

Selección por cercanía: son aquellas aplicaciones que obtienen información contextual de forma manual. En este paradigma los objetos o servicios que se le presentan al usuario dependen de la distancia a la que se encuentren.

Reconfiguración contextual automática: son aquellas aplicaciones que obtienen información contextual de forma automática. En este paradigma la aplicación se reconfigura para actuar de forma diferente dependiendo del contexto en el que el usuario esté presente.

Comandos contextuales: son aquellas aplicaciones que ejecutan comandos dependiendo del contexto en forma manual. En este paradigma las aplicaciones actúan de manera diferente ante la misma entrada por el usuario dependiendo del contexto. Por ejemplo, imprimir en la impresora más cercana, o que un mismo botón realice diferentes acciones dependiendo del contexto del usuario.

Acciones ejecutadas por contexto: son aquellas aplicaciones que ejecutan comandos dependiendo del contexto de forma automática. En este paradigma la aplicación se basa en un conjunto de reglas predefinidas para realizar una acción cuando se detecte un contexto determinado.

Por otra parte, [Pascoe, 1998] da una serie de características generales de toda aplicación sensible al contexto independientemente de su función o interfaz. Así las clasifica en cuatro niveles diferentes en base a las características que presenta, en donde cada nivel engloba los anteriores:

Sensado contextual: es el nivel más básico. La aplicación simplemente detecta el contexto y se lo presenta al usuario de una manera conveniente.

Adaptación contextual: la aplicación no sólo detecta el contexto sino que lo utiliza para modificar su comportamiento.

Descubrimiento de recursos contextuales: la aplicación es capaz de encontrar recursos dentro del mismo contexto del usuario y utilizarlos mientras permanezcan en el mismo contexto. Por ejemplo, el teléfono celular del usuario detecta que hay una pantalla sin usar en la cercanía y puede utilizarla temporalmente para mostrar la información de una forma menos compacta.

Aumento contextual: además de detectar, reaccionar e interactuar con el contexto, la aplicación puede aumentar el ambiente con mayor información. Esto se logra asociando información digital con un contexto particular. Dependiendo del punto de vista del usuario, la información digital puede estar aumentando la realidad, o la realidad aumentando la información digital.

[Dey, 2000] nota que las dos clasificaciones anteriores comparten algunas similitudes pero que se centran en aspectos diferentes, provocando así que las definiciones no sean completas. Por ejemplo, Pascoe define el concepto de *contexto aumentado* que no aparece en la clasificación de Schilit, pero no menciona la presentación de comandos contextuales. Por esto decide dar su propia clasificación fusionando las anteriores:

Presentación de información o servicios. Esta característica fusiona la *selección por cercanía* y los *comandos contextuales* de Schilit con la noción de presentar el contexto al usuario de Pascoe.

Ejecución automática, que es una fusión de las *acciones ejecutadas por contexto* de Schilit con la *adaptación contextual* de Pascoe.

Etiquetado de contexto, que es lo mismo que el *aumento contextual* de Pascoe.

Dey luego agrega que en esta clasificación no hace diferencia entre información y servicios, y que el descubrimiento de recursos no debe ser tratado por separado sino que está dentro de las dos primeras características.

En cambio, [C. Emmanouilidis et al, 2013] realizan una clasificación de los diferentes parámetros de contexto en lugar de las aplicaciones. Por más que su estudio esté centrado en guías móviles en particular, su clasificación resulta útil para gran parte de las aplicaciones sensibles al contexto, y es por esto que lo tomamos en cuenta al escribir este trabajo. En [C. Emmanouilidis et al, 2013] los autores separan en cinco categorías todas las características del usuario o de la aplicación que ellos consideran que pueden ser tomadas como contexto. Estas categorías son: Usuario, Sistema, Ambiente, Social y Servicio. Estas categorías se pueden apreciar en la Figura 1.

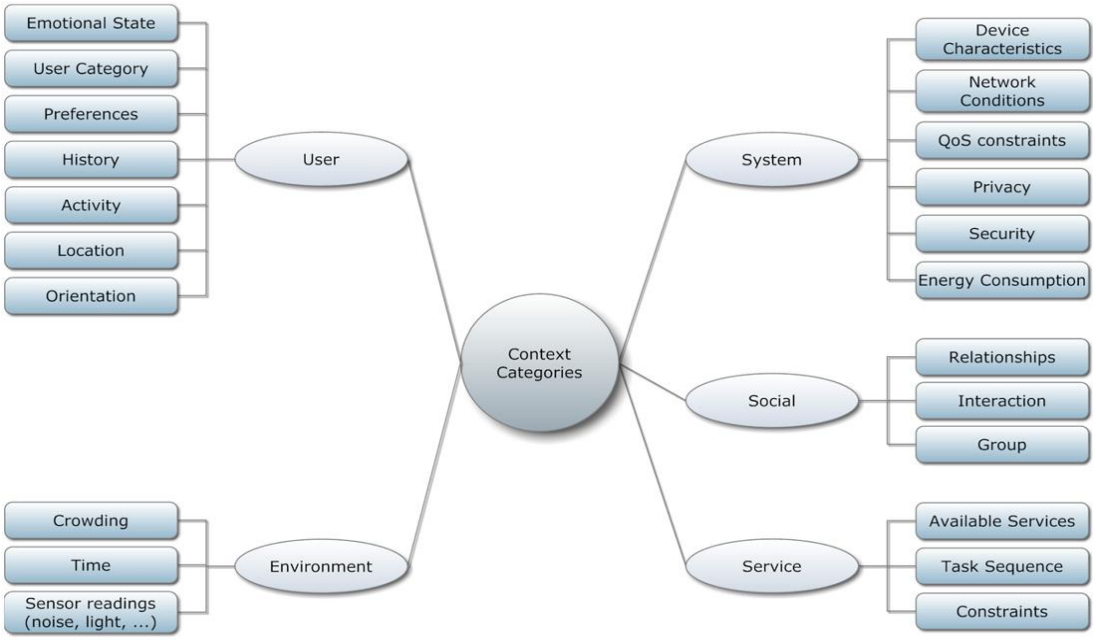

Figura 1: Clasificación de contexto [C. Emmanouilidis et al, 2013]

Se puede observar en la Figura 1, que dentro de la categoría Usuario, que quizás es la más importante en lo que concierne a aplicaciones sensibles al contexto, se encuentran parámetros como las preferencias, ubicación, o hasta el estado emocional; Sistema se refiere a todo el contexto no funcional, mayormente lo relevante a restricciones tecnológicas; Ambiente engloba las características del ambiente físico como el tiempo, ruido o luz; Social es una categoría que está creciendo en importancia últimamente con el desarrollo de las redes sociales, la misma tiene en cuenta las relaciones e interacciones del usuario con otras personas y los grupos a los que pertenece; y por último, Servicio representa a las funcionalidades que están activas o no en cierto momento.

A su vez, [C. Emmanouilidis et al, 2013] realizaron una investigación sobre las aplicaciones sensibles al contexto existentes para detectar cuánto de estas categorías estaban siendo utilizadas. Descubrieron que el contexto más utilizado es el de la ubicación del usuario, y que gran parte de las demás características no estaban siendo explotadas. Esto se puede apreciar en la Figura 2. Se puede observar que el contexto más usado es la posición del usuario.

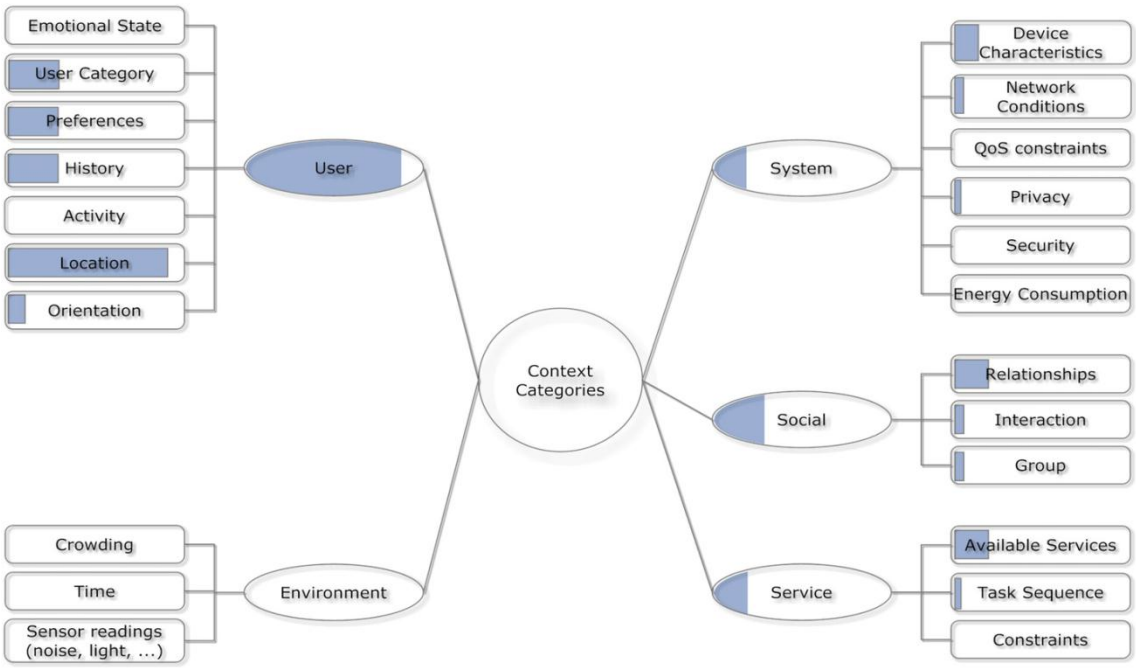

Figura 2: Uso de contexto [C. Emmanouilidis et al, 2013]

Todas estas clasificaciones vistas sirven para identificar los diferentes contextos que puede tener una aplicación, para así establecer las características principales que debemos tener en cuenta al momento de desarrollarla. Cuantos más contextos se tengan en cuenta en una aplicación, más compleja se volverá el diseño de la misma.

2.1.3 Modelos de Contexto

A continuación presentaremos algunas ideas existentes de diferentes autores sobre modelos de aplicaciones sensibles al contexto.

2.1.3.1 CAMUS Context Model

En el paper presentado por [Shehzad et al,2004] modelan un framework para aplicaciones sensibles al contexto utilizando OWL (Web Ontology Language) y su arquitectura CAMUS introducida en un paper anterior. En el mismo categorizan a las entidades relacionadas en cinco grupos: agentes, dispositivos, ambiente, lugar y tiempo. Estos se pueden observar en la Figura 3. Dada la importancia que tiene para los autores el lugar y el tiempo en las aplicaciones contextuales, se encuentran en su propia clasificación en lugar de pertenecer a la misma categoría que las demás entidades del ambiente. Toda la información provista por las entidades son tomadas por los módulos de razonamiento que son los encargados de determinar el contexto en el que se encuentra la aplicación. Estos módulos son divididos en dos grandes grupos:

- Los módulos de razonamiento ontológico que permiten encontrar relaciones de subsunción (entre subconcepto y súper-concepto), relaciones de instancia (el individuo A

es una instancia del concepto C) y consistencia del conocimiento contextual. Para todo esto OWL provee una expresividad eficiente y muy potente que los autores han decidido aprovechar. Un ejemplo de este razonamiento es el siguiente: “localizado en” es una propiedad transitiva y “es parte de” es una sub-propiedad de “localizado en”. Por lo tanto, si el sujeto A está “localizado en” Cama, y Cama “es parte de” Dormitorio que “es parte de” Casa, entonces el sistema deduce que el sujeto A está “localizado en” Dormitorio y Casa.

- Los módulos de razonamiento lógico que se basan en un conjunto de reglas para determinar si el usuario se encuentra en un contexto determinado cuando se cumple una serie de condiciones. Estas condiciones pueden ser “antes”, “después”, “actividad actual”, “actividad intencionada”, “desconectado de”, etc.

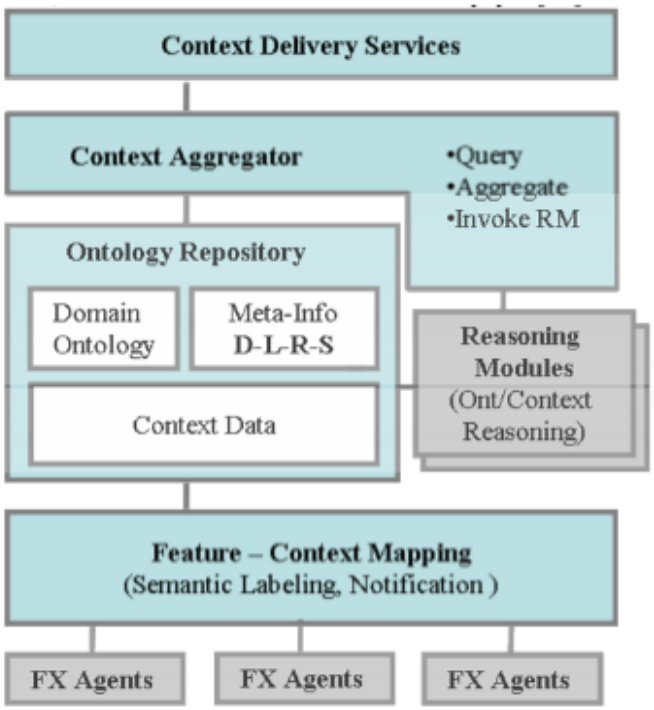

Figura 3: Arquitectura de CAMUS Context Model [Shehzad et al, 2004]

2.1.3.2 Context Toolkit

Por su parte, [Dey et al, 2001] desarrollaron un framework en Java para crear módulos que luego puedan ser utilizados por cualquier aplicación. Estos módulos se dividen en cinco categorías: widgets, interpretadores, agregadores, servicios y descubridores. En la Figura 4 se puede avisar las relaciones entre estas cinco categorías mencionadas.

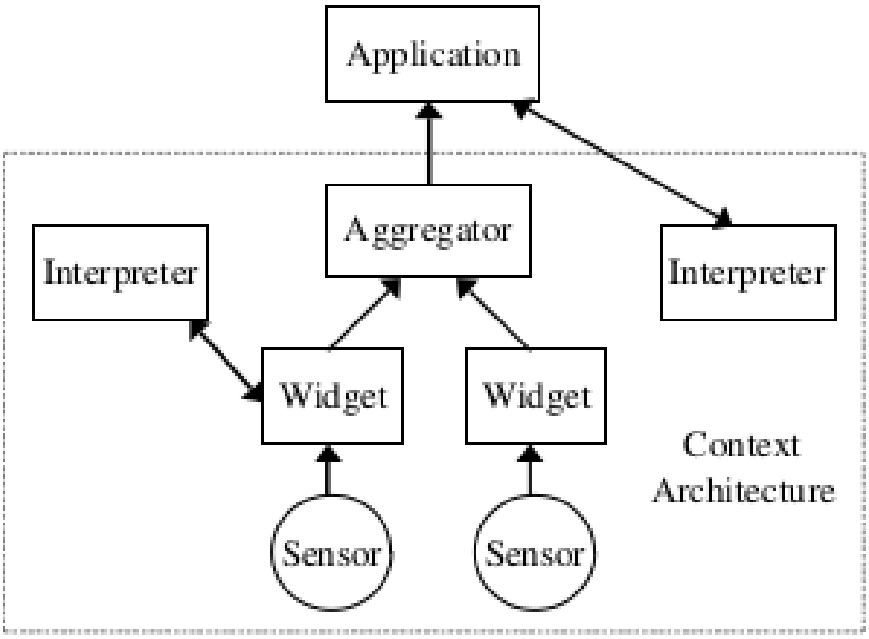

Figura 4: Context Toolkit Model [Dey et al, 2001]

A continuación se brindan más detalles de los elementos descriptos en la Figura 4.

Los widgets son los encargados de esconder los detalles de los dispositivos que están siendo utilizados. Evitan al programador tener que involucrarse con la complejidad del sensor usado para la aplicación. Por ejemplo, si se sensa la ubicación de un usuario a través de tarjetas magnéticas, un sistema de posicionamiento dentro de un edificio, GPS o radio frecuencia no debería provocar ningún cambio en el comportamiento de la aplicación. A su vez, los widgets abstraen esta información y la proveen según las necesidades de la aplicación. Volviendo al ejemplo del widget de posicionamiento del usuario, si el sensado es dentro de un edificio o en una ciudad, el widget notificará cuando el usuario pase de una habitación a la otra, o de una calle a otra, y no proveerá cambios menores a la aplicación.

Usualmente las aplicaciones necesitan un nivel mayor de abstracción del contexto de lo que los sensores o widgets pueden proveer. Coordenadas geográficas pueden ser traducidas a nombres de calles o avenidas usando alguna información geográfica del lugar, o se puede inferir nueva información en base a datos más simples como la presencia de personas en una habitación sumado a un ruido ambiente elevado puede llevar a la deducción de que el usuario se encuentra en una reunión. Los interpretadores son los responsables de llevar a cabo esa abstracción. Generalmente toman datos de uno o más widgets y lo transforman en información más abstracta para la aplicación.

La responsabilidad de los agregadores reside solamente en agrupar los diferentes widgets, que pueden estar potencialmente distribuidos, en un solo componente de software. Así cada agregador puede facilitar a la aplicación al recuperar el contexto completo de una entidad de una manera simple y reutilizable.

A diferencia de las categorías anteriores, la responsabilidad de los servicios no es de recuperar o facilitar el contexto, sino que se encargan de llevar a cabo la acción vinculada al contexto detectado según la lógica de la aplicación. Las implementaciones de estos servicios son, generalmente, acciones recurrentes en varias aplicaciones, como por ejemplo, mostrar un mensaje. Los mismos facilitan el desarrollo de aplicaciones dado que, al igual que los widgets, no es necesario que el desarrollador sea consciente de la implementación del mismo y pueda acceder a acciones comunes de manera instantánea.

Por último se encuentran los descubridores. Su tarea es la de mantener un registro de todos los componentes existentes en el framework. Esto es, saber qué widgets, interpretadores, agregadores y servicios están disponibles, notificar ante la aparición o falla de uno de ellos, y proveer la capacidad de buscar entre todos por algún criterio. Las aplicaciones utilizan estos descubridores para encontrar los componentes que le son de interés y abstraen a la misma de saber en dónde se encuentra cada uno de estos.

2.1.3.3 Context-Aware Framework

Como último ejemplo de modelo de aplicaciones sensibles al contexto veremos el desarrollado por [Fortier et al, 2007].

Este modelo presenta una arquitectura orientada a objetos que separa todo lo referente a contexto con el modelo de dominio puro de la aplicación. Así, la lógica de modelo está presente en la implementación de la aplicación y es independiente de la presencia de la capa de contexto. Dicha capa posee todas las clases encargadas del contexto y es adaptable a toda aplicación a través de la sub-clasificación de las mismas, haciéndolo un framework de caja blanca.

El framework trabaja en base al disparo de eventos ante los cambios de contexto y en respuesta a esos disparos, generalmente, viene asociada una acción. Todo el funcionamiento desde el cambio de contexto hasta la realización de la acción adecuada viene provisto por las clases que componen la capa de contexto. La Figura 5 muestra las principales clases del framework. Estas clases se describen a continuación:

AwareObject: Es la contracara de un objeto del dominio en la capa de contexto. Este representa a un objeto dado en uno o más ambientes de adaptación. El *AwareObject* es el encargado de propagar todos los eventos de contexto producidos por el objeto en cuestión a los ambientes de adaptación a los que pertenece. Para esto, cuenta con un conjunto de *ContextFeatures* que dan aviso de los cambios que se producen en ese objeto del dominio.

ContextFeature: Representa una característica de un objeto del modelo puro de dominio con una cierta relevancia y que forma parte del contexto. Se encarga de guardar el valor de esa característica y notificar al *AwareObject* adecuado cuando este valor cambia.

AdaptationEnvironment: Es el canal por el que se comunican los *AwareObjects*. Cada implementación de las subclases puede proveer un funcionamiento diferente ante los eventos disparados. La granularidad con la que se crean estos ambientes puede ser tan pequeña (como un

ambiente sólo para un *AwareObject* específico) o tan grande (como un ambiente para toda la aplicación) como se quiera.

EventHandler: Cuando un evento fue disparado por un *ContextFeature* ante un cambio de contexto, es capturado por un *AwareObject*. El mismo, a su vez, transmite el evento a todos los *AdaptationEnvironment* a los que está suscripto. Todos los *EventHandler* que estén suscriptos a un ambiente serán notificados de este nuevo evento. Son los responsables de decidir si el mismo corresponde ser tratado por ellos y, si es el caso, realizar la acción acorde.

MatchingPolicy: Es la clase de la que hacen uso los *EventHandler* para decidir si un evento les es de interés o no. Algunas implementaciones pueden decidir sobre la clase del evento disparado, sobre el *AwareObject* que lo disparó, o simplemente tratar cualquier evento.

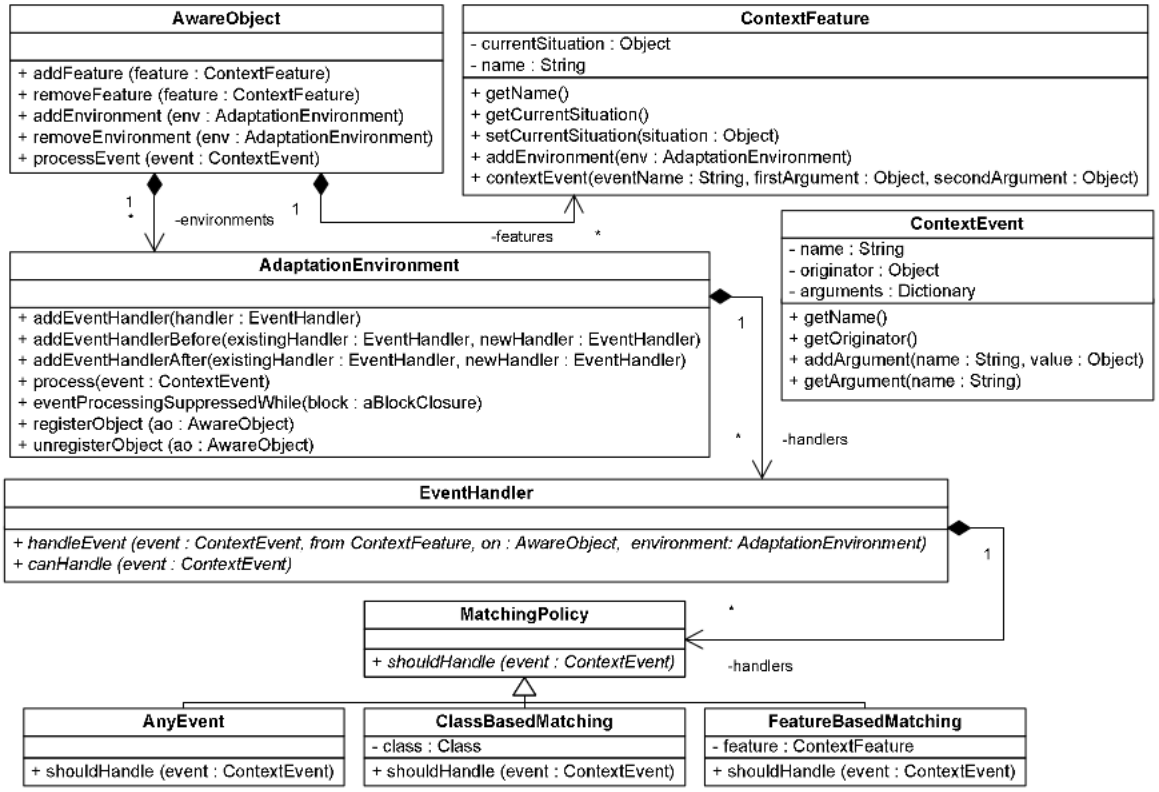

Figura 5: Context-aware Model [Challiol et al, 2007]

Este modelo se fue desarrollando a lo largo de varios papers de los mismos autores, pero en ninguno de ellos detallan las implementaciones de las clases mostradas ni cómo interactúan para que los eventos sean propagados y se llegue a ejecutar la acción determinada por los *EventHandler*.

2.2 Kinect

Kinect [1] es un controlador de juego y entretenimiento creado por Alex Kipman, desarrollado por Microsoft y lanzado a la venta para la consola Xbox 360 en noviembre 2010 y en junio 2011 para Windows 7 y 8. Kinect fue anunciado por primera vez el 2 de junio de 2009 en la Electronic Entertainment Expo 2009 como “Project Natal”.

El sensor de Kinect es una barra horizontal conectada a una base circular con un eje de articulación de rótula, y cuenta con un sensor de profundidad, una cámara a color, un micrófono de múltiples matrices y un emisor infrarrojo, permitiendo que sea posible detectar la ubicación de los usuarios, sus movimientos e incluso sus voces.

Todos estos sensores en un mismo dispositivo permiten que sea usado de tal forma que el cuerpo del usuario sea lo único necesario para interactuar con las aplicaciones desarrolladas. A su vez, Microsoft provee un SDK que permite a los desarrolladores tener al alcance cosas como captura de movimiento de todo el cuerpo en 3D, reconocimiento facial y reconocimiento de voz.

Por más que el dispositivo haya sido pensado para videojuegos, se han ideado e implementado otros usos. Algunos ejemplos son: producir escaneos 3D de alta calidad, traducción del lenguaje de señas, controlar robots con el movimiento del cuerpo, recuperación para pacientes que sufrieron un accidente cerebrovascular, etc.

En la Figura 6 se puede apreciar una imagen de Kinect v1.

Figura 6: Kinect for Windows v1

Kinect es capaz de detectar hasta seis personas que estén dentro de su campo de visión pero puede monitorear el movimiento de sólo dos de ellas. Cada una de las personas detectadas es asignada un *id* de las cuales Kinect, por defecto, toma a aquellas que primero fueron detectadas para monitorear sus movimientos. Las aplicaciones pueden definir su propia lógica para

1 https://www.microsoft.com/en-us/kinectforwindows/

seleccionar a las personas, como por ejemplo aquellas que estén más cerca de la cámara, o la persona que tenga su mano levantada.

En la Figura 7 se puede apreciar que el esqueleto detectado de las dos personas posee varios puntos. Cada uno de ellos representa una articulación que reconoce el Kinect y que, varias veces por segundo, son brindados junto con sus posiciones absolutas en tres dimensiones. A su vez, existen dos modos en los que el usuario puede ser monitoreado. El primero es el modo por defecto en donde el usuario se encuentra parado y su esqueleto se compone de 20 articulaciones diferentes. El otro modo fue pensado para monitorear a una persona que esté sentada (aunque no es necesario) y su esqueleto posee sólo las 10 articulaciones de la parte superior del cuerpo. Cada una de estas articulaciones puede estar en uno de tres estados diferentes: puede encontrarse *monitoreada* si es claramente visible por el Kinect, *inferida* donde no puede verse y su posición es inferida, o *no-monitoreada*, por ejemplo, las 10 articulaciones inferiores en el modo sentado.

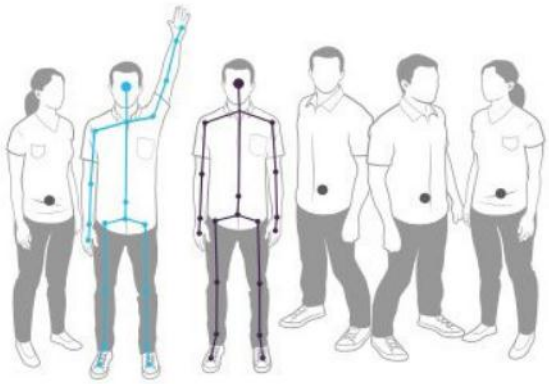

Figura 7: Esqueleto detectado [https://msdn.microsoft.com/en-us/library/hh973074.aspx]

Como ya mencionamos anteriormente existen muchos usos interesantes para el Kinect fuera del ámbito de los videojuegos. A continuación describimos algunos que encontramos interesantes:

1. **Herramienta de dibujo:** Binbin Xu, un estudiante de la escuela politécnica de Milán, desarrolla en su tesis [Binbin Xu, 2014] una herramienta que hace uso del Kinect para detectar los movimientos y gestos de niños con el fin de que dichos movimientos sean traducidos en pinceladas de colores sobre un dibujo o un lienzo en blanco. El mismo menciona que este proyecto está basado en reconocer la relación entre actividad física y procesos cognitivos, y asegura que está respaldado por evidencia proveniente de la psicología y la neurobiología. En la Figura 8 se puede ver la aplicación siendo usada en uno de sus modos, el cual consta de conectar los puntos en orden usando movimientos de la mano.

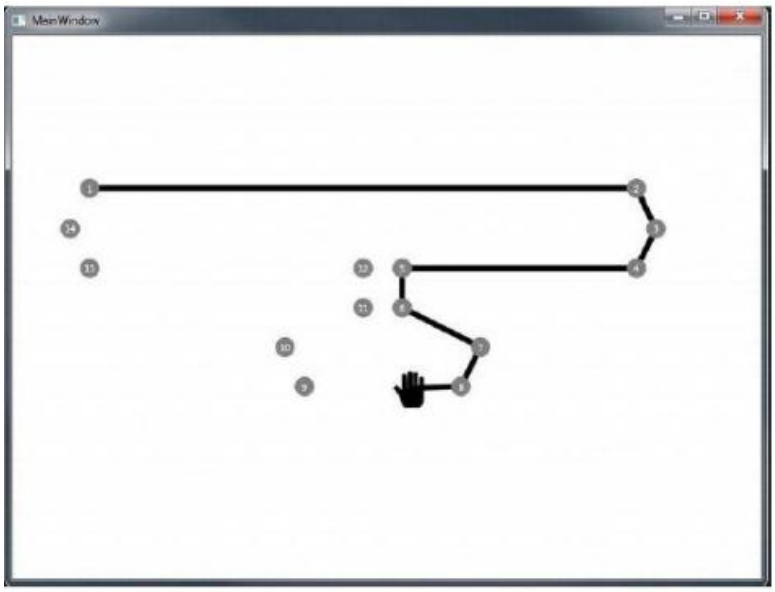

Figura 8: Aplicación de dibujo [Binbin Xu, 2014]

2. **Detectar artritis reumatoide:** En el artículo publicado por [Bond and Curran, 2014] utilizan el Kinect como sensor para detectar artritis reumatoide en pacientes. Los autores eligieron este dispositivo por ser bastante avanzado, estar a un precio muy accesible y contar con un SDK que permite a los desarrolladores tener acceso a este tipo de tecnología a un costo muy reducido. La aplicación solicita al usuario que haga algunos ejercicios con las manos frente a la cámara del Kinect para tomar medidas como capacidad de estirar la mano, o tiempo que tarda en cerrarla. Estas medidas son grabadas y pueden ser luego evaluadas por un médico. En la Figura 9 se puede apreciar al usuario haciendo los ejercicios que le requiere la aplicación y cómo es representada la mano.

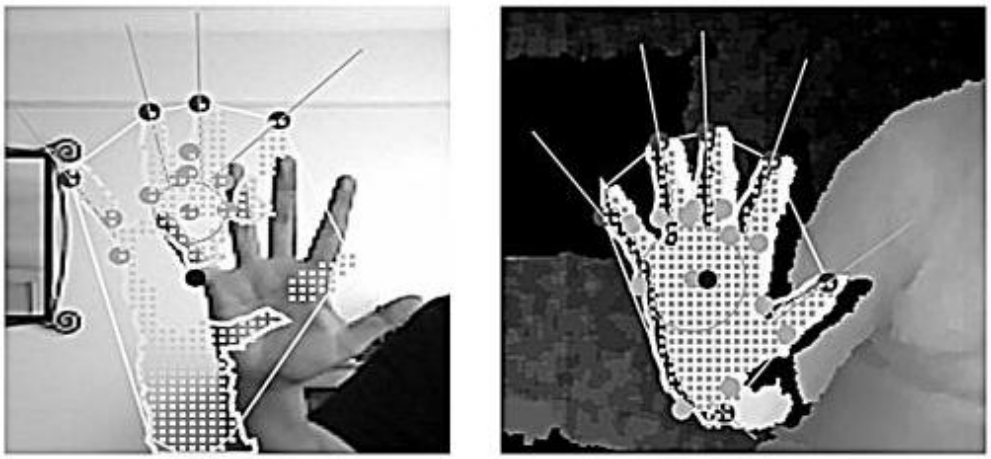

Figura 9: Aplicación para detectar artritis reumatoide [Bond and Curran, 2014]

3. **Control de imágenes médicas**: En una sala de operaciones o cualquier otro ambiente médico crítico es de vital importancia que el lugar esté totalmente esterilizado. A su vez, el uso de imágenes médicas en estos tipos de ambientes son de gran ayuda para los médicos. El problema se presenta en el momento que el médico desea interactuar con el sistema que esté instalado para acceder a diferentes imágenes dado que, como se debe mantener la esterilización, el médico no puede tener contacto físico. Es por esto que en [Gallo et al, 2011] se desarrolló un sistema utilizando el Kinect donde el médico, utilizando gestos, es capaz de navegar a través de la base de datos de imágenes del sistema sin salir del ambiente crítico y poner en riesgo al paciente. En la Figura 10 se puede ver al médico interactuando con la aplicación.

Figura 10: Control de imágenes médicas [Gallo et al, 2011]

4. **Control de vuelo de un cuadricoptero:** En [Stowers et al, 2011] los autores proclaman que este fue el primer intento de utilizar el Kinect para una aplicación de control de robots en tiempo real. Los mismos hicieron uso del sensor de profundidad para controlar la altitud de un cuadricoptero. El experimento se realizó atando el Kinect al cuadricoptero y apuntando hacia el suelo, como se puede ver en la Figura 11. El Kinect, a su vez, estaba conectado por un cable USB a una laptop por lo que el cuadricoptero se encontraba limitado en su movimiento. Sin embargo, el experimento sirvió para demostrar que el Kinect es capaz de trabajar en ambientes dinámicos y es una buena opción como sensor en el ámbito de la robótica por su bajo costo, alta frecuencia de imágenes y un sensor de profundidad certero.

Figura 11: Kinect atada a un cuadricoptero [Stowers et al, 2011]

3. Modelo Propuesto

En este capítulo presentaremos primero la problemática que se intenta resolver y luego pasaremos a explicar el modelo propuesto que resuelve la problemática planteada. Primero se mostrará un modelo inicial de gestos, luego una descripción de la capa de contexto a utilizar y para finalizar se mostrará cómo el modelo de gestos puede ser combinado con una capa de contexto para lograr flexibilidad.

3.1 Presentación de la problemática

Como se ha visto en la Sección 2.2 existen varios usos del Kinect pero ninguno de ellos muestra una separación clara entre el dominio o lógica de la aplicación y la recuperación de los datos a través del Kinect. Esto puede ser extendido a todo tipo de sensores; las aplicaciones no deberían ser totalmente dependientes de los dispositivos que se estén utilizando. Si en un futuro se pretende cambiar el mecanismo de sensado o surge una nueva tecnología que a los desarrolladores les interesa para su aplicación, es muy probable que para adaptarse a ese cambio sea necesario reescribir el código completo. A su vez, las aplicaciones mencionadas en la sección anterior pueden verse como buenos ejemplos de aplicaciones sensibles al contexto, pero ninguno de los artículos menciona o determina claramente el contexto de su aplicación.

La problemática que se busca resolver es contar con un modelo general de gestos desacoplado del mecanismo de sensado, que luego en el prototipo a implementar se tomarán del Kinect. Este modelo se definirá de manera independiente a las acciones que desencadenan dichos gestos. Es decir, se deberá tener la definición de los gestos y por otra parte las acciones que se pueden realizar cuando se detectan los mismos. Este modelo de gestos permitirá que cada usuario configure cómo cada gesto es interpretado, y qué acciones se dispararán cuando se detecta dicho gesto.

Cabe destacar que el modelo de gestos definirá gestos generales, por ejemplo, mover la mano derecha o la mano izquierda. Por otro lado, se desea poder contar con la posibilidad de que para distintos usuarios esos gestos tengan una semántica diferente (acciones distintas). Esto requiere que se modele el concepto de usuario y su configuración particular de gestos.

Con el modelo de gestos uno esperaría poder definir, por ejemplo, que el *usuario1* defina que el gesto de mover la mano derecha significa "*enter*", mientras un usuario2 defina el mismo gesto para indicar "*rotación*".

El modelo tiene que ser capaz de detectar los cambios de contexto de un usuario (sus gestos) y acorde a esto reaccionar para aplicar las acciones configuradas para este usuario. Para resolver los aspectos de contexto se utilizará como base los conceptos presentados en [Fortier et al, 2007]. De esta manera se logrará independizar el contexto del dominio de gestos que se quiere representar.

3.2 Modelo de Gestos inicial

Como ya se mencionó en la Sección 3.1, la aplicación maneja varios usuarios. Por lo tanto, tendremos que definir una clase para representarlos. Llamaremos a esta clase *User* y por ahora como variables de instancia sólo tendrá el nombre del usuario y un código que lo identifique. A su vez, necesitaremos de otra clase llamada *UserManager* para poder manejar la colección completa de usuarios. Además, deberá mantener una referencia a aquellos que estén actualmente interactuando con el sistema (esto se realiza mediante la relación *currentUsers*). Por otra parte, tendremos una clase llamada *SkeletonManager* que se encargará de obtener los datos de los movimientos de los usuarios, a través de los sensores, junto con un valor que identifique al usuario que realizó el movimiento. Estos datos están representados por la interfaz *Skeleton,* la cual tendrá una implementación diferente acorde al sensor que se utiliza, por ejemplo, identificar la posición de cada parte de interés del cuerpo. Una vez obtenida o generada una instancia que implemente la interfaz *Skeleton*, el *SkeletonManager* se la provee a la instancia de *UserManager*. Es responsabilidad de la clase *UserManager* mantener un mapeo entre los identificadores del sensor y el usuario correspondiente. En la Figura 12 se puede apreciar un diagrama de las clases mencionadas.

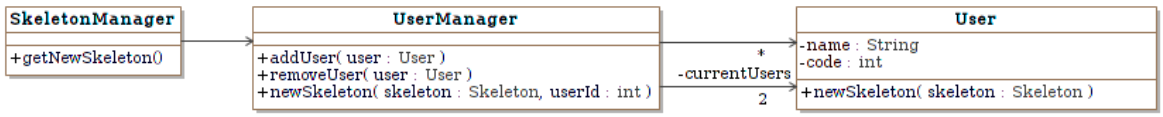

Figura 12: Modelado de los usuarios

A continuación pasaremos a describir el otro pilar central del modelo, los gestos. Para los mismos contamos con una clase abstracta *Gesture* (ver Figura 13) que contiene una lista ordenada de *GestureParts*, el contenido de dicha lista así como su longitud es definido estáticamente por cada sub-clase concreta de *Gesture.* Cabe destacar que *GestureParts* es una interfaz que permite representar el comportamiento general a cualquier parte de un gesto. Las clases particulares que implementen esta interfaz se comportaran como partes de gestos. Se puede apreciar en la Figura 13 que la clase *Gesture* implementa el método *next()* que dado un *GesturePart* en particular retorna el siguiente en la lista ordenada de sus partes. En la Figura 13 se pueden apreciar la clase *Gesture* junto a una sub-clase particular denominada *RightHandWave*. Además, se crean dos partes de gestos que al momento de la instanciación y uso permitirán realizar el gesto de saludar con la mano derecha (*RightHandWave*). Las clases *RightHandWave, RightHandWavePart1 y RightHandWavePart2* fueron creadas para ejemplificar como se extiende el modelo de gestos propuesto, mostrando así un ejemplo simple de gesto y sus partes. Notar que, como el modelo está definido de manera general (con las clases *Gesture* y la interfaz *GestureParts*) puede ser extendido para representar cualquier gesto.

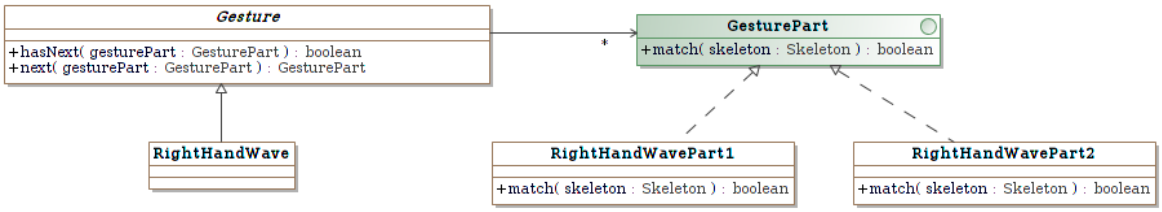

Figura 13: Gesture y GesturePart

Cabe volver a mencionar que las clases que implementan la interfaz *GesturePart* mencionada anteriormente son los que componen, en conjunto, un gesto completo. Estas clases deben implementar el método *match()* que recibe un esqueleto y devuelve un booleano. La semántica de este método es la de verificar si ciertas articulaciones del esqueleto recibido coinciden con lo definido por cada clase que implementa dicha interfaz. Por ejemplo, el gesto de saludar con la mano derecha (*RightHandWave*) implica dos partes repetidas una cierta cantidad de veces. Estas partes son:

1) La muñeca de la mano derecha debe estar por encima y a la derecha del codo derecho (*RightHandWavePart1*).
2) La muñeca de la mano derecha debe estar por encima y a la izquierda del codo derecho (*RightHandWavePart2*).

Estas dos partes se pueden repetir, como se mencionó anteriormente, un número de veces, con lo cual estas partes aparecerán en la lista de partes del gesto en sí. Cuando el usuario realice dicho gesto, el mismo se va a considerar realizado cuando el usuario realice todas las partes del mismo en el orden especificado en dicha lista de partes del gesto.

Para poder llevar un control del gesto realizado por el usuario es que se implementa la clase llamada *UserGesture* (ver Figura 14). Esta clase cuenta con una referencia a una instancia de *Gesture* y una relación denominada *currentGesturePart* que permite indicar cuál es el siguiente *GesturePart* que debe ser leído por el/los sensore/s. Esto permite controlar la realización de un gesto. Cada usuario puede estar realizando distintos gestos, los cuales pueden involucrar diferentes partes.

La clase *UserGesture* define las variables *MAX_FRAME_COUNT,* para indicar el tiempo máximo entre la realización de dos partes, y *frameCount* para mantener el tiempo transcurrido desde la realización de la parte anterior. Cabe destacar que el tiempo es medido en cantidad de esqueletos procesados y la razón por la que se debe tener en cuenta es que la mayoría de los sensores proveen múltiples datos de entrada por segundo, por lo que es razonable pensar que un usuario es incapaz de realizar un gesto completo en fracciones de segundo.

La clase *UserGesture* cuenta con el método *update()* que recibe un esqueleto y verifica si no se excedió el tiempo entre dos partes, como también si el *GesturePart* actual se realizó con éxito. En caso afirmativo, le solicita a su instancia de *Gesture* la siguiente parte que lo compone. El

método devuelve un booleano que indica si ya se completaron todas las partes o no del gesto en cuestión.

Es de esperarse que los usuarios tengan varios gestos que pueden realizar, es por eso que existe la problemática de identificar cuál, de entre todos los gestos, es aquel que el usuario está intentado ejecutar. Para representar esto, cada usuario cuenta con una instancia de la clase *UserGestureManager* que contiene una colección de *UserGesture.* Esto permite estar controlando simultaneamente por la realización de varios gestos, y cuando uno de ellos es completado satisfactoriamente se reinicia el estado de todos los demás. La decisión de reiniciar el estado se debe a que los sensores siguen capturando el esqueleto del usuario y el mismo podría haber realizado un movimiento que, en conjunto con algunos de los movimientos realizados del gesto anterior, completan un segundo gesto que no se pretendía realizar. Cuando una instancia de *User* recibe un esqueleto se lo pasa a su instancia de *UserGestureManager* para que éste lo verifique contra todos los gestos. Si alguno de ellos es completado, se deben ejecutar las acciones correspondientes a dicho gesto y reiniciar todos los demás para poder comenzar a procesar gestos nuevos.

Como se mencionó en la Sección 3.1 es deseable que las acciones a ejecutar ante cada gesto sea completamente dinámico y configurable a cada usuario. Por esto es que modelamos la clase llamada *GestureActionsAssociation* que contiene una asociación entre un gesto y una acción que el mismo desencadenará al momento de completarse exitosamente. El usuario contará con una colección de instancias de esta clase que denotan, sólo para ese usuario, qué acciones se ejecutan ante la realización de cada gesto. Notar que es posible que existan varias instancias de *GestureActionsAssociation* que hagan referencia al mismo gesto pero a diferentes acciones. Esto permite que un mismo gesto desencadene varias acciones y que agregar o remover una de ellas sea tan simple como agregar o eliminar una instancia de la colección de *GestureActionsAssociation* del usuario. Las acciones a su vez son representadas por objetos que implementen la interfaz *Action,* la cual define el método *execute().* Cada clase que implementa esta interfaz definirá este código acorde a la semántica de cada acción concreta. Notar que la clase *Action* se define usando el patrón de diseño Command [Gamma et al, 1994]. Cada instancia de *User* recorrerá su colección de *GestureActionsAssociation*, en el momento que se complete un gesto, para encontrar todas las acciones que se deban ejecutar. En la Figura 14 se pueden apreciar las clases mencionadas anteriormente.

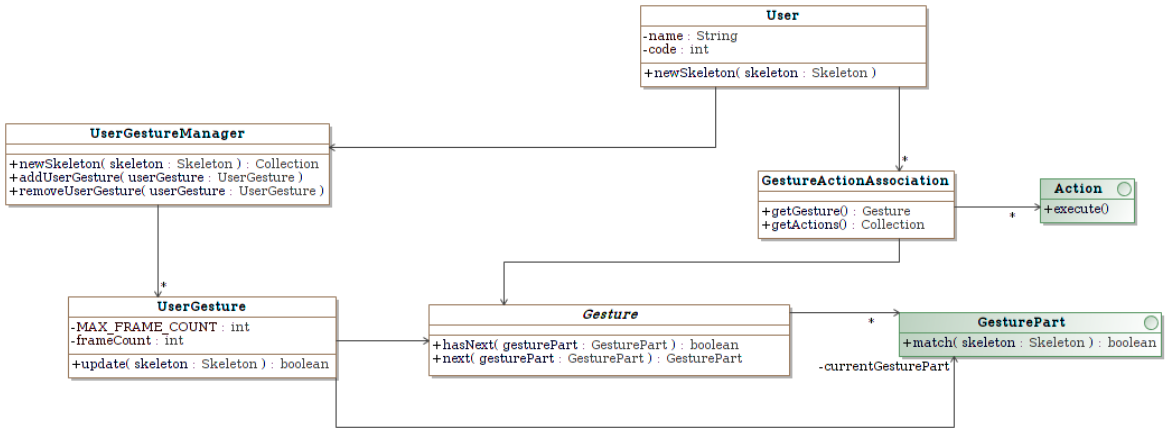

Figura 14: GestureManager y Actions

Notar que el modelo descripto es totalmente genérico, por esto es posible re-utilizarlo para múltiples aplicaciones donde se requiera la detección de gestos. Cada aplicación que utilice este modelo debe implementar los gestos específicos que se quieren detectar y las acciones que se quieren ejecutar para dicha aplicación. Los mecanismos de detección de gestos y ejecución de acciones asociada a los mismos vienen provistos por el modelo propuesto.

En la Figura 15 se muestra el modelo de gesto propuesto. Se puede apreciar en dicha figura que las clases relacionadas al gesto de "*saludar con la mano derecha*" no son parte del modelo general, se habían introducido anteriormente solo como ejemplificación de cómo extender la clase *Gesture* o implementar la interfaz *GesturePart*.

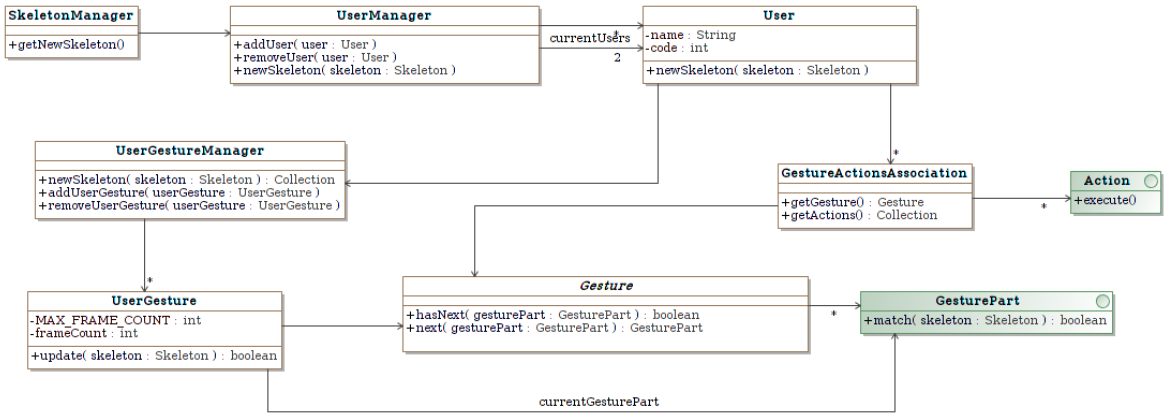

Figura 15: Modelo de gestos propuesto

3.3 Capa de Contexto Propuesta

La capa de contexto propuesta se basará en el modelo ya mencionado en la Sección 2.1.3.3 [Fortier et al, 2007], en dicha sección se presentaron las principales clases del mismo en la

Figura 5. Para una ayuda al lector volveremos a mostrar dicho modelo en la Figura 16[2], para luego proponer algunas modificaciones en la capa de contexto propuesta en este trabajo.

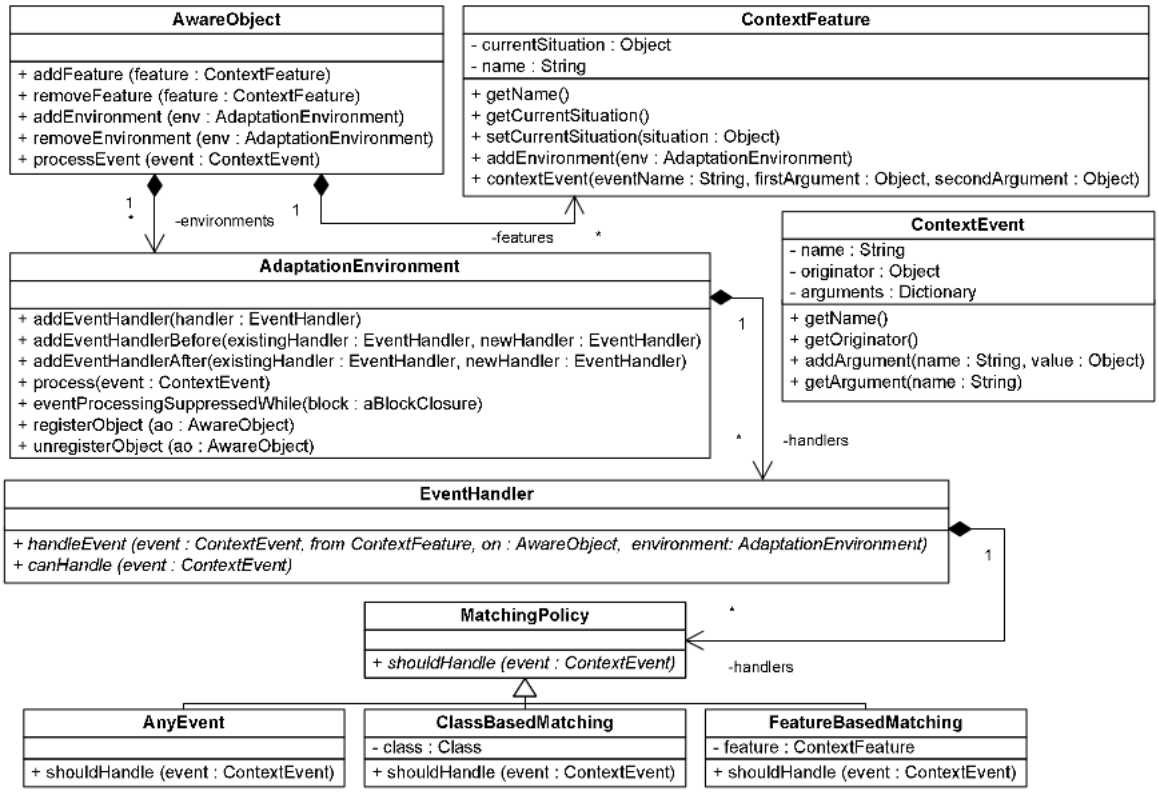

Figura 16: Context-aware Model [Fortier et al, 2007]

[2] Recordemos la semántica de cada clase presentada.

AwareObject: Es la contracara de un objeto del dominio en la capa de contexto. Este representa a un objeto dado en uno o más ambientes de adaptación. El *AwareObject* es el encargado de propagar todos los eventos de contexto producidos por el objeto en cuestión a los ambientes de adaptación a los que pertenece. Para esto, cuenta con un conjunto de *ContextFeatures* que dan aviso de los cambios que se producen en ese objeto del dominio.

ContextFeature: Representa una característica de un objeto del modelo puro de dominio con una cierta relevancia y que forma parte del contexto. Se encarga de guardar el valor de esa característica y notificar al *AwareObject* adecuado cuando este valor cambia.

AdaptationEnvironment: Es el canal por el que se comunican los *AwareObjects*. Cada implementación de las subclases puede proveer un funcionamiento diferente ante los eventos disparados. La granularidad con la que se crean estos ambientes puede ser tan pequeña (como un ambiente sólo para un *AwareObject* específico) o tan grande (como un ambiente para toda la aplicación) como se quiera.

EventHandler: Cuando un evento fue disparado por un *ContextFeature* ante un cambio de contexto, es capturado por un *AwareObject*. El mismo, a su vez, transmite el evento a todos los *AdaptationEnvironment* a los que está suscripto. Todos los *EventHandler* que estén suscriptos a un ambiente serán notificados de este nuevo evento. Son los responsables de decidir si el mismo corresponde ser tratado por ellos y, si es el caso, realizar la acción acorde.

MatchingPolicy: Es la clase de la que hacen uso los *EventHandler* para decidir si un evento les es de interés o no. Algunas implementaciones pueden decidir sobre la clase del evento disparado, sobre el *AwareObject* que lo disparó, o simplemente tratar cualquier evento.

A continuación se mencionan las modificaciones que se proponen en este trabajo tomando como base el modelo presentado en la Figura 16.

En cuanto a la clase *AdaptationEnvironment* creemos que no es necesario la sub-clasificación de la misma para proveer diferentes comportamientos de propagación de eventos (tal como se menciona en [Fortier et al, 2007]), sino que la responsabilidad de esta clase debería ser solamente la de notificar a todos los *EventHandlers* suscriptos a ella del nuevo evento; y son estos los encargados de proveer un comportamiento diferente en cada implementación. Esta es una variación en la capa de contexto que se propone en este trabajo. A su vez, al notificar, los *AdaptationEnvironment* se pasan a sí mismos como parámetro por si algún *EventHandler* necesita de la referencia de algún otro *ContextFeature* o *AwareObject*. Esto se debe a que los *EventHandlers* no son conscientes de a qué *AdaptationEnvironment* pertenecen, lo que permite que una misma instancia de *EventHandler* se utilice en varios de estos canales.

En el modelo presentado en la Figura 16, para proveer comportamiento contextual de objetos del dominio puro de la aplicación era necesario sub-clasificar la clase *AwareObject*. Dado que en este trabajo solo consideramos contexto de los objetos del domino, decidimos en el modelo propuesto que la clase *AwareObject* tenga directamente una relacion de conocimiento al objeto del modelo al que representa. Esto requiere que los objetos de los que se desee tener comportamiento contextual implementen la interfaz *Observable*. En el caso que al desarrollador le sea más conveniente sub-clasificar en lugar de implementar los métodos de la interfaz, proveemos una clase llamada *ContextObservable* que implementa la interfaz *Observable* y tiene todos los métodos ya definidos.

Las implementaciones de las clases *EventHandler* y *MatchingPolicy* quedaran casi iguales a lo presentado en [Fortier et al, 2007], tanto sea sus responsabilidades e interfaces. El único cambio propuesto es el renombre del método *canHandle()* de la clase *EventHandler* por *shouldHandle()*, para concordar con el método del mismo nombre de la clase *MatchingPolicy*. El método *handleEvent(),* presente en la clase *EventHandler,* se define usando el patrón de diseño Template [Gamma et al, 1994]. Éste ejecuta primero el método *shouldHandle()* para verificar si el evento debe ser manejado por la instancia, el cual delega la decisión en la instancia de *MatchingPolicy* asociada con ese *EventHandler*. Luego, en caso positivo, ejecuta el método abstracto *doHandle*() que contiene una acción diferente en cada sub-clase.

Otra clase que se encontraba en el modelo original y es redefina en el modelo propuesto es la clase *ContextEvent*. Esta clase se responsabiliza de representar un evento que haya sucedido en la aplicación. Para ello, es necesario sub-clasificarla por cada evento distinto que se quiera representar. Es en este punto en donde se difiere del modelo anterior. Decidimos que los eventos sean distinguibles por sub-clases en lugar de nombres debido a que estos pueden encontrarse repetidos dentro de una aplicación de contexto compleja. Si esto fuera a ocurrir no se podría distinguir entre dos eventos que semanticamente son distintos pero tienen el mismo nombre. Se definió en el constructor de la clase que se le otorguen tanto el objeto que originó el evento, como la característica o valor modificado, para ser almacenados y recuperados por cualquier objeto que le interese saber los detalles del mismo.

En la Figura 17 se pueden apreciar los cambios mencionados anteriormente, y la diferencia respecto del modelo usado como base presentado en la Figura 16. Notar que lo respectivo a la clase *ContextFeature* se tratará más adelante ya que requiere un análisis más extenso, por esta razón todavia no se presenta en la Figura 17.

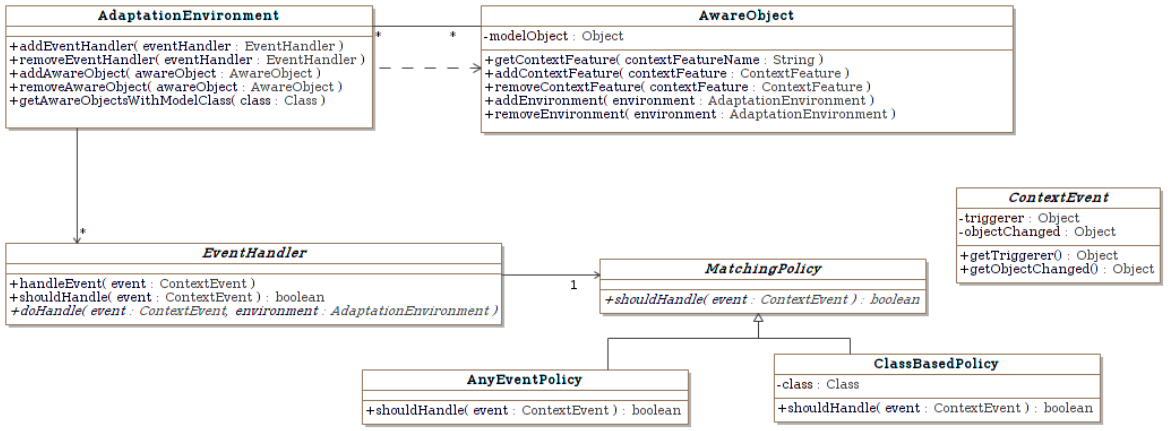

Figura 17: Modificaciones realizadas al modelo de contexto de la Figura16

La clase *ContextFeature* del modelo de base (Figura 16) se responsabiliza de almacenar los valores de todas las características del dominio de la aplicación para poder disparar un evento ante el cambio de las mismas. Creemos que esta visión es muy extremista dado que es posible que el desarrollador que utilice el modelo propuesto puede querer adaptar una aplicación ya existente para contextualizar ciertos aspectos, pero es mucho el trabajo que requiere extraer las características deseadas para llevarlas a la capa de contexto. Por esto, se ha decidido crear tres sub-clases diferentes de *ContextFeature: ValueHolderContextFeature*, *ObserverContextFeature* y *CompositeContextFeature*. La primera es similar a la ya propuesta en el modelo base (Figura 16), la instancia almacenará el valor de una característica del dominio y alertará ante cualquier cambio que se produzca, pero, utilizaremos ésta sub-clase sólo para aquellas propiedades de los objetos que son puramente contextuales y que pueden ser agregadas o removidas dinámicamente. Por ejemplo, si se requiere considerar el posicionamiento del usuario como un contexto relevante, existirá un *AwareObject* (representando al usuario) que conocerá a un *ValueHolderContextFeature*, el cual tendrá el valor de la posición del usuario en algún formato particular. El valor de esta posición puede ser obtenido de diferentes maneras, por ejemplo utilizando un GPS o un lector de QR. Así, el dominio sólo es consciente de que existe un servicio de posicionamiento pero se abstrae de la manera en la que está implementado. A su vez, si en cualquier momento se quiere dejar de monitorizar el posicionamiento del usuario es tan simple como remover el *ValueHolderContextFeature* de la colección del *AwareObject* que representa a dicho usuario.

La segunda sub-clase propuesta como se menciono anteriormente la denominamos *ObserverContextFeature*. Como su nombre lo sugiere, ésta sub-clase se encarga sólo de observar a las clases del dominio y disparar un evento cuando la característica observada cambia. La

diferencia con la anterior (*ValueHolderContextFeature*) reside en que el manejo de esa propiedad es puramente exclusivo del dominio y, por lo tanto, la aplicación debe contener toda la lógica referida a la representación de la característica, su almacenamiento y cuándo debe ser modificada. Así, podemos contextualizar aquellas propiedades que pueden ser útiles para el contexto de la aplicación pero que son parte esencial de la lógica de la misma, como también, en el caso de aplicaciones implementadas previamente a la utilización de esta capa, contextualizar propiedades haciendo pequeñas modificaciones a la clase que las contiene.

Por último se tiene una tercera sub-clase llamada *CompositeContextFeature* que no tiene ningún comportamiento especial salvo el de agrupar a varios *ContextFeature* bajo uno solo. Como su nombre lo indica esta clase es implementada siguiendo el patrón de diseño Composite [Gamma et al, 1994]. Una de los posibles usos de esta clase es, por ejemplo, disparar un evento especial cuando otros dos *ContextFeature* hayan disparado sus propios eventos en un margen de tiempo relativamente cercano. Esta y otras funcionalidades posibles pueden ser implementadas al sub-clasificar la clase *CompositeContextFeature* provista en el modelo propuesto en esta sección.

Las sub-clases de *ContextFeature* mencionadas anteriormente se pueden apreciar en la Figura 18.

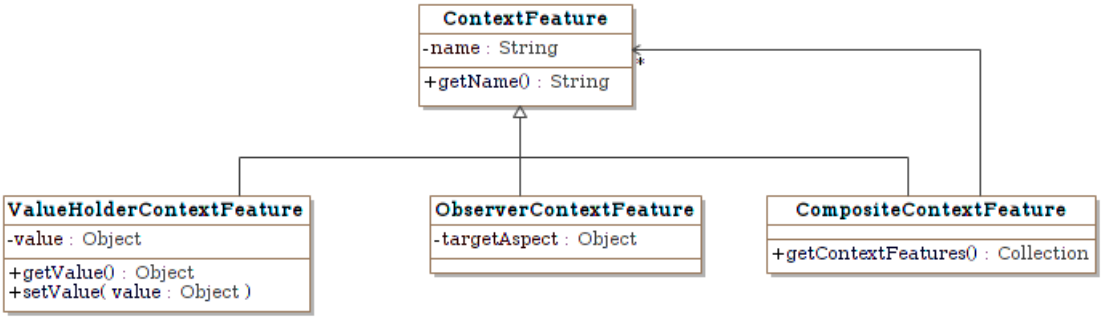

Figura 18: Jerarquía de ContextFeature propuesta

Para complementar a la clase *ContextFeature* y poder crear los eventos correspondientes a cada cambio, introducimos una nueva clase llamada *ContextEventFactory* que respeta el patrón de diseño *Abstract Factory* [Gamma et al, 1994]. Esta nueva clase es la encargada de instanciar una sub-clase de *ContextEvent* en particular a partir del objeto que disparó el cambio y la característica cambiada. De esta forma la construcción de eventos es delegada en estas fábricas permitiendo que los *ContextFeature* pasen de ser un hot-spot de caja blanca a uno de caja negra, como era el caso del modelo de base. Supongamos que existen un *ValueHolderContextFeature* y un *ObserverContextFeature* en donde ambos tienen que disparar el mismo evento. En este caso, los dos *ContextFeature* conocen al mismo *ContextEventFactory*.

Estas clases se pueden apreciar en la Figura 19.

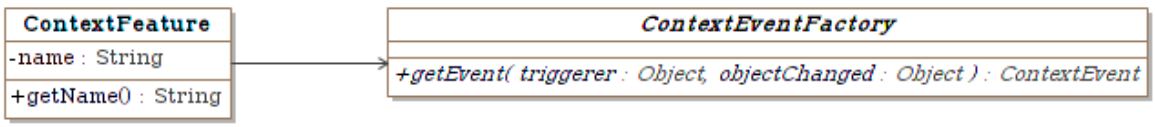

Figura 19: Clase ContextEventFactory

El modelo propuesto para de la *Capa de Contexto* se puede apreciar en la Figura 20. Se pueden apreciar todas las clases mencionadas anteriormente y la relación entre las mismas. De esta manera, queda presentada una capa de contexto general que puede ser utilizada para cualquier dominio particular.

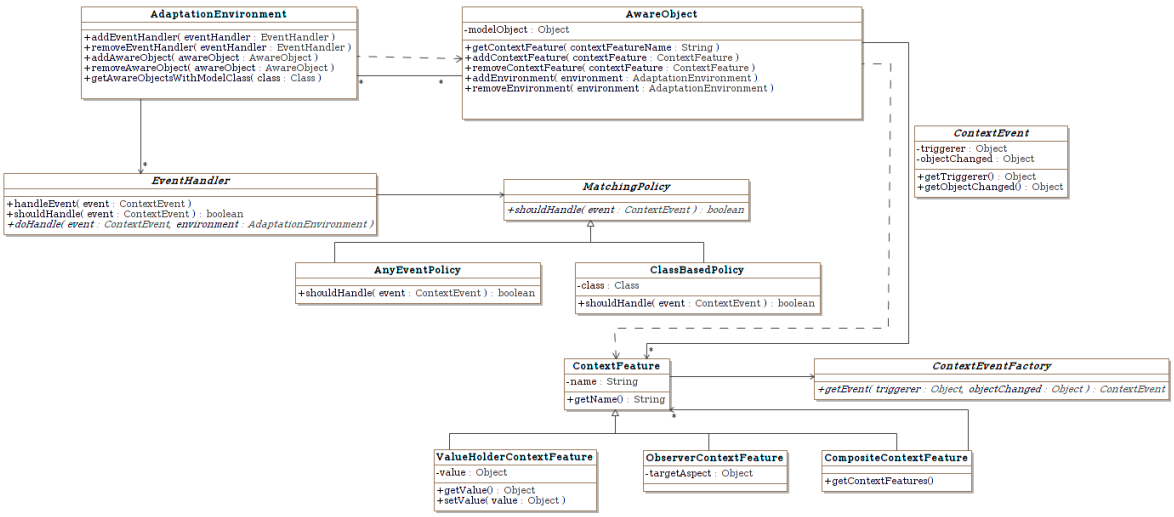

Figura 20: Modelo de contexto completo

Notar que en el Anexo A se describe cómo usar, de manera general, la *Capa de Contexto* definida (Figura 20).

3.3.1 Funcionamiento del Mecanismo de Eventos

Una vez presentada nuestra Capa de Contexto (modelo mostrado en la Figura 20), pasaremos a explicar con detalle el funcionamiento de la misma desde el momento en que se dispara un nuevo evento, la interacción de las distintas clases de la capa de contexto, y cómo se logra ejecutar la acción deseada para dicho evento.

El primer paso en la vida de un evento es su creación. Un nuevo evento es creado cuando un valor de alguna característica de contexto cambia. Es decir, al cambiar un *ContextFeature* se crea un evento. Acorde al modelo de contexto propuesto en esta tesis, pueden suceder dos situaciones:

- se modifica el valor almacenado por un *ValueHolderContextFeature*,
- o bien se modifica una característica representada por alguna clase del dominio observada por un *ObserverContextFeature*.

Sea cual sea la razón, ambas clases, al ser sub-clasificaciones de *ContextFeature*, actúan de la misma manera. En primer lugar le piden a la instancia de *ContextEventFactory*, que tienen almacenada desde su creación, que les provea de un nuevo evento a partir del objeto que lo disparó (esto es, él mismo en el caso del *ValueHolderContextFeature* y el objeto observado en el

caso del *ObserverContextFeature*). Este evento además conoce el objeto o característica que fue modificada. En este instante, la instancia de *ContextEventFactory* creará un evento dependiendo de su implementación. Cabe recordar que la clase *ContextEventFactory* es abstracta, como también lo es el método que se encarga de crear un nuevo evento, por lo que cada sub-clase producirá un evento diferente; siempre y cuando se haya instanciado correctamente el *ContextFeature* y coincidan las clases del objeto modificado y el disparador con los requeridos por la sub-clase. En caso contrario se levantará una excepción. Esto es responsabilidad del que instancia nuestro modelo.

En segundo lugar, el *ContextFeature* notifica al *AwareObject* que lo observa de este nuevo evento. A continuación, en las Figuras 21.a y 21.b, se pueden apreciar diagramas de secuencia de la interacción de estas clases como ayuda visual a la explicación anterior. En ambas figuras se hace uso de un ejemplo donde se representa el lugar donde está posicionado el usuario. En la Figura 21.a los cambios de ubicación son notificados por la propia instancia de *User* y observados por una instancia de *ObserverContextFeature.* En cambio, en la Figura 21.b, el valor de la ubicación del usuario se encuentra almacenado en una instancia de *ValueHolderContextFeature.* Se puede observar que ante el cambio de posición ambos *ContextFeature* delegan en sus *ContextEventFactory* (el cual, en este caso, son el mismo) la creación de un nuevo evento, el cual es pasado a sus *AwareObject.* Los métodos *update()* y *handleEvent()* representan de forma simplificada todo el manejo de notificaciones por el patrón *Observer* que se encuentra asociado a la invocación de ambos métodos.

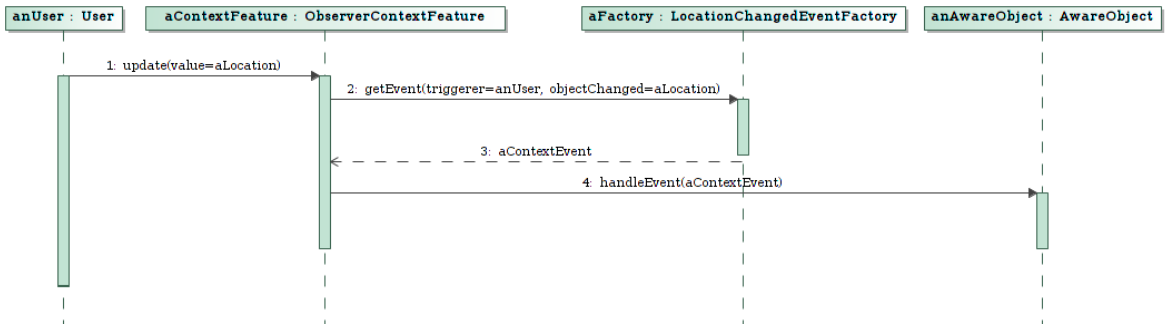

Figura 21.a: Secuencia de interacción desde un *ObserverContextFeature a su AwareObject*

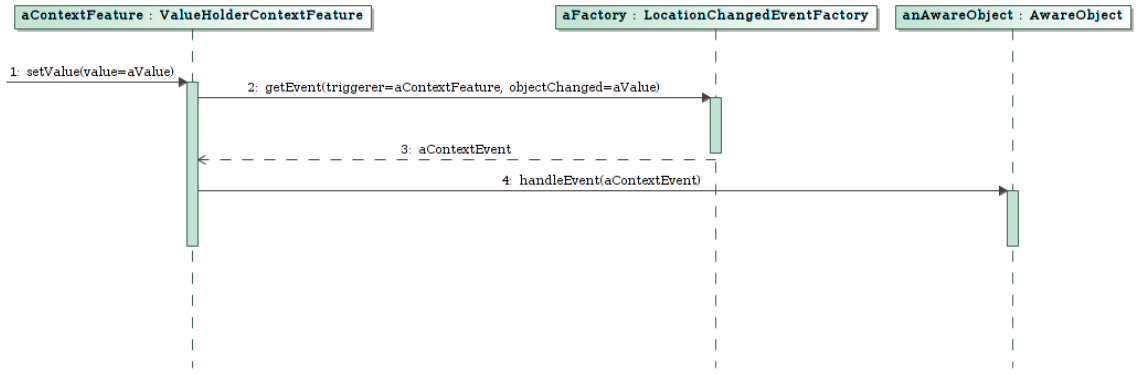

Figura 21.b: Secuencia de interacción desde un *ValueHolderContextFeature* a su AwareObject

La clase *AwareObject*, luego de recibir un nuevo evento de uno de sus *ContextFeature*, lo distribuirá a los *AdaptationEnvironment* a los que esté suscripto. Estos *AdaptationEnvironment* pueden verse como canales de difusión, por lo que su única responsabilidad es la de notificar a aquellos que puedan estar interesados del nuevo evento. Es decir, aquellos *EventHandlers* interesados en estos cambios. Para esto el *AdaptationEnvironment* envía el mensaje *handleEvent() a cada uno de sus EventHandlers.* El método *handleEvent()* definido en general para cualquier handler, llama a su vez a los métodos *shouldHandle()* y acorde al resultado de este último, luego llama al método *doHandle().*

El método *shouldHandle()* se encarga de decidir si es apropiado que la instancia de *EventHandler* maneje el evento que le fue pasado por el *AdaptationEnvironment*. Esta decisión es delegada a otra clase llamada *MatchingPolicy.* Cada *EventHandler* conoce a una instancia de *MatchingPolicy* particular desde el momento de su creación. Algunos ejemplos de posibles sub-clase de *MatchingPolicy* puede ser, por ejemplo, manejar cualquier evento (retornar siempre true), decidir si se debe manejar a partir de la clase del evento, o decidir a partir de la clase que disparó el evento.

El método *doHandle()* representa la acción que se quiere tomar ante el evento disparado. Como es de esperarse, el método es abstracto y cada sub-clase concreta de *EventHandler* representará una acción, acorde a cómo este definido este método.

En la Figura 22 se puede apreciar un diagrama de secuencia que continúa al presentado en las Figuras 21.a y 21.b. Es decir, se muestra la secuencia de mensajes que se desencadenan a partir del método *handleEvent()* que recibe el *AwareObject*, y que fue descripto anteriormente.

De esta manera, quedó presentado el funcionamiento general de la Capa de Contexto definida. Se puedo apreciar en los diferentes diagramas de secuencia que este flujo es general independiente del dominio para el que puede ser utilizado.

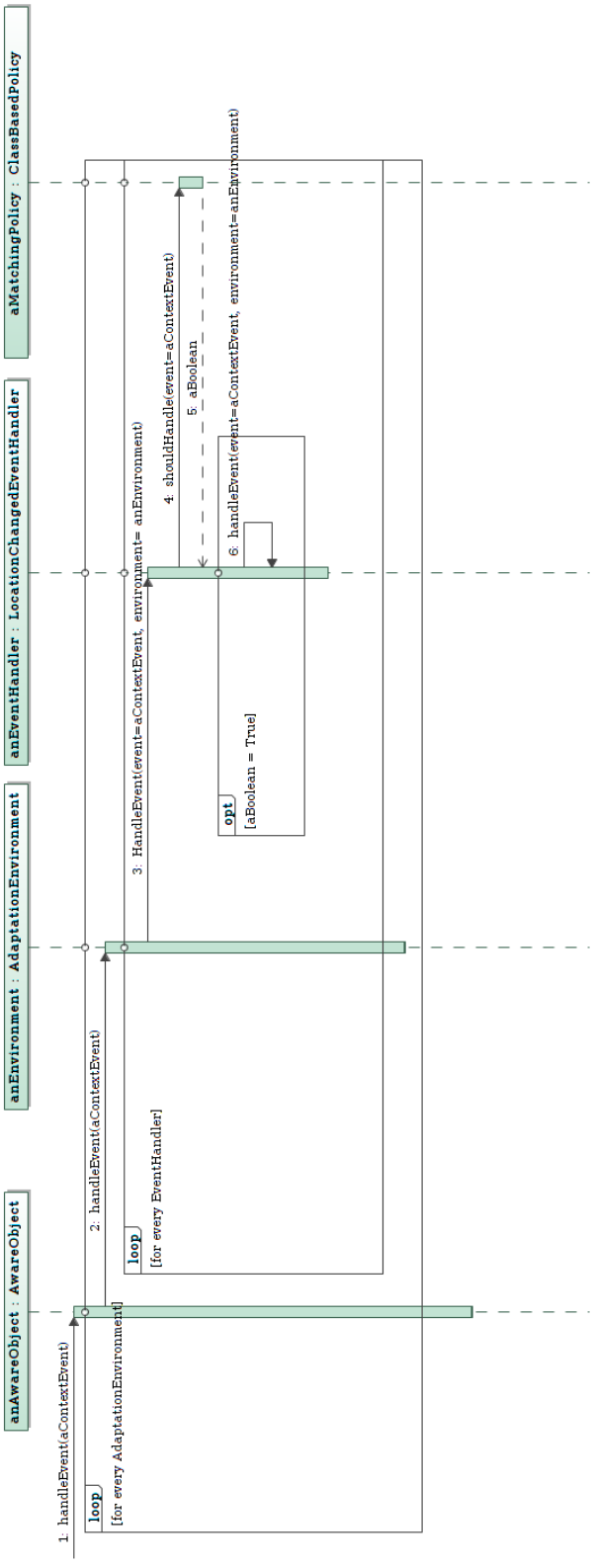

Figura 22: Secuencia generada desde el método *handleEvent*() de un *AwareObject*

3.4 Modelo de Gestos combinado con la Capa de Contexto

En la Sección 3.2 se presento un *Modelo de Gestos* y en la Sección 3.3 un *Modelo de Contexto General*. En esta sección se realizará una combinación de ambos, aprovechando así las ventajas del manejo general de contexto de nuestra capa especializada en reaccionar a los cambios contextuales. Esta combinación va a impactar en el Modelo de Gestos para lograr que el funcionamiento del mismo sea considerado desde el uso de contexto. Esto se puede realizar debido a que los distintos gestos que realice el usuario pueden ser considerados contextos relevantes, y cuando se realiza un nuevo gesto, esto puede disparar distintos eventos.

Es esta sección se mostrará las modificaciones que se debieron realizar para que el *Modelo de Gestos* de la Sección 3.2, se pueda integrar con el Modelo de Contexto General presentado en la Sección 3.3. Siendo en este caso, como se mencionó anteriormente, los gestos un dominio particular de aplicación. Los cambios que se proponen al *Modelo de Gestos* (Sección 3.2) y que serán explicados en detalle a continuación son los siguientes:

a) Conocimiento del *SkeletonManager* de la instancia del *UserManager.*
b) Conocimiento de cada instancia de *User* a su *UserGestureManager*.
c) Conocimiento de cada instancia de *User* a sus múltiples instancias de *GestureActionsAssociation*.

Cabe mencionar que estas alteraciones al *Modelo de Gestos* están hechas con dos objetivos en mente. Primero se quiere desacoplar distintos sectores de la arquitectura, por ejemplo, las clases involucradas con los sensados (*SkeletonManager* en nuestro caso) no tengan conocimiento de la existencia de usuarios o gestos en el sistema; o que las clases involucradas con los usuarios (*User* y *UserManager)* no sean dependientes del objetivo del sistema, que es la detección de gestos. En segundo lugar, que la extensión del sistema, y consecuente interacción de las clases ya existentes con las nuevas, no implique una modificación del código de las primeras.

Pasaremos entonces a explicar en detalle cada una de las modificaciones:

a) **Conocimiento del *SkeletonManager* de la instancia del *UserManager***

En primer lugar eliminaremos el conocimiento de la clase *SkeletonManager* de la instancia de *UserManager*. Para poder mantener la notificación de lectura de nuevos esqueletos sin este conocimiento explícito se utilizará un evento de contexto específico, al que llamaremos *NewSkeletonContextEvent*. A su vez, utilizaremos un *AwareObject* que represente al objeto y un *ObserverContextFeature* que disparará dicho evento cuando la instancia de *SkeletonManager* notifique de la obtención de un nuevo esqueleto de un usuario. Es así que ya no es necesario el conocimiento por parte del *SkeletonManager* de la instancia de *UserManager* para notificar de un nuevo esqueleto, sino que el *SkeletonManager* se encargará sólo de notificar a través de la capa de contexto y, todo aquel objeto que esté interesado en estos eventos, serán alertados con la inclusión de un *EventHandler* específico. En el caso del aviso a la clase *UserManager* se realizará a través de la implementación de la

clase *NewSkeletonHandler* que es sub-clase de *EventHandler* y que, al recibir una instancia de *NewSkeletonContextEvent*, notificará a la instancia de *UserManager* para determinar el usuario al que pertenece el esqueleto.

En la Figura 23 se puede apreciar estos cambios y las nuevas clases propuestas. Se puede ver en la parte de arriba de la figura las clases relacionadas al contexto, se observan las extensiones realizadas, por un lado el *NewSkeletonHandler* y por otro el *NewSkeletonContextEvent*. En la parte inferior de la figura se puede observar las clases del dominio de gestos, donde ya no hay relación en ese nivel entre *SkeletonManager* y *UserManager.*

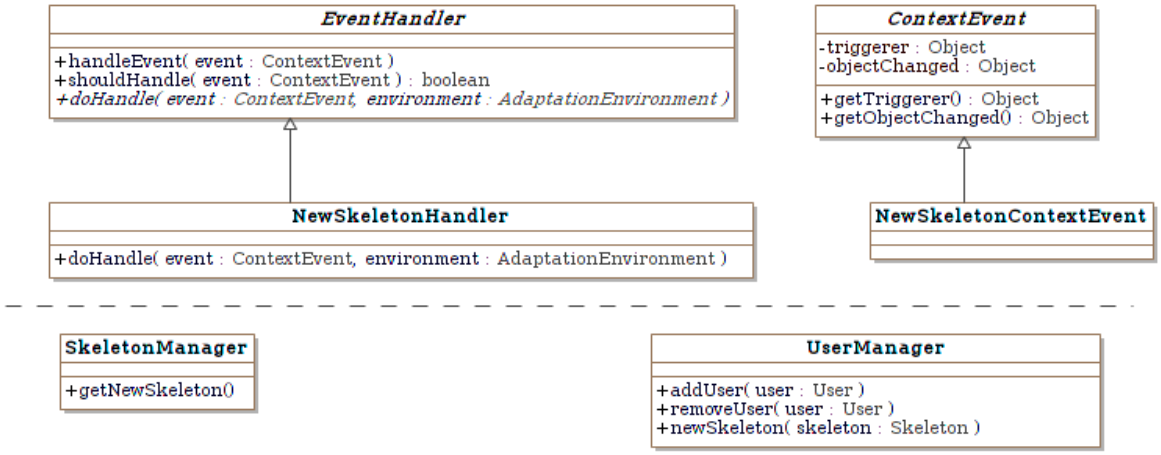

Figura 23: Primera modificación del Modelo de Gestos para combinarse con la Capa de Contexto general

En la Figura 24 se puede apreciar un diagrama de instancias que refleja los cambios propuestos en la Figura 23, permitiendo apreciar cómo se da la relación entre las clases de contexto creadas y el dominio particular de gestos. Se puede apreciar que un *AwareObject* conoce a un objeto de la clase *SkeletonManager*, a su vez este *AwareObject* tiene un *ObserverContextFeature* que conoce a esta misma instancia de *SkeletonManager.* Esto permite que cualquier cambio en la instancia de *SkeletonManager* sea detectada por el *ObserverContextFeature*, y así este generará el evento particular *NewSkeletonContextEvent.*

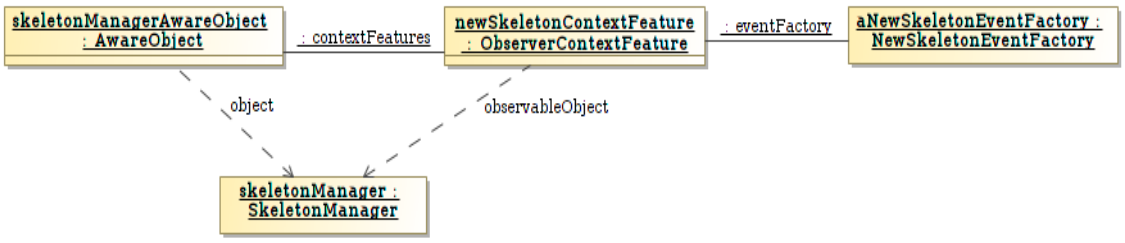

Figura 24: Modelo de Instancia que refleja la relación entre un *AwareObject* y *SkeletonManager*

b) Conocimiento de cada instancia de *User* a su *UserGestureManager*

La siguiente modificación que proponemos es la de eliminar el conocimiento de la clase *User* a su *UserGestureManager*, dado que la segunda implica la existencia de gestos y, como dijimos anteriormente, queremos desacoplar estos sectores del modelo. Sin embargo, la necesidad de la clase *UserGestureManager* para conectar los usuarios con los gestos sigue existiendo y, además, debemos mantener la relación entre cada instancia de *User* y de *UserGestureManager.* Esta asociación puede mantenerse haciendo uso de la clase *ValueHolderContextFeature* para que contenga una referencia a una instancia de *UserGestureManager*. Este *ValueHolderContextFeature* será agregado a la colección de *ContextFeatures* del *AwareObject* de la instancia de *User* correspondiente, que habremos creado con anterioridad. En la Figura 25 se puede apreciar un diagrama de instancias que refleja dicho cambio. Así, siempre que se necesite el *UserGestureManager* de un *User* en particular se podrá acceder a través del *AwareObject* de esa instancia de *User*. Para ello, se podrá solicitar a través del nombre que se haya elegido para la instancia de *ValueHolderContextFeature*.

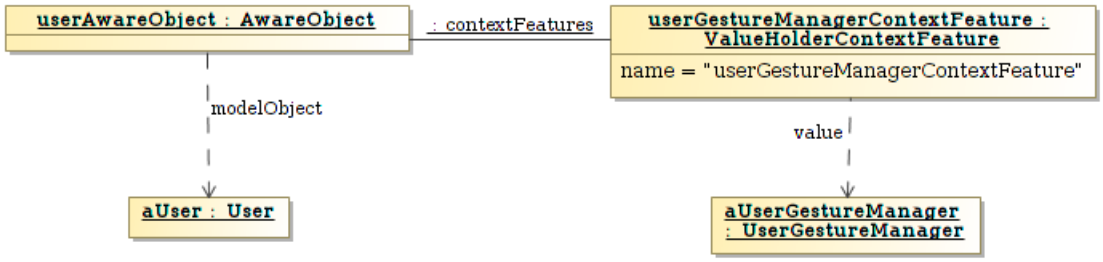

Figura 25: Relación entre el *User* y su *UserGestureManager a través de la capa de contexto*

Cabe destacar que ya no es necesario que las clases *UserManager* y *User* tengan el método *NewSkeleton* dado que el mismo implica un conocimiento de los aspectos relacionados a los gestos y una responsabilidad de notificar explícitamente al mismo de la presencia de un nuevo esqueleto. Esta responsabilidad puede ser transmitida al *NewSkeletonHandler* definido previamente (en el punto a, ver Figura 23). El mismo, para empezar, obtendrá una instancia de *User* que se la solicitará al *UserManager* a través del ID que se le fue asignado por el sensor. Luego, obtendrá la instancia de *UserGestureManager* a través del *AwareObject* del usuario y el *ValueHolderContextFeature* que lo representa, y se le notificará del nuevo esqueleto.

Notar, además, que la clase *UserGestureManager* debe notificar cada vez que se haya completado con éxito un gesto. Para ello cada instancia de esta clase tendrá su propio *AwareObject* y *ObserverContextFeature* que notificarán del suceso del evento. A este evento lo llamaremos *GestureRecognizedContextEvent* y existirá, a su vez, una sub-clase de *EventHandler* llamada *GestureRecognizedHandler* que se encargará de buscar las acciones correspondientes al gesto realizado y ejecutarlas. Notar que el *GestureRecognizedContextEvent* sólo tiene una referencia al gesto que se completó y a la

instancia de *UserGestureManager.* Lo que implica que durante la ejecución del *GestureRecognizedHandler* se tiene una referencia a la instancia de *UserGestureManager* que disparó el evento, pero esta no es consciente de la existencia ni de usuarios ni de acciones, por lo tanto, no puede tener una referencia directa a ninguna de ellas. No se propone agregar este conocimiento porque, como se dijo anteriormente, el objetivo es independizar estas secciones de la arquitectura para poder ser separadas en módulos distintos. Por otra parte, se puede solicitar al *AdaptationEnvironment* (en donde se encuentra la instancia de *GestureRecognizedHandler*) todos los *AwareObject* que hagan referencia a una instancia de *User*. El problema es que no se puede saber cuál de todos ellos fue el encargado de ejecutar el gesto. Es por esto que se propone que exista un *AdaptationEnvironment* [3] por cada usuario aparte del anterior que englobaba los eventos de todo el sistema. Con este cambio, existiendo una referencia del *GestureRecognizedHandler* sólo en estos nuevos *AdaptationEnvironment* de granularidad más pequeña y no en el global, y siendo el *AwareObject* de un usuario y el *AwareObject* del *GestureManager* de ese usuario las únicas instancias presentes en la colección de estos *AdaptationEnvironment (*particulares para cada usuario). De esta manera, es posible recuperar al usuario y estar seguros de que es el mismo que ejecutó el gesto. Una vez obtenido al usuario se puede acceder a los *GestureActionsAssociations* y de ellos ejecutar todas las acciones asociadas al gesto que se ejecutó.

En la Figura 26 se pueden apreciar los cambios realizados mencionados anteriormente. En la parte superior de la figura, se puede observar las clases extendidas, *GestureRecognizedHandler* y *GestureRecognizedContextEvent*. En la parte inferior, el modelo de gestos modificado permite observar que el *User* ya no conoce al *UserGestureManager*. La relación se dará a nivel de instancias en el modelo de contexto, y esto se mostrará más adelante.

[3] Cabe recordar que la utilización de *AdaptationEnvironments* de diferentes granularidades ya fue propuesta en el framework de [Fortier et al, 2007]

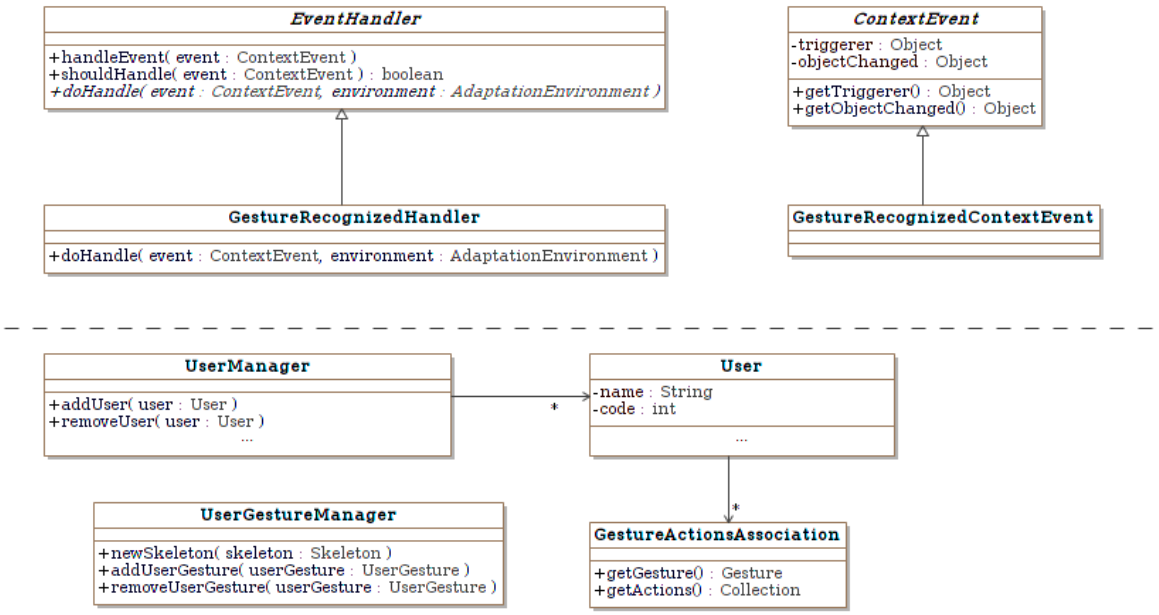

Figura 26: Segunda modificación del Modelo de Gestos para combinarse con la Capa de Contexto general

En Figura 27 se muestra un diagrama de instancias para reflejar un ejemplo de la utilización de los nuevos *AdaptationEnvironment* de granularidad más pequeña mencionados anteriormente, permitiendo así, apreciar la relación que se establece entre ambas capas, contexto y gestos. Se puede observar que se cuenta con un *AdaptationEnvironment* general que conoce a los *AwareObject* del usuario y del *UserGestureManager*. Por otro lado, se cuenta con un *AdaptationEnvironment* para un usuario particular, el cual conoce al handle *GestureRecognizedHandler.*

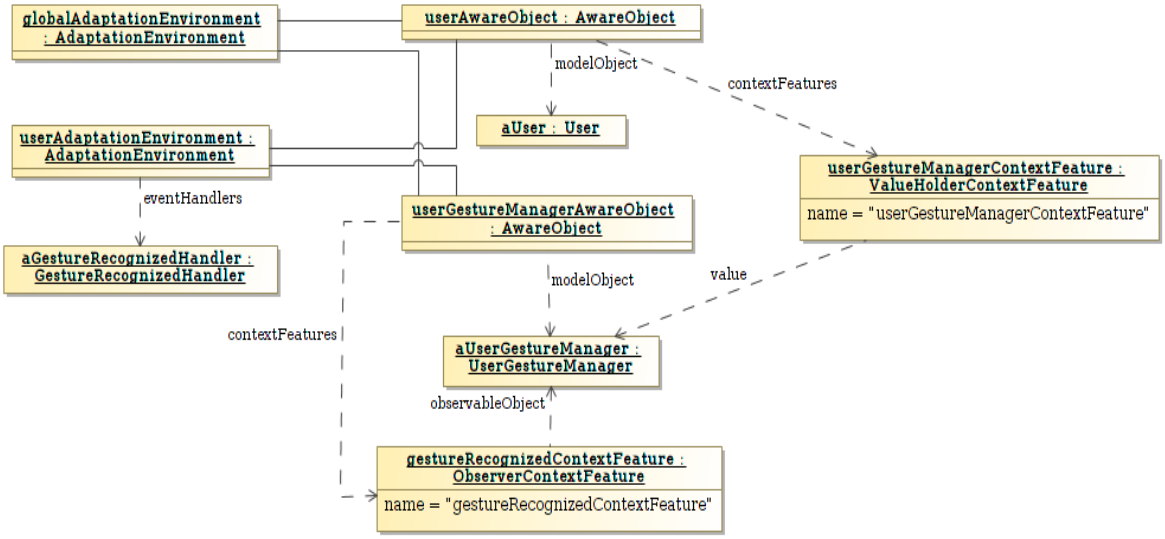

Figura 27: Diferentes instancias de *AdaptationEnvironment* una general y para un usuario

Como se pudo apreciar en la Figura 27 el *AwareObject* del *UserGestureManager* cuenta con una característica de contexto que observa al *UserGestureManager* para detectar los nuevos gestos.

c) Conocimiento de cada instancia de *User* a sus múltiples instancias de *GestureActionsAssociation*

Por último, para desacoplar totalmente los aspectos de los usuarios de los gestos y de las acciones, proponemos la eliminación del conocimiento directo de las instancias de *User* de sus múltiples instancias de *GestureActionsAssociation*. Para ello pasaremos la colección de *GestureActionsAssociations* a una colección de *ValueHolderContextFeatures* donde cada uno contendrá una referencia a uno de los anteriores y serán agregados al *AwareObject* de la instancia de *User* correspondiente. En la Figura 28 se puede observar el cambio a nivel de modelado. No hay que incorporar ningún concepto adicional al contexto, ya sean handlers o eventos, por eso solo se muestra los conceptos relacionados a los gestos.

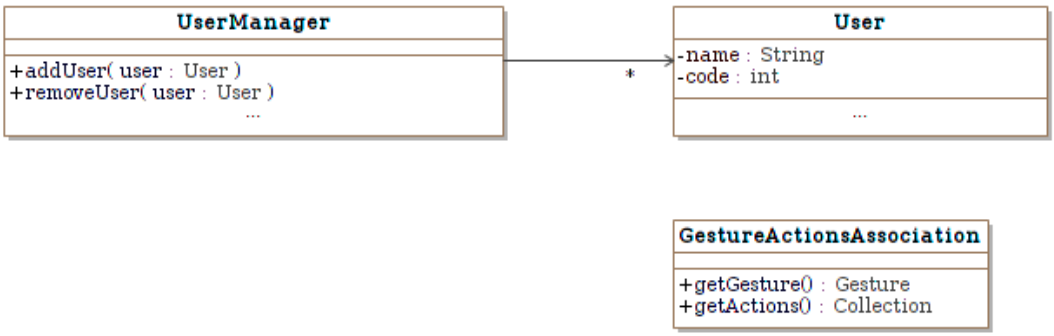

Figura 28: Tercera modificación del Modelo de Gestos para combinarse con la Capa de Contexto general

En la Figura 29 se puede apreciar un diagrama de instancias que ejemplifica este cambio propuesto en la Figura 28. Se puede observar que el *AwareObject* de un usuario tiene dos *ValueHolderContextFeatures* con dos *GestureActionsAssociations* distintos, esto permite ejemplificar que el usuario tiene configurado dos gestos con sus respectivas acciones.

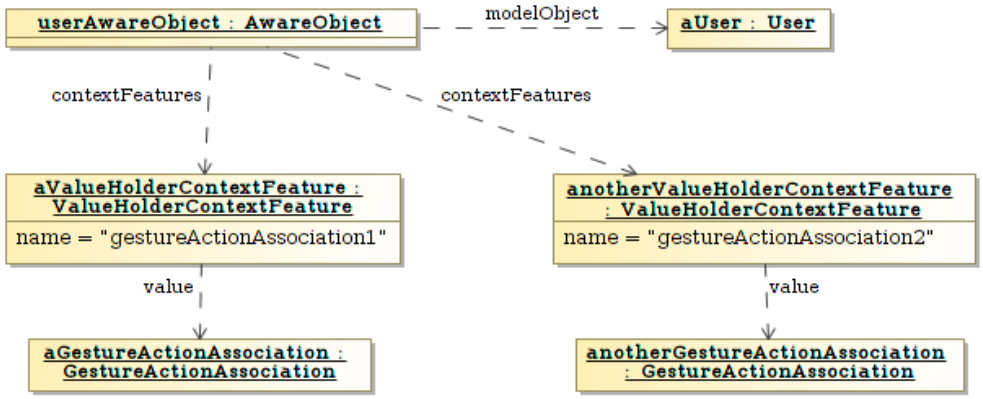

Figura 29: Dos *GestureActionsAssociation* asociados a un usuario.

La solución presentada hasta el momento en la Figura 29 genera un problema, a continuación se lo describe y se brinda una solución al mismo. Como dijimos anteriormente la clase *GestureRecognizedHandler* busca y ejecuta las acciones correspondientes del usuario una vez que se detectó la ejecución satisfactoria de un gesto, pero la instancia de *User* ya no tiene una referencia explícita a esta colección, por lo que el *handler* deberá obtener las instancias de *GestureActionsAssociation* a través del *AwareObject* del usuario. Como no se puede saber a priori cuántas de estas asociaciones tiene el usuario, puede o no existir la asociación que contenga al mismo gesto, y definir una estrategia de nombres para cada instancia y recuperarlas de una en una no resulta elegante o eficiente. Para resolver esto se propone englobar a todos estos *ValueHolderContextFeature* bajo un único *CompositeContextFeature* al que se le puede dar un nombre único y ser recuperado fácilmente. En la Figura 30 se puede apreciar el diagrama de instancias con la inclusión del *CompositeContextFeature.*

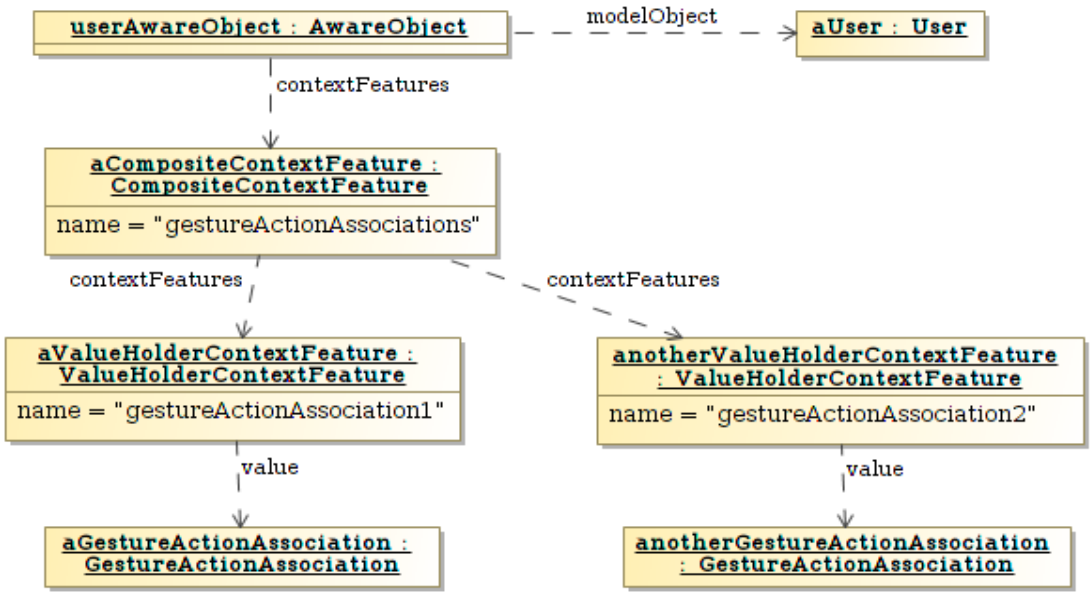

Figura 30: Incorporación de un *CompositeContextFeature* para agrupar dos *ValueHolderContextFeature*

Una vez realizados todas las modificaciones propuestas para combinar el *Modelo de Gestos* con la *Capa de Contexto* general, se presentará el modelo resultante de gestos combinación. Esto se puede apreciar en la Figura 31, se puede apreciar en la parte superior todas las clases creadas relacionadas al contexto, y en la parte inferior como quedan modelados los gestos para poderse integrar con los aspectos de contexto definidos.

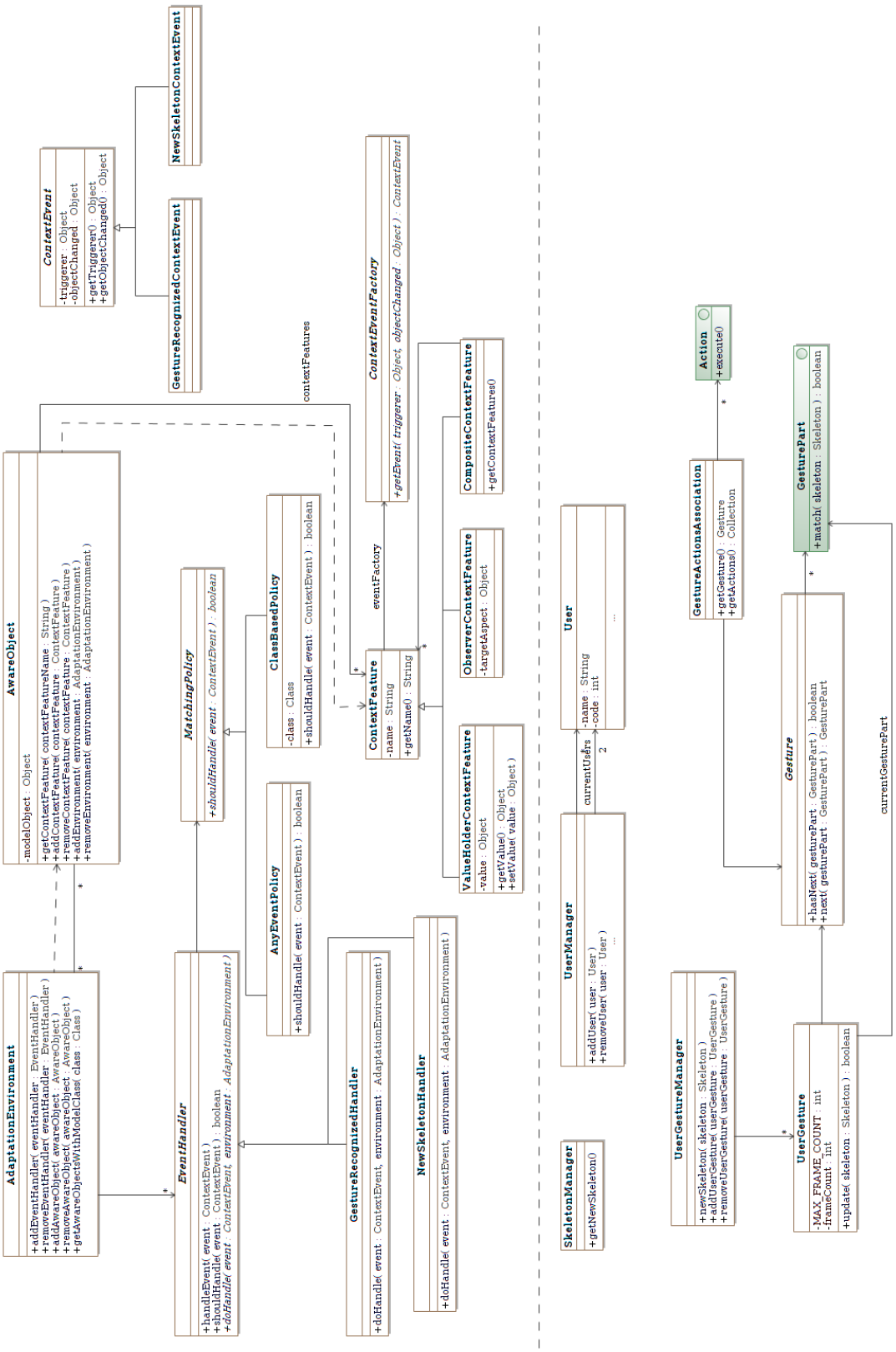

Figura 31: Combinación del Modelo de Gestos con la Capa de Contexto

El modelo presentado en la Figura 31, será el que se usará como base en el prototipo que se presenta en el Capítulo 4. Para ejemplificar de manera integral su funcionamiento en las Figuras 32.a y 32.b se puede apreciar un diagrama de secuencia completo que muestra la interacción de las diferentes clases ante la realización con éxito de un gesto.

Comenzamos con la ejecución a partir de la detección del completado con éxito de un gesto por parte de un *UserGestureManager.* El mismo notificará de dicho evento a todos sus *observers,* entre los cuales existirá una instancia de *ObserverContextFeature* que estará detectando nuevos cambios específicos, en este caso, nuevos gestos. Este *ObserverContextFeature* solicitará la creación de un nuevo evento a su instancia de *EventFactory* que le proveerá de un nuevo evento de la clase *GestureRecognizedContextEvent.* Luego, pasará este evento al *AwareObject* de la instancia de *UserGestureManager*. Todo esto ya fue descripto y explicado de forma general en la Figura 21.a dado que es el funcionamiento normal de un *ObserverContextFeature*, por lo que se decidió obviarlo en el siguiente diagrama para poder concentrarnos en el resto de la secuencia. Así, en la Figura 32.a, comenzamos con el *AwareObject* de la instancia de *UserGestureManager* propagando el evento de nuevo gesto en todos los *AdaptationEnvironments* en los que se encuentra. Puntualmente nos interesa el *AdaptationEnvironment* específico del usuario que realizó el gesto dado que en él se encuentra el *GestureRecognizedHandler.* Este último corroborará que el evento tiene que ser manejado por él, preguntándoselo a su *MatchingPolicy*, y luego solicitará al *AdaptationEnvironment* el *AwareObject* del usuario, al que le pedirá el *CompositeContextFeature* que contiene todos los *ValueHolderContextFeature* referentes a los *GestureActionsAssociation* de ese usuario.

En la Figura 32.b retomamos con la ejecución del *GestureRecognizedHandler* que tomará todos los *ValueHolderContextFeatures* del *CompositeContextFeature* y obtendrá de cada uno la instancia de *GestureActionsAssociation* que contienen. Luego, si el gesto del evento coincide con el gesto del *GestureActionsAssociation* se ejecutarán todas las acciones relacionadas.

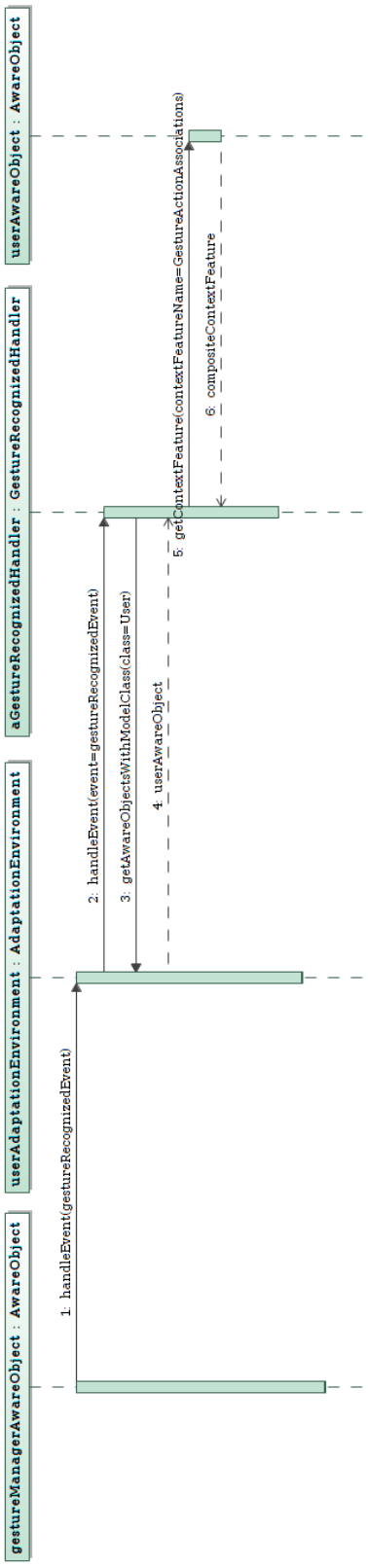

Figura 32.a: Primera parte secuencia de nuevo gesto

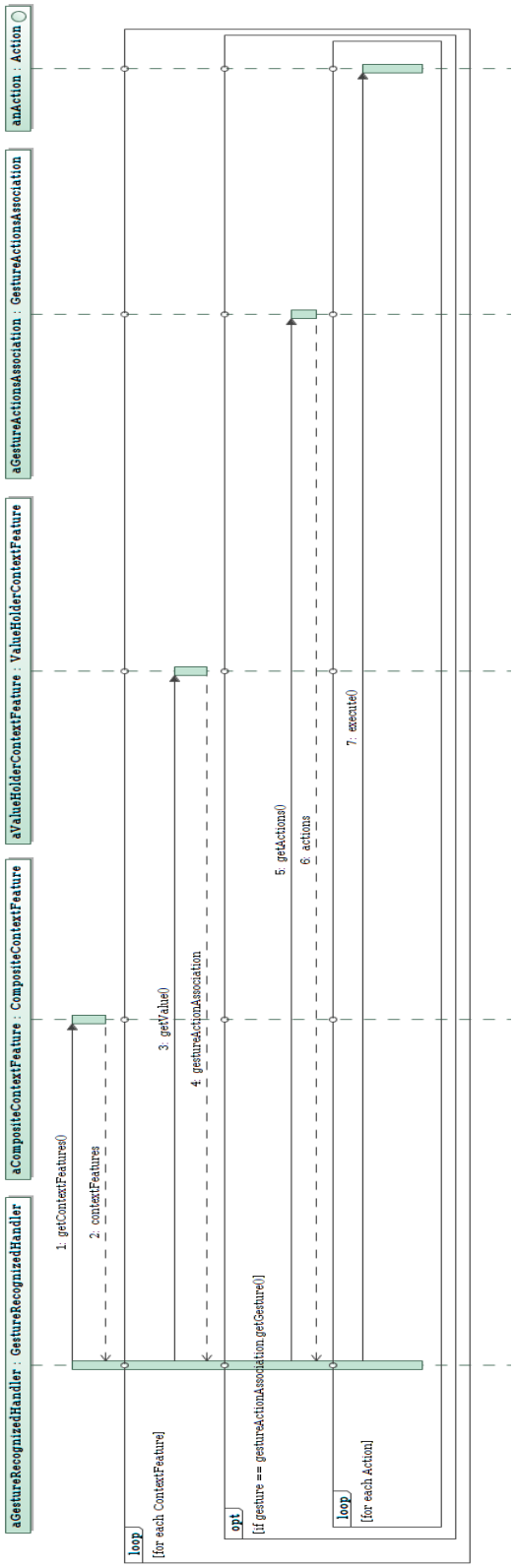

Figura 32.b: Segunda parte secuencia de nuevo gesto

4. Implementación del prototipo

En este capítulo describiremos todo lo referente a la implementación del prototipo. Daremos una explicación de la idea y modelado del mismo, y cómo hacer uso del *Modelo de Gestos* y de la *Capa de Contexto* descriptas en la Sección 3.4, así como también las problemáticas y cuestiones interesantes que se encontraron durante el desarrollo.

La idea del prototipo consiste en una aplicación de escritorio que, utilizando el Kinect, uno o dos usuarios a la vez puedan realizar gestos y un mismo gesto signifique diferentes cosas dependiendo de quién lo haga. Para que sea sencillo de ver el funcionamiento y la detección de los usuarios por parte del Kinect y que, a su vez, las acciones que representen los gestos sean notables a simple vista, se propone que el prototipo consista de un streaming de las posiciones de los esqueletos tomados por el Kinect y que los gestos realizados se vean reflejados en el display de una imagen.

Las librerías de Kinect pueden utilizarse con los lenguajes C#, C++ y Visual Basic. Para el desarrollo del prototipo se eligió C# como lenguaje de programación y se utilizó Visual Studio 2013[4] como entorno de desarrollo.

4.1 Modelado para el prototipo

Para el modelado del prototipo veremos que haremos uso de todo lo propuesto en la Sección 3.4 y extenderemos la clase *Gesture* como también implementaremos acciones específicas de nuestro dominio que implementen la interfaz *Action,* y nuevas clases que implementen la interfaz *GesturePart* para que hagan uso las sub-clases de *Gesture* que introduciremos en este capítulo. Cabe mencionar que no es necesario implementar la clase *Skeleton* dado que viene provista por la librería de Kinect y la misma es utilizada por el sensor para notificar de los cambios ocurridos.

Los gestos que implementaremos serán los de saludar con la mano derecha y saludar con la mano izquierda, *RightHandWave* y *LeftHandWave* respectivamente. La clase *RightHandWave* ya fue introducida en la Sección 3.2 como ejemplo de un gesto junto con sus partes. Recordemos que dichas partes se llamaban *RightHandWaveGesturePart1* y *RightHandWaveGesturePart2,* y la posición en la que debe estar un esqueleto para satisfacer cada una de ellas es la siguiente:

1) La muñeca de la mano derecha debe estar por encima y a la derecha del codo derecho (Clase *RightHandWavePart1*).
2) La muñeca de la mano derecha debe estar por encima y a la izquierda del codo derecho (Clase *RightHandWavePart2*).

[4] https://www.visualstudio.com/

El segundo gesto, *LeftHandWave*, también se compone de dos partes a las que llamaremos *LeftHandWaveGesturePart1* y *LeftHandWaveGesturePart2*, y las posiciones para satisfacerlas son las siguientes:

1) La muñeca de la mano izquierda debe estar por encima y a la izquierda del codo izquierdo (Clase *LeftHandWavePart1*).
2) La muñeca de la mano izquierda debe estar por encima y a la derecha del codo izquierdo (Clase *LeftHandWavePart2*).

Dado que estos gestos tienen sólo dos partes y es posible que se completen por un movimiento del usuario sin que fuera la intención de este, es que se decidió que ambos gestos consten de tres repeticiones de su correspondiente par de partes. Al concluir las repeticiones es que se considera que dicho gesto esta completo, y así se ejecuta la acción asociada al mismo.

En la Figura 33 se pueden apreciar las nuevas clases introducidas mencionadas anteriormente, las sublases de *Gesture* como así también partes de un gesto (que implementan la interfaz *GesturePart*)

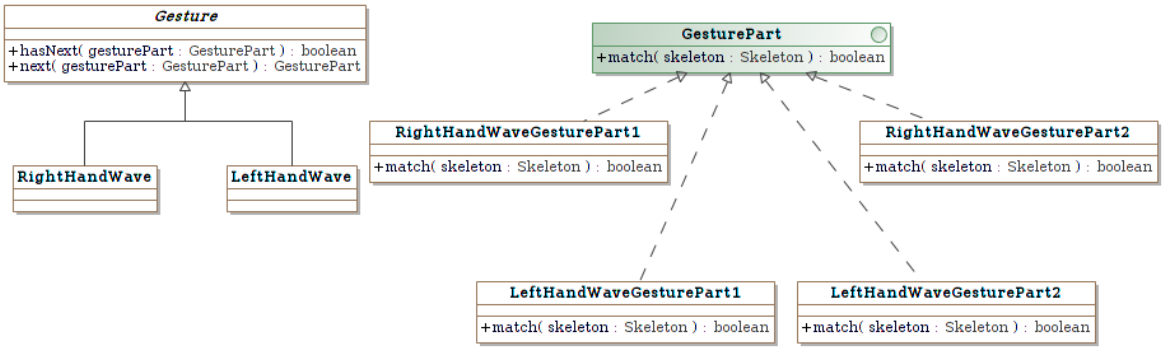

Figura 33: Gestures y GestureParts introducidos para el prototipo

Las otras clases que introduciremos son aquellas que implementan la interfaz *Action.* Creamos una por cada gesto y por cada usuario que puede estar activo en el sistema dando un total de cuatro acciones. Como dijimos en la introducción de esta sección, estas acciones involucran un cambio en el *display* de una imagen que estará visible a los usuarios, por lo que proponemos las siguientes:

1) Rotar la imagen a la izquierda (Clase *RotateImageLeftAction)*
2) Rotar la imagen a la derecha *(*Clase *RotateImageRightAction)*
3) Agrandar la imagen *(*Clase *ZoomInImageAction)*
4) Achicar la imagen *(*Clase *ZoomOutImageAction)*

Las nuevas acciones mencionadas anteriormente se pueden apreciar en la Figura 34.

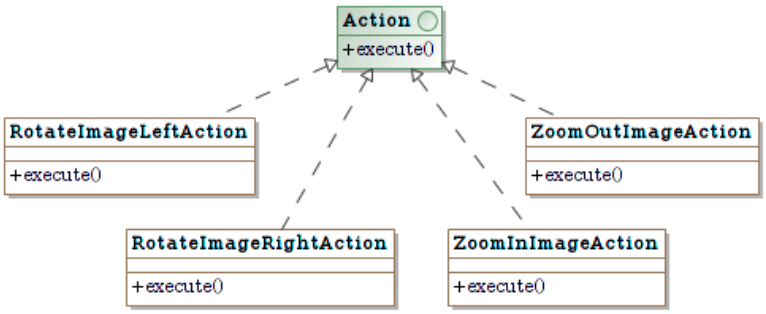

Figura 34: *Actions* introducidas para el prototipo

En la Figura 35 mostramos un diagrama de instancias que muestra cómo se relacionan estas acciones con los diferentes usuarios en el prototipo.

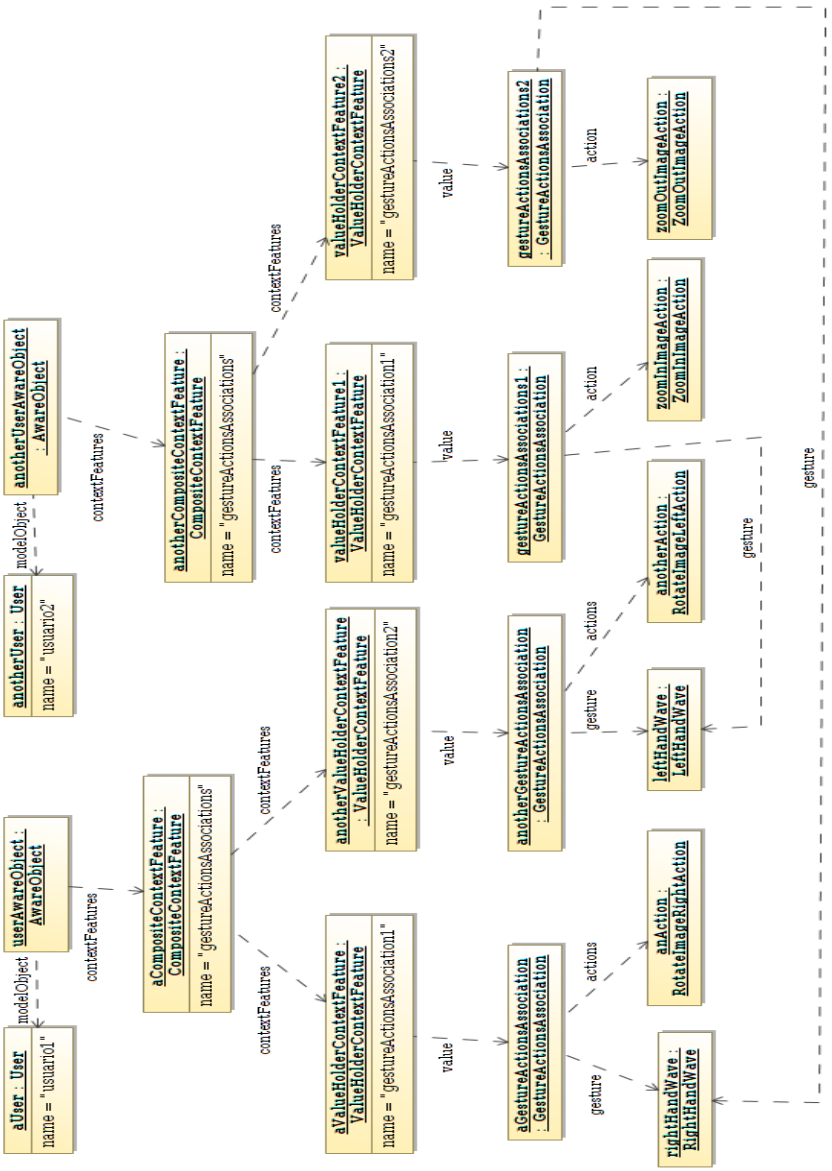

Figura 35: Instancias de dos usuarios con sus acciones para cada gesto

Se puede observar en la Figura 35 que al *usuario1* se le asignaron las acciones referentes a la rotación de la imagen, mientras que al *usuario2* se le asignaron las acciones referentes al cambio del tamaño de la imagen.

En la Figura 36 se puede apreciar un diagrama de instancias que muestra parte de las instancias referentes a contexto del prototipo. Especialmente los diferentes *AdaptationEnvironments*, los *AwareObjects* y *EventHandlers* que están asociados a estos.

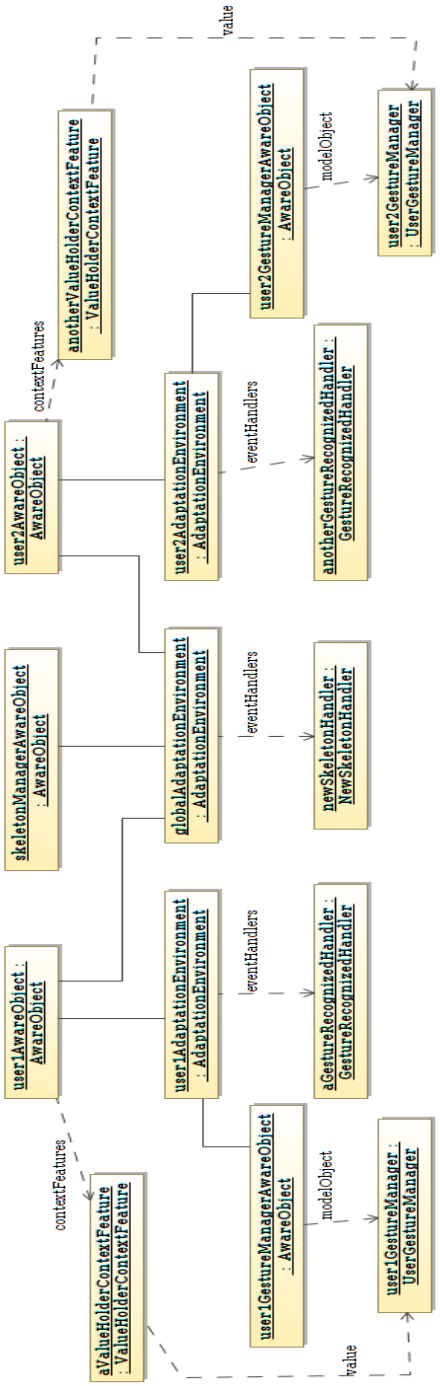

Figura 36: Instancias de contexto del prototipo

4.2 Interfaz del prototipo

En esta sección mostraremos cómo interactúan los usuarios con el sistema y qué es lo que se le muestra a los mismos. La interfaz del prototipo se divide en dos secciones. En la primera se muestra un dibujo de los esqueletos detectados por el Kinect sobre un fondo negro. En la Figura 37 se puede apreciar la detección de una única persona, mientras que en la Figura 38 se puede apreciar cómo se detectan dos personas distintas.

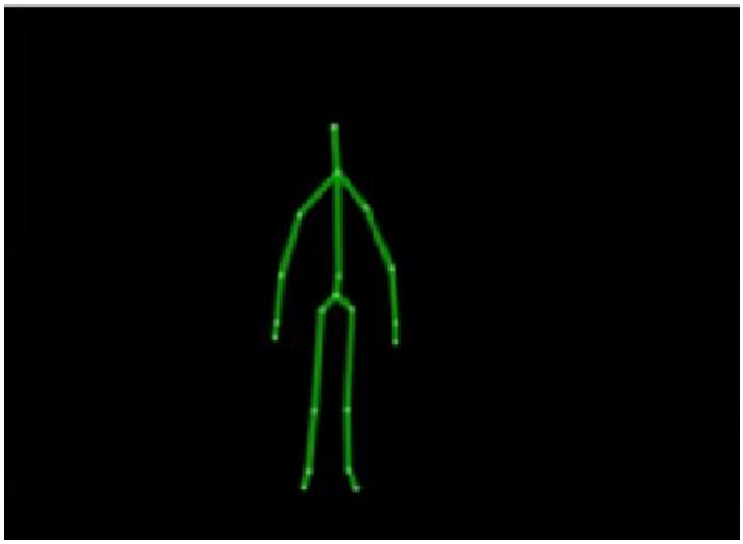

Figura 37: Captura de pantalla de un esqueleto

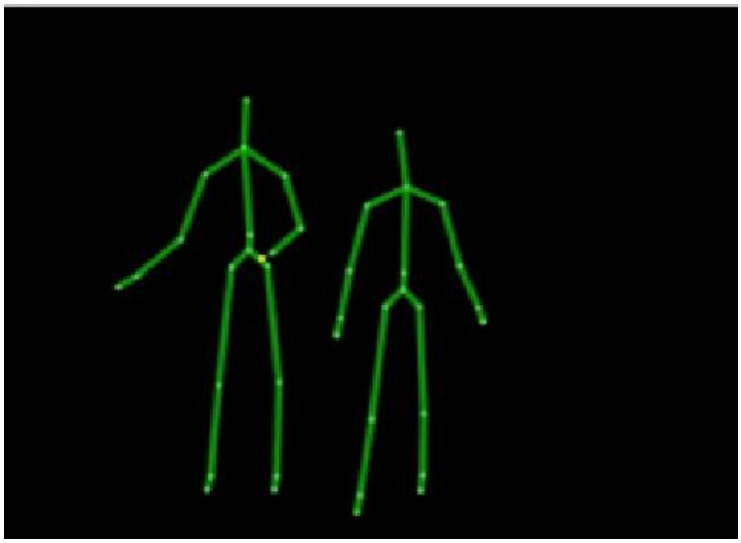

Figura 38: Captura de pantalla de dos esqueletos

Cabe mencionar que la visualización de los esqueletos mostrados en las figuras anteriores no viene provista por el Kinect, sino que es responsabilidad del desarrollador tomar los datos sobre los esqueletos y generar figuras a partir de ellos. En la Sección 4.3 entraremos en detalle sobre esta cuestión.

La segunda sección de la interfaz es la que muestra la imagen con la que los usuarios interactúan. Es decir, es aquella que se verá afectada (en zoom o rotación) acorde a los gestos que realicen los usuarios. La misma se puede apreciar en la Figura 39.

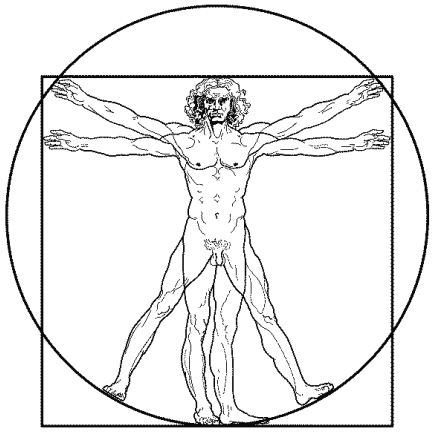

Figura 39: Imagen con la que interactúan los usuarios

4.3 Problemáticas y cuestiones interesantes para discutir

En esta sección comentaremos acerca de los problemas que surgieron durante el desarrollo del prototipo y de algunas cuestiones interesantes que quisiéramos resaltar del mismo.

A) *AdaptationEnvironment* de granularidad más pequeña

Durante el desarrollo del prototipo, y en particular, la implementación de la propagación del evento de nuevo gesto encontramos que, durante la ejecución del handler *GestureRecognizedHandler,* no teníamos referencia al usuario que había realizado dicho gesto. Como el lector podrá recordar, en la Sección 3.4.b ya se presentó este problema y se propuso una solución para el mismo, la cual consistía en tener un *AdaptationEnvironment* por usuario y cada uno de estos ambientes debe contener a una instancia de *GestureRecognizedHandler*. Originalmente esto no estaba contemplado cuando se propuso el modelo y tuvo que ser agregado al mismo una vez que se detectó durante la implementación.

B) Asociación de identificadores de esqueletos a usuarios

Uno de los primeros problemas que se encontró durante el desarrollo del prototipo fue asignar cada uno de los esqueletos detectados por el Kinect a uno de los usuarios. Para ello se utilizó el identificador que asigna el sensor a los esqueletos que detecta y que se encuentra presente dentro de la clase *Skeleton.* Luego de que el Kinect detecta

a un usuario por primera vez y le asigna un identificador a su esqueleto, le asignará siempre el mismo identificador a cada uno de los esqueletos que genere para dicho usuario. Es por esto que es posible utilizar dichos identificadores para mapearlos con los usuarios del prototipo.

Lo más adecuado es que el responsable de este mapeo sea el *NewSkeletonHandler* dado que al mismo le llegan todos los esqueletos detectados y es capaz de recuperar todos los usuarios a través del *UserManager.* Los otros dos candidatos eran el mismo *UserManager* y el *SkeletonManager*, pero esto no es recomendable dado que la asignación de esta responsabilidad implica, implícitamente, un conocimiento entre ambas secciones de la arquitectura; cuestión que queríamos evitar al momento de combinar el *Modelo de Gestos* con la *Capa de Contexto*.

Otra problemática interesante referente al mapeo de esqueletos a usuarios es la asignación de un identificador diferente al mismo usuario una vez que este dejó de ser detectado y luego volvió al campo de visión del Kinect. Para resolver esto se planteó una estrategia que consiste en asignar el identificador al primer usuario que esté logueado pero que todavía no tenga asignado un identificador. Luego, si se recibe una secuencia de esqueletos donde ninguno de ellos tiene el identificador asignado a un usuario, se puede deducir que el usuario salió del campo de visión y, si vuelve a ser detectado, se le asignará un identificador diferente. Es por esto que es seguro romper el mapeo que se tenía para dicho usuario así se le asignará un nuevo identificador al momento de detectarse nuevamente su esqueleto.

Por ejemplo, supongamos que el *usuario-1* y el *usuario-2* se encuentran logueados en el sistema. Ahora supongamos que hay una única persona que está siendo detectada por el Kinect con un identificador X de esqueleto, por lo tanto, se asumirá que es el *usuario-1* y que los esqueletos de ese usuario son aquellos con identificador X. Si la persona sale del rango de visión y vuelve a entrar, el Kinect le dará otro identificador Y a esa misma persona. Como el *usuario-1* tenía asignado el identificador X, el prototipo supondrá que esta persona es el *usuario-2*. Luego, como se recibirán una secuencia de esqueletos con identificador Y y ninguno con identificador X, se asumirá que el *usuario-1* salió del rango de visión y se romperá el mapeo de dicho usuario con el identificador X. Liberando así este identificador para cuando el Kinetic detecte un nuevo esqueleto.

C) *EventFactory* de propósito general

Durante el desarrollo del prototipo se ha detectado que la mayoría de las sub-clases de *EventFactory* tienen código similar, y lo único que las diferencia son los tipos de *ContextEvent* que instancian y los tipos de los parámetros que le son pasados. Para hacer más sencillo el desarrollo se ha creado un *GeneralEventFactory* que es capaz de instanciar cualquier *ContextEvent* siempre y cuando lo único que deba hacer es llamar al constructor de la clase con los dos parámetros que le son pasados. Esto no podría ser posible si no se hiciera uso de la técnica conocida como *reflection* que viene

provista por algunos lenguajes como C# (en el que se desarrolló el prototipo), Java o SmallTalk. Por lo tanto, a toda instancia de *GeneralEventFactory* se le debe proveer la clase de *ContextEvent* que va a crear y las clases de sus parámetros, y así podrá reemplazar a las demás sub-clases de *EventFactory.*

D) Implementación de la clase *ContextObservable*

Durante el desarrollo del prototipo se notó que todas las clases del dominio que se quisiera que fueran contextualizadas debían implementar la interfaz I*Observable* para poder notificar a todos sus *observers* de los cambios que se producen en ella, más específicamente para el caso del prototipo, para notificar a las instancias de *ObserverContextFeature* que observaran a dicho objeto. Como se notó que este comportamiento no varía entre las implementaciones de cada clase se decidió implementar una clase llamada *ContextObservable* que implementa la interfaz *IObservable* con todos sus métodos.

Ahora el desarrollador de las clases del dominio tiene dos opciones, o bien implementa la interfaz *IObservable* como se proponía anteriormente a la implementación de esta nueva clase, o, si es que no tiene inconvenientes en la jerarquía de clases, definir a su clase como sub-clase de *ContextObservable* y tendrá así todo el comportamiento ya definido.

E) Utilización de la librería del Kinect

Para poder hacer uso de los datos provistos por el Kinect se deben realizar unos pasos previos en el código. Primero, al incluir la librería, se tiene la clase *KinectSensor* a la que se le puede pedir una lista de Kinects y de ellos seleccionar uno de los que se encuentren en estado *Connected.* Una vez seleccionado el Kinect a utilizar se deben habilitar las funciones que se utilizarán del mismo. En nuestro caso sería habilitar el streamer de esqueletos y luego declarar un método que será invocado por el sensor cada vez que se obtiene un nuevo esqueleto. Este método debe ser agregado a la lista de listeners del sensor para que el mismo sepa que debe invocarlo. Por último se debe iniciar el sensor llamando a su método `Start()`.

F) Dibujo del esqueleto

La librería de Kinect no provee ningún método al cual pasarle un objeto dibujable para que dibuje un esqueleto, sino que esta funcionalidad recae en el usuario de la librería. Sin embargo, la instalación de los drivers del Kinect viene con algunos programas de ejemplo para poder probar las funcionalidades del dispositivo y, a su vez, con todo el

código fuente[5]. Algunos de estos ejemplos proveen un dibujado en tiempo real de los esqueletos que toma el Kinect y, en su código fuente, se encuentran formas de cómo hacerlo. En el desarrollo de este prototipo se ha utilizado estos ejemplos para guiar la implementación de esta funcionalidad en el mismo.

La idea general de la implementación es la siguiente: dado que la clase *Skeleton* provee una posición absoluta de cada articulación del esqueleto que detecta, se puede dibujar cada una de ellas con un punto en el plano y luego, dibujar una línea entre cada par de articulaciones adyacentes para representar los huesos. A su vez, es posible que algunas articulaciones no las haya detectado, generalmente porque están obstruidas por algún objeto, y en lugar de dar la posición real de la articulación provee una estimación de dónde podría estar. Cada articulación viene acompañada junto a un enumerativo que indica si la articulación fue detectada correctamente, no fue detectada, o su posición es un valor estimativo. Con esta información se puede dibujar la misma y los huesos que surgen de ella de forma diferente para hacérselo saber al usuario.

G) Realización de gestos en simultaneo

Es posible que durante la ejecución de la aplicación un usuario realice movimientos que satisfagan dos gestos en simultáneo, por ejemplo en nuestro prototipo, el usuario podría saludar con ambas manos a la vez. Lo que provoca esto es que se ejecuten las acciones ligadas a dichos gestos en una sucesión rápida. En nuestro prototipo esto no es un problema dado que no genera efectos no deseados, pero podría llegar a ser un problema en otra aplicación que use nuestro *Módulo de Gestos.* Es por eso que se ha implementado una opción en el *UserGestureManager* que reinicia el estado de todos los gestos del usuario una vez que se ha completado con éxito uno de ellos. Esta opción es elegida por los programadores al momento de instanciar el *UserGestureManager* y puede ser modificada en cualquier momento.

H) Errores de detección del Kinect

La detección de los usuarios no es perfecta. Dependiendo de la posición del Kinect y la posición del usuario con respecto a este, es posible que no se detecte correctamente al usuario y se realicen estimaciones de las posiciones de ciertas articulaciones como mencionamos en el punto F. Generalmente esto sucede cuando el usuario está a una distancia muy corta del sensor, o no está completamente dentro del campo de visión del mismo, ya sea porque ciertas partes del cuerpo están obstruidas por algún objeto o se encuentran fuera del rango de visión de la cámara.

[5] Los ejemplos provistos por los desarrolladores de Kinect se pueden encontrar en el siguiente enlace: http://www.microsoft.com/en-us/kinectforwindows/develop/sample-code.aspx

Puede ocurrir que estas estimaciones realizadas no tengan ninguna coherencia y las articulaciones se encuentren en posiciones imposibles de alcanzar por cualquier persona. Incluso, las estimaciones varían entre frame y frame, provocando que cambien las posiciones de las articulaciones estimadas y, en consecuencia, se realice un gesto no deseado.

En la Figura 40 mostramos un ejemplo de mala estimación a causa de la cercanía del usuario con el sensor, lo cual genero una rotación de la imagen. Se puede apreciar que el esqueleto dibujado muestra que sus parten no están bien detectadas, solo las líneas verdes están detectadas, el resto es estimativo.

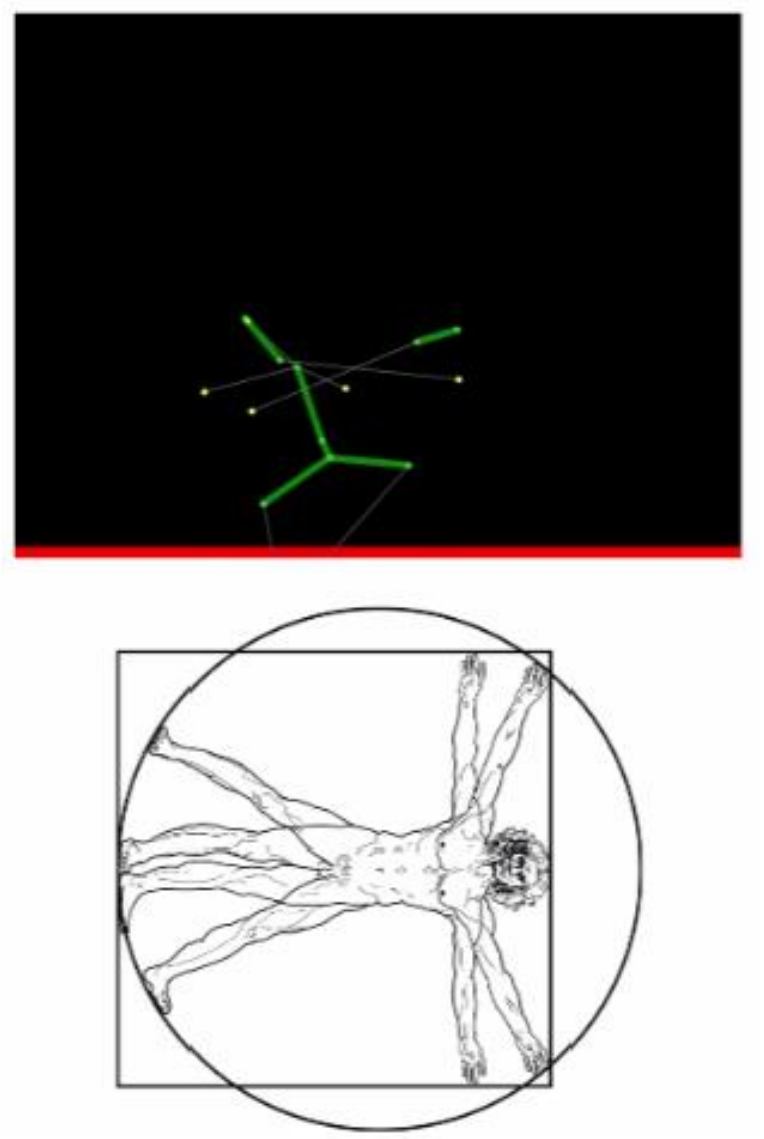

Figura 40: Malas estimaciones debido a cercanía del usuario respecto del Kinect

En la Figura 41 se puede apreciar otro caso de mala estimación debido a que los brazos del usuario están fuera del rango de visión de la cámara. Y en este caso siguió

rotándose la imagen, pero esta vez para la izquierda. Notar que tanto en la Figura 40 como en la 41 hay una línea roja indicando que hay articulaciones que tocan el borde correspondiente, tanto sea abajo de la imagen del esqueleto como arriba del mismo.

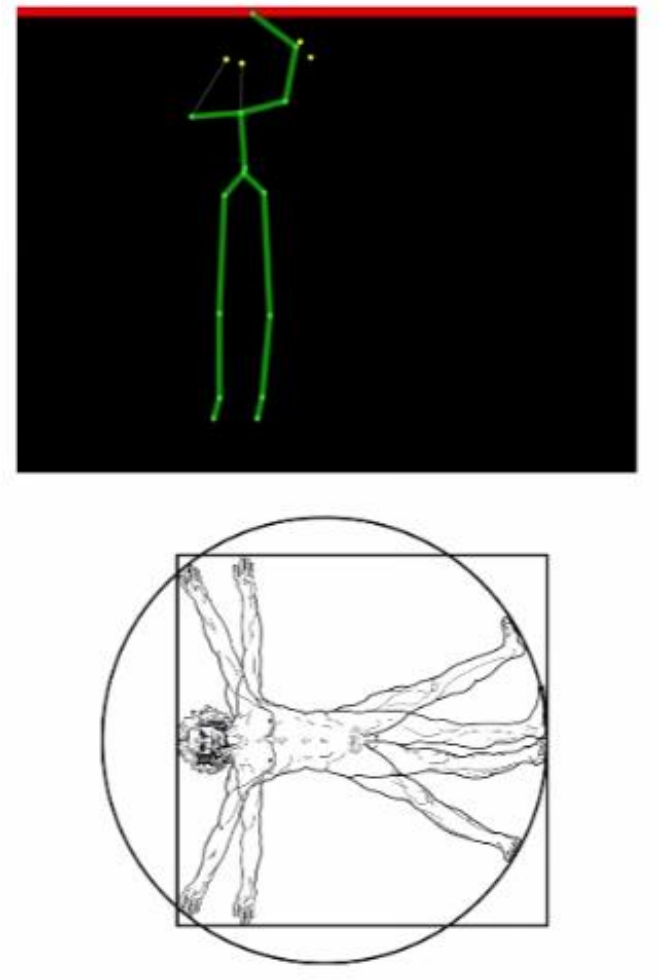

Figura 41: Malas estimaciones debido al usuario fuera del rango de visión

Durante las pruebas realizadas del prototipo se han encontrado estos inconvenientes y, en ambos casos, se ejecutaban las acciones de manipulación de la imagen debido a que en las implementaciones de las partes de los gestos sólo se tenía en cuenta la posición de cada articulación y se obviaba si la misma había sido bien detectada o era una estimación. Esto fue modificado en el prototipo y se recomienda a cualquier usuario del *Modelo de Gestos* que también tenga en cuenta esta cuestión para evitar comportamiento inesperado en su aplicación.

I) Obstrucción de un usuario por parte de otro

Durante las pruebas realizadas del prototipo se ha notado que si se encuentran dos usuarios interactuando con la aplicación y, en algún momento, un usuario se posiciona frente al otro, el Kinect dejará de rastrear el esqueleto del usuario que se encuentra

detrás debido a que no tendrá visión del mismo. A su vez, cuando deje de estar obstruido se volverá a rastrear su esqueleto pero con un identificador diferente. El desarrollador debe tener esto en cuenta para detectar si afecta o no al uso normal de su aplicación, y en caso positivo plantear una estrategia para solucionarlo.

J) Orientación del usuario

Otra cuestión interesante que se ha detectado en el uso del Kinect es que la cámara no discierne la orientación del usuario con la misma, es decir, si se encuentra de frente o de espaldas a la cámara. Esto provoca que si el usuario se encuentra de espaldas y mueve el brazo derecho, el esqueleto que recibirá la aplicación indicará que se está moviendo el brazo izquierdo. Esto puede provocar que se ejecuten gestos que no corresponden con los movimientos que está haciendo el usuario, por ejemplo en nuestro prototipo, saludar con la mano derecha de espaldas ejecutará las acciones asociadas a saludar con la mano izquierda.

Atacar este problema queda fuera del alcance de este trabajo pero algunas soluciones que se proponen estudiar son las de hacer uso de la detección facial que provee la librería del Kinect o hacer un estudio más detallista de los movimientos de los usuarios, por ejemplo, las personas tienden a hacer cualquier movimiento con sus brazos hacia adelante, por lo que mirando las posiciones en el eje Z (profundidad) se podría determinar si el usuario se encuentra o no de espaldas.

5. Casos de uso del prototipo

En este capítulo se presenta algunas secuencias de uso del prototipo descripto en el Capítulo 4.

A continuación daremos una secuencia de imágenes que muestran la realización de un gesto y su consecuente acción. Tomamos como ejemplo el gesto *RightHandWave* el cual, para ser completado, el usuario debe satisfacer cada una de sus partes. Estas partes son las mismas que se introdujeron en la Sección 3 pero, para que el gesto parezca un saludo y, además, no se ejecuten las acciones relacionadas por un movimiento del usuario sin la intención de hacer el gesto, es que se decidió que cada par de partes deba ser realizado tres veces seguidas. En las Figuras 42.a a 42.g se podrá observar al usuario realizando este gesto y cómo, en consecuencia, la imagen rota. Notar que a la derecha de las imágenes incluimos un diagrama de instancias simplificado en el que muestra el estado de la sub-clase de *Gesture* en cuestión y la relación con sus partes.

En la Figura 42.a se puede observar el esqueleto del usuario parado, y la instancia de *RightHandWave* espera por una instancia de *RightHandWavePart1* (esto indicado con la relación *siguiente*).

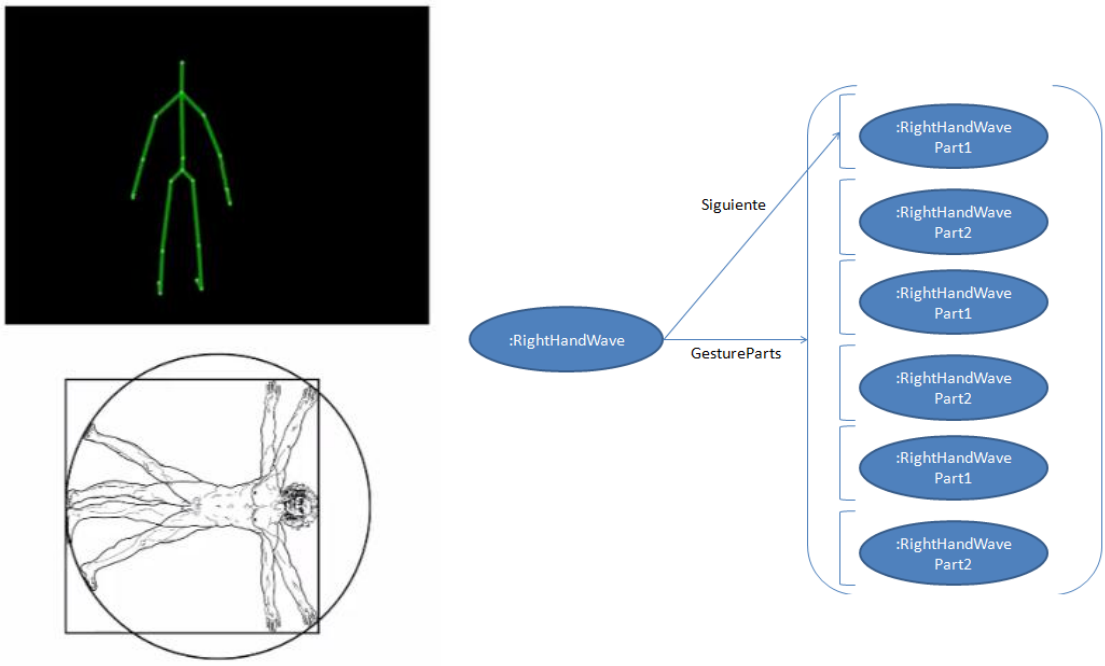

Figura 42.a: Estado original de instancias del gesto *RightHandWave*

En la Figura 42.b se puede ver que el usuario hizo un gesto que coincidió con *RightHandWavePart1,* y ahora la relación de *siguiente* está a la espera de una parte de gesto *RightHandWavePart2*.

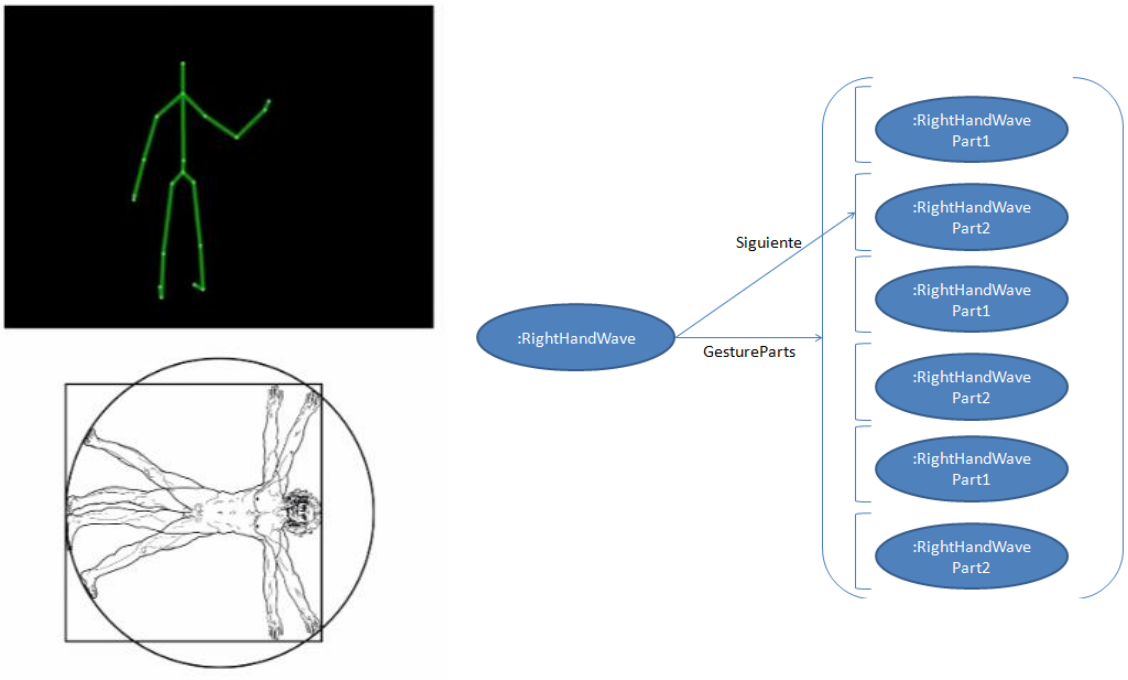

Figura 42.b: Primera parte de realización de gesto *RightHandWave*

En la Figura 42.c el usuario hizo un gesto que coincidió con *RightHandWavePart2,* y esto afectó la relación de *siguiente*, que ahora queda esperando por una parte de gesto *RightHandWavePart1*.

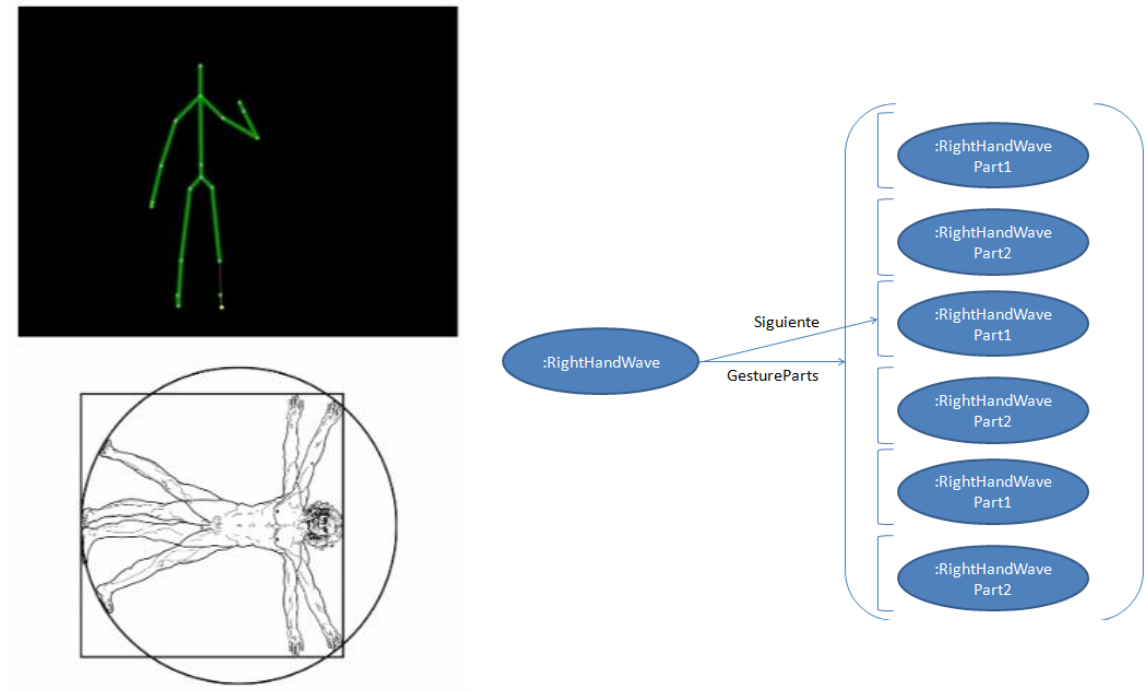

Figura 42.c: Segunda parte de realización de gesto *RightHandWave*

En la Figura 42.d el usuario hizo un gesto que coincidió con *RightHandWavePart1,* y esto afectó la relación de *siguiente*, que ahora queda esperando por una parte de gesto *RightHandWavePart2*. Esta parte de gesto fue realizada en la Figura 42.e quedando así, la relación de *siguiente* que ahora queda esperando por una parte de gesto *RightHandWavePart1*.

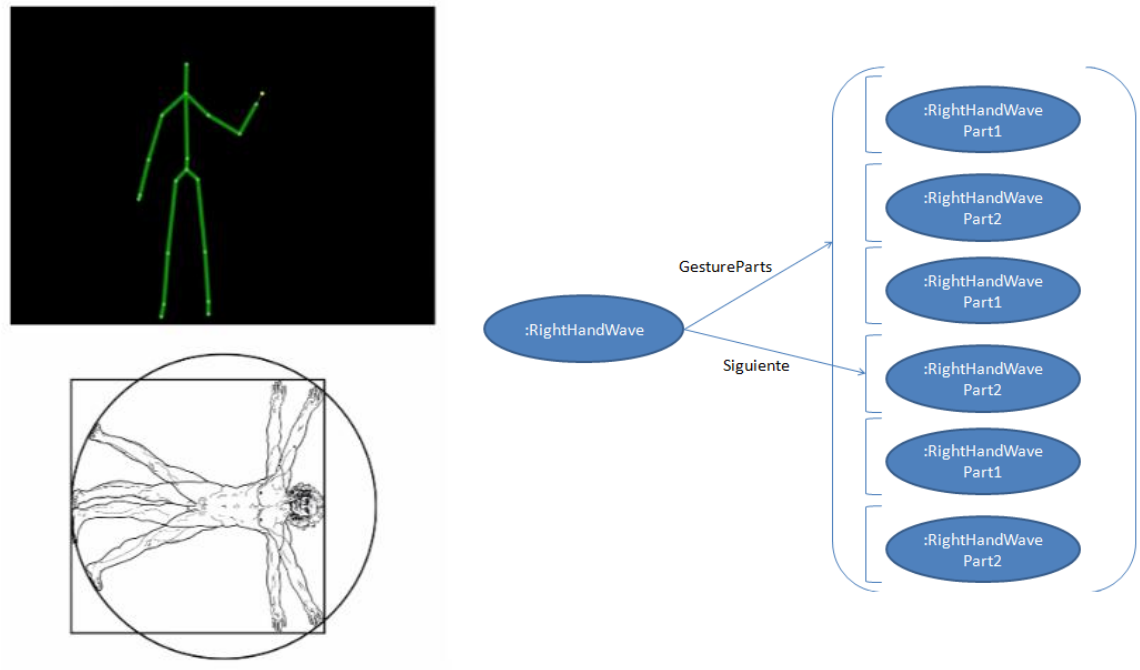

Figura 42.d: Tercera parte de realización de gesto *RightHandWave*

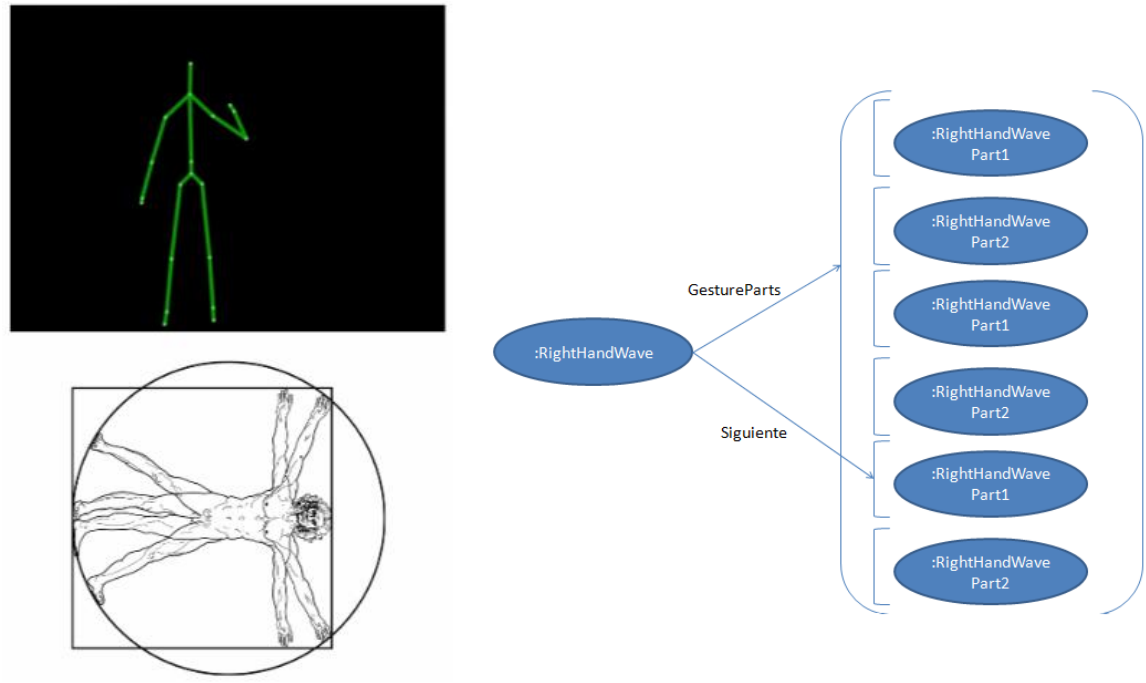

Figura 42.e: Cuarta parte de realización de gesto *RightHandWave*

En la Figura 42.f el usuario hizo un gesto que coincidió con *RightHandWavePart1,* y esto afectó el valor de la relación de *siguiente* (*RightHandWavePart2*). En la Figura 42.g se muestra que se realizó la parte de gesto *RightHandWavePart2,* y al quedar completo el gesto en sí, se ejecuto la acción asociada a este gesto, que, en este caso, es rotar a la derecha la imagen. La relación de *siguiente* queda indicando la parte de gesto *RightHandWavePart1*.

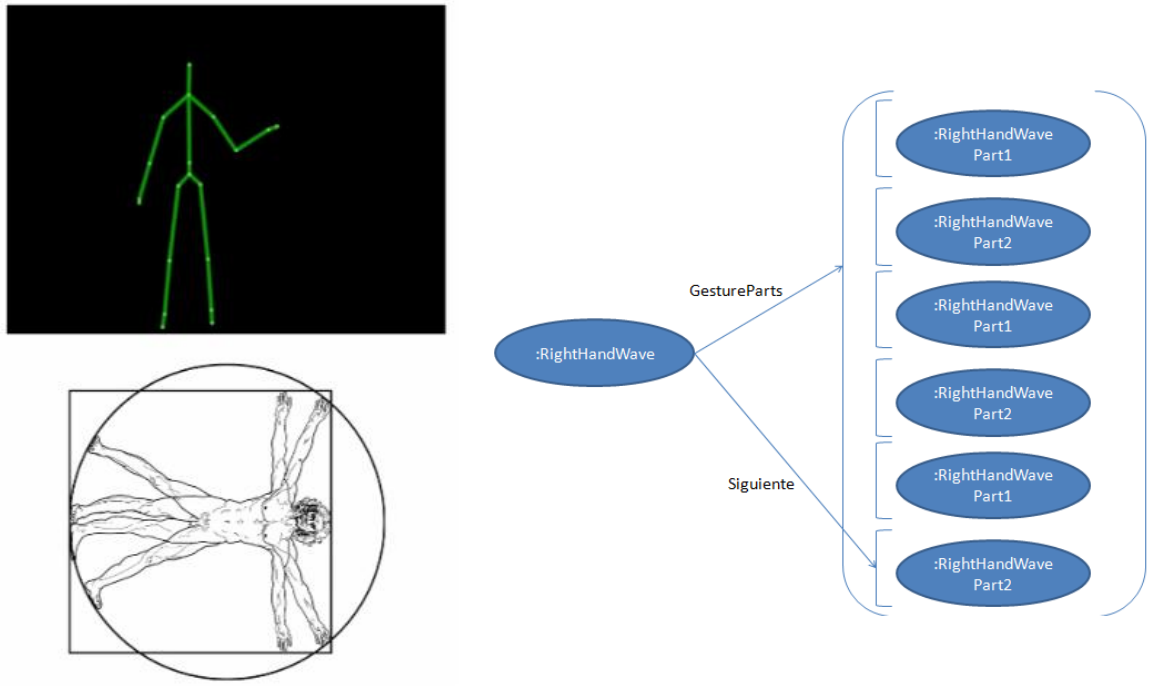

Figura 42.f: Quinta parte de realización de gesto *RightHandWave*

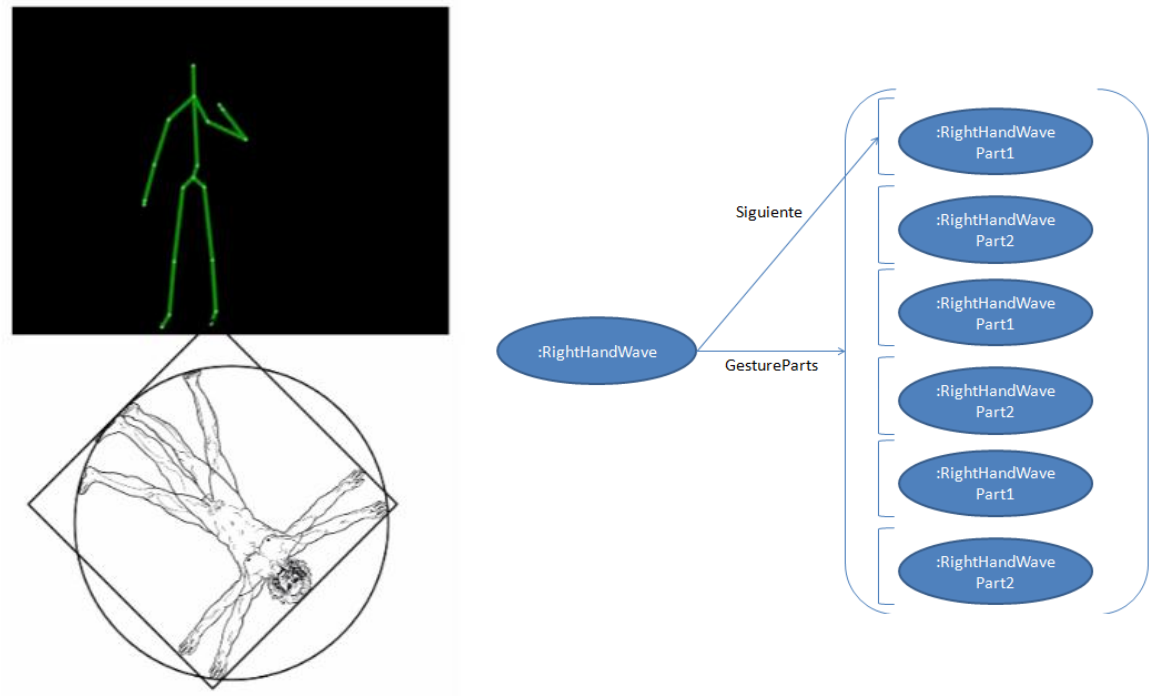

Figura 42.g: Sexta parte de realización de gesto *RightHandWave*

El lector podrá apreciar que en el momento en que se satisface la sexta y última parte del gesto, es cuando se dispara la secuencia descripta en las Figuras 32.a y 32.b, y en consecuencia, la imagen rota. A su vez, notar que luego de realizado satisfactoriamente el gesto, el mismo vuelve a su estado original para poder ser completado nuevamente.

A continuación se muestra en las Figuras 43.a a 43.g al usuario realizando el gesto de saludar con la mano izquierda. Nuevamente, a la derecha de cada imagen del esqueleto se encuentra un diagrama de instancias reducido que indica el estado del gesto y la relación con sus partes.

En la Figura 43.a se puede observar el esqueleto del usuario parado, y la instancia de *LeftHandWave* espera por una instancia de *LeftHandWavePart1* (esto indicado con la relación *siguiente*).

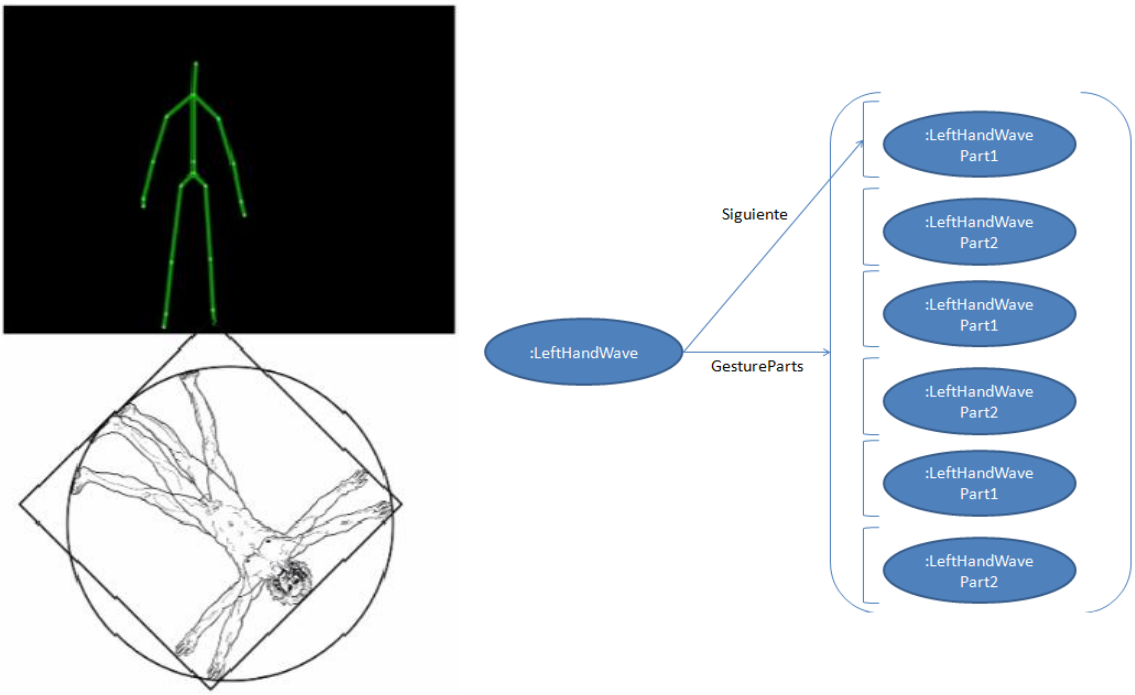

Figura 43.a: Estado original de instancias del gesto *LeftHandWave*

En la Figura 43.b se puede ver que el usuario hizo un gesto que coincidió con *LeftHandWavePart1,* y ahora la relación de *siguiente* está a la espera de una parte de gesto *LeftHandWavePart2*.

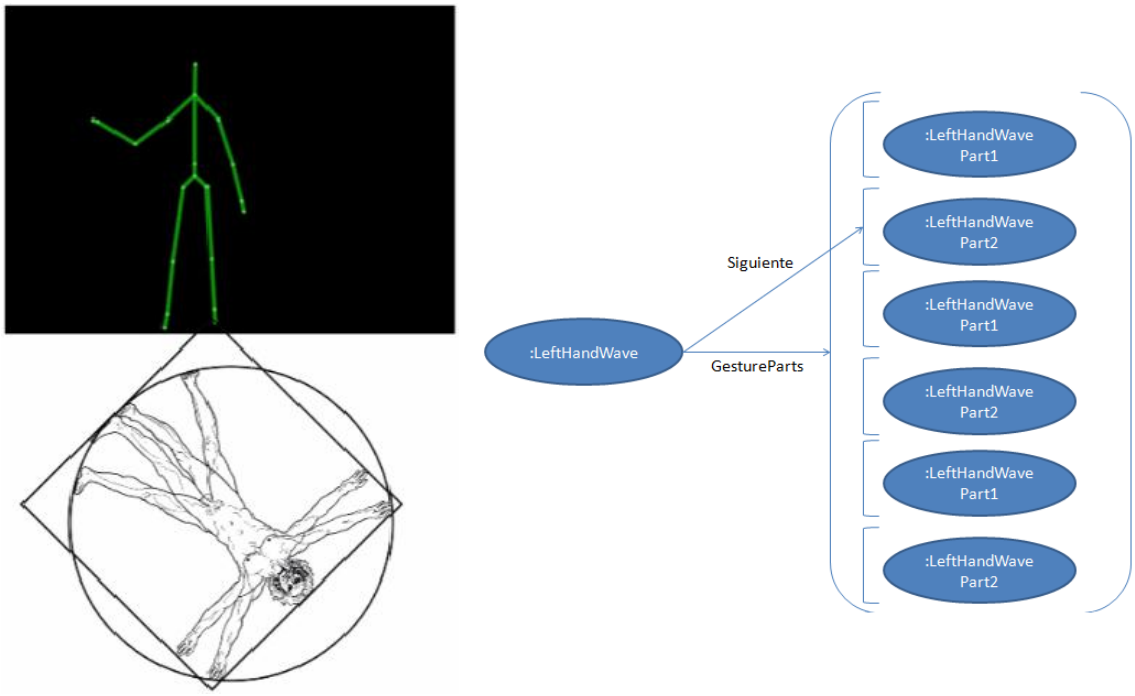

Figura 43.b: Primera parte de realización del gesto *LeftHandWave*

En la Figura 43.c el usuario hizo un gesto que coincidió con *LeftHandWavePart2,* y esto afectó la relación de *siguiente*, que ahora queda esperando por una parte de gesto *LeftHandWavePart1*.

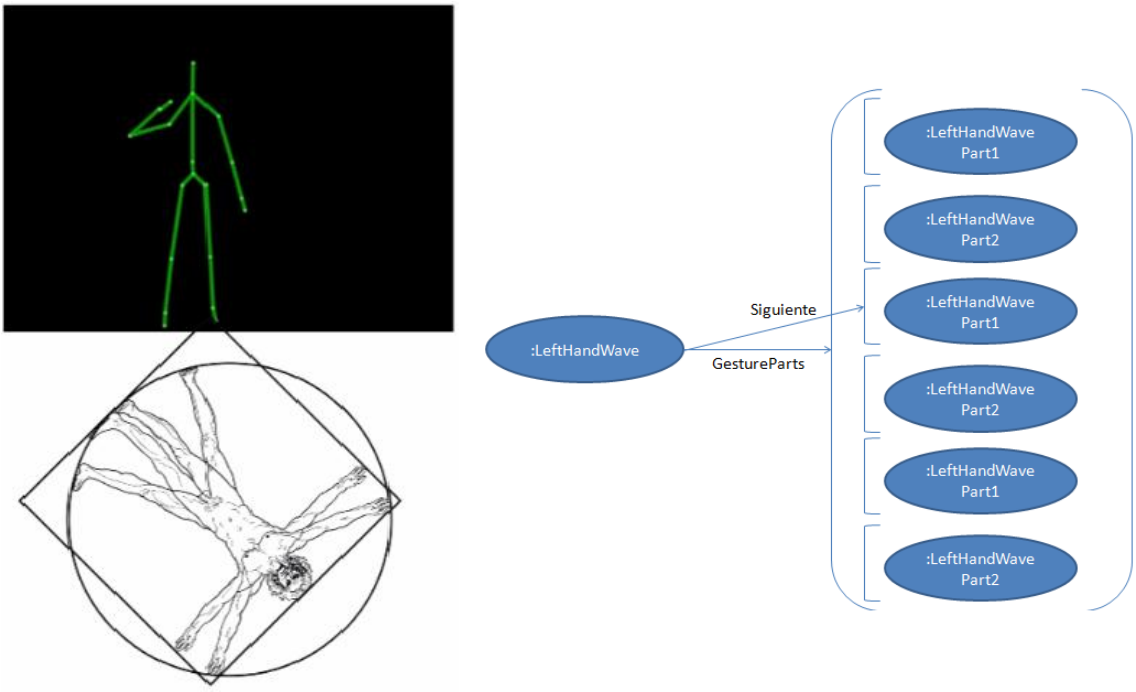

Figura 43.c: Segunda parte de realización del gesto *LeftHandWave*

En la Figura 43.d el usuario hizo un gesto que coincidió con *LeftHandWavePart1,* y esto afectó la relación de *siguiente*, que ahora queda esperando por una parte de gesto *LeftHandWavePart2*. Esta parte de gesto fue realizada en la Figura 43.e quedando así, la relación de *siguiente* esperando por una parte de gesto *LeftHandWavePart1*.

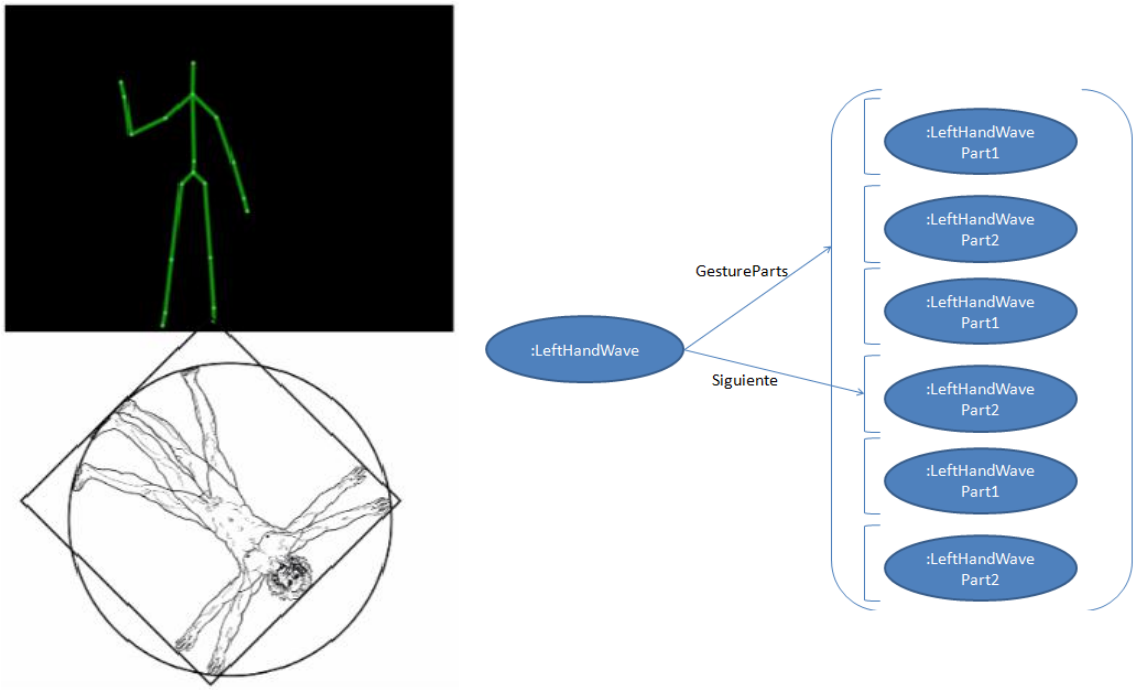

Figura 43.d: Tercera parte de realización del gesto *LeftHandWave*

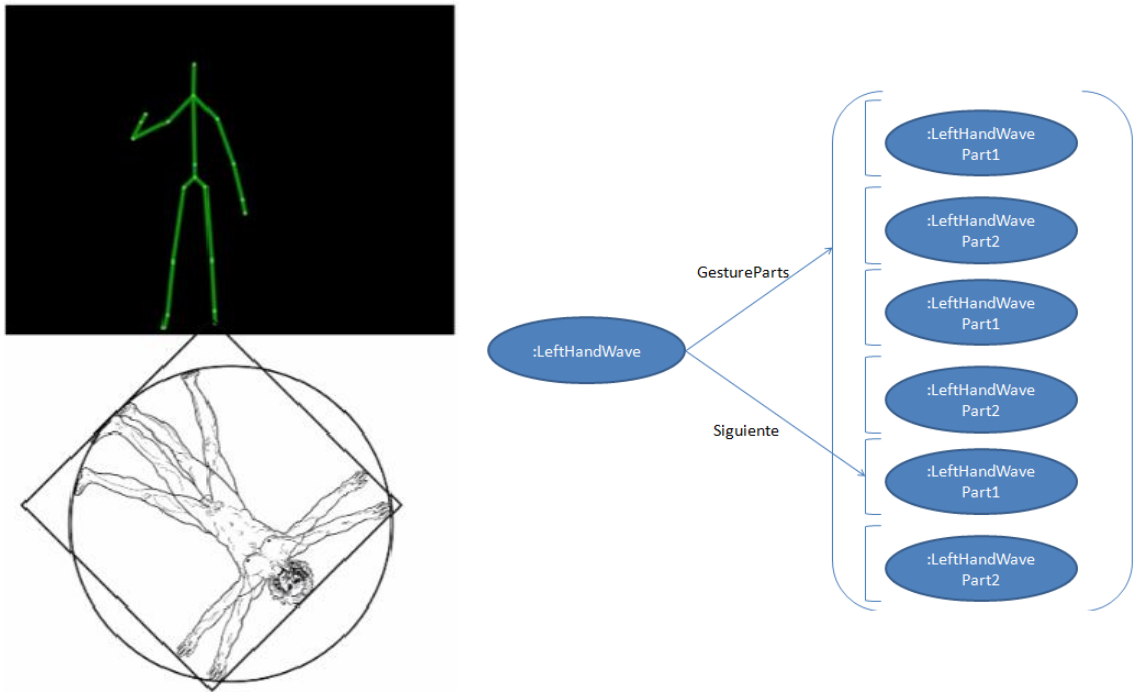

Figura 43.e: Cuarta parte de realización del gesto *LeftHandWave*

En la Figura 43.f el usuario hizo un gesto que coincidió con *LeftHandWavePart1,* y esto afectó el valor de la relación de *siguiente* (*LeftHandWavePart2*). En la Figura 43.g se muestra que se realizó la parte de gesto *LeftHandWavePart2,* y al quedar completo el gesto en sí, se ejecutó la acción asociada a este gesto, que, en este caso, es rotar a la izquierda la imagen. La relación de *siguiente* queda indicando la parte de gesto *LeftHandWavePart1.*

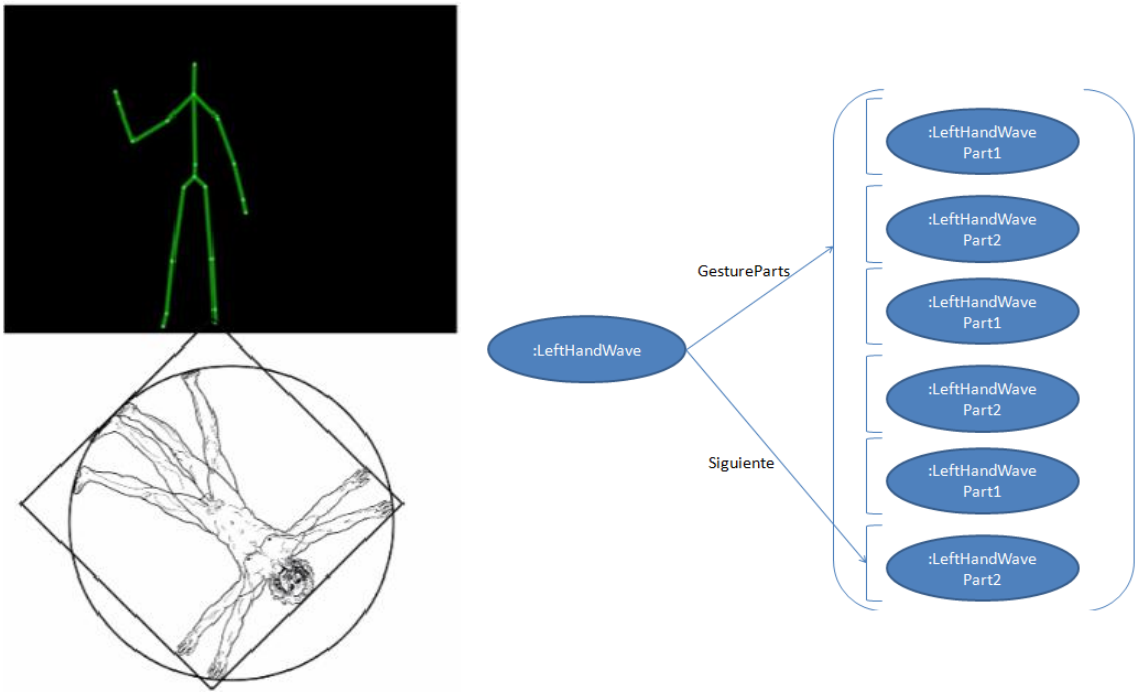

Figura 43.f: Quinta parte de realización del gesto *LeftHandWave*

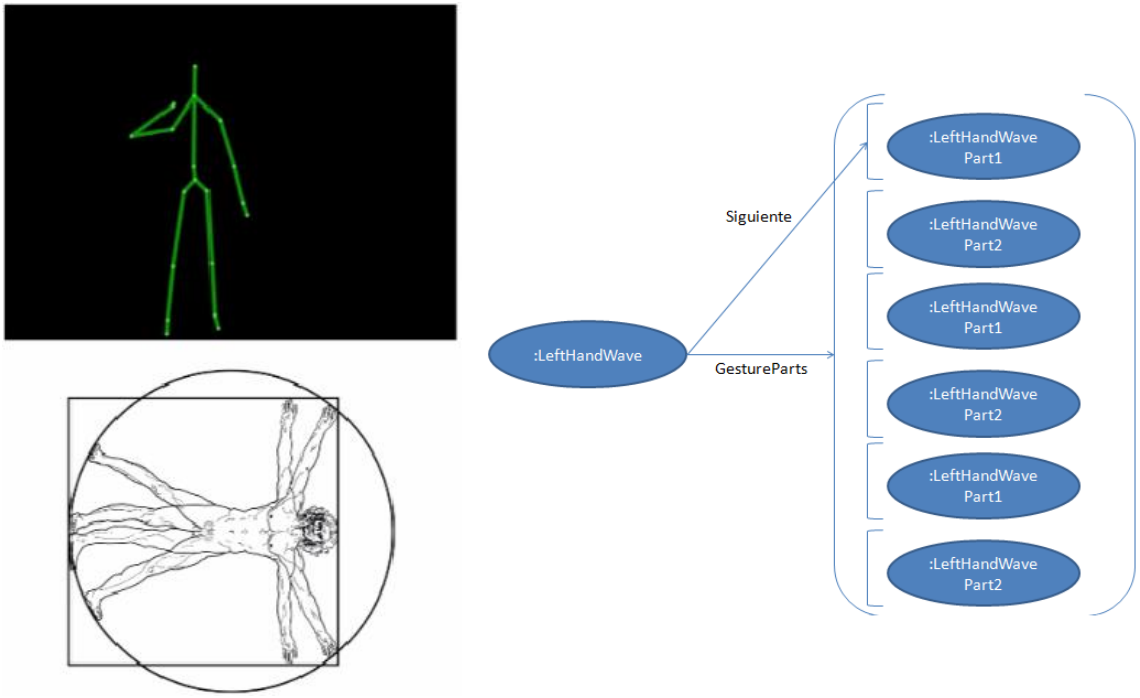

Figura 43.g: Sexta parte de realización del gesto *LeftHandWave*

Como último caso de uso mostraremos a dos usuarios realizando gestos simultáneamente. Los mismos se pueden apreciar en las Figuras 44.a a 44.d. En este caso omitimos los diagramas de instancias reducidos como mostramos anteriormente pero el lector puede imaginar el estado de las instancias para ambos usuarios dado que son muy similares a los anteriores (Figuras 42.a a 42.g y Figuras 43.a a 43.g.).

En la Figura 44.a se pueden observar los esqueletos de dos usuarios parados. Cada una de sus instancias de gestos se encuentra esperando que se satisfaga su primer parte. Identificaremos como *usuario-1* al esqueleto de la izquierda y *usuario-2* al esqueleto de la derecha. La imagen relacionada a las acciones de los gestos es compartida por ambos usuarios.

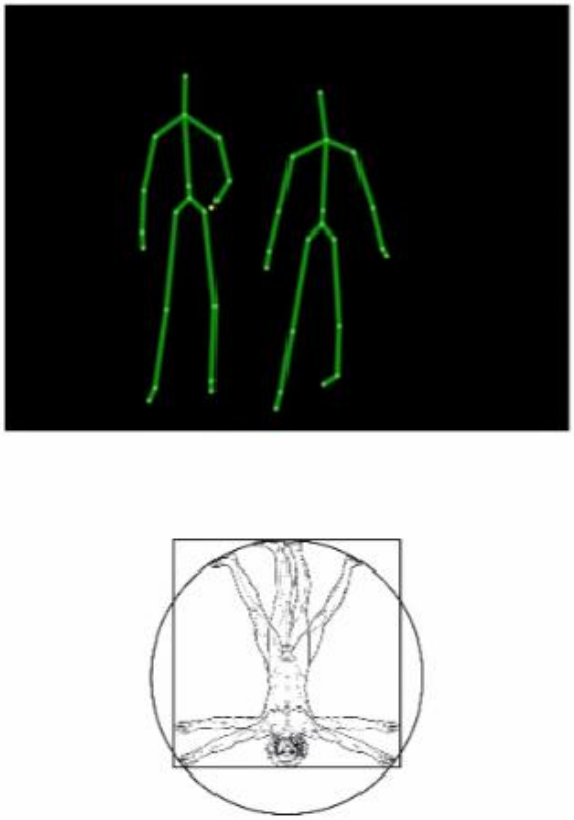

Figura 44.a: Primera parte de interacción de dos usuarios

En la Figura 44.b se puede ver dos secuencias de gestos de ambos usuarios, en particular, el *usuario-1* realizó las primeras dos partes del gesto *RightHandWave (RightHandWavePart1* y *RightHandWavePart2)*, por lo tanto su instancia de *RightHandWave* estará esperando por su tercer parte (*RightHandWavePart1)*. Por otro lado, el *usuario-2* realizó las primeras dos partes del gesto *LeftHandWave (LeftHandWavePart1* y *LeftHandWavePart2),* por lo tanto su instancia de *LeftHandWave* estará esperando por su tercer parte (*LeftHandWavePart1)*.

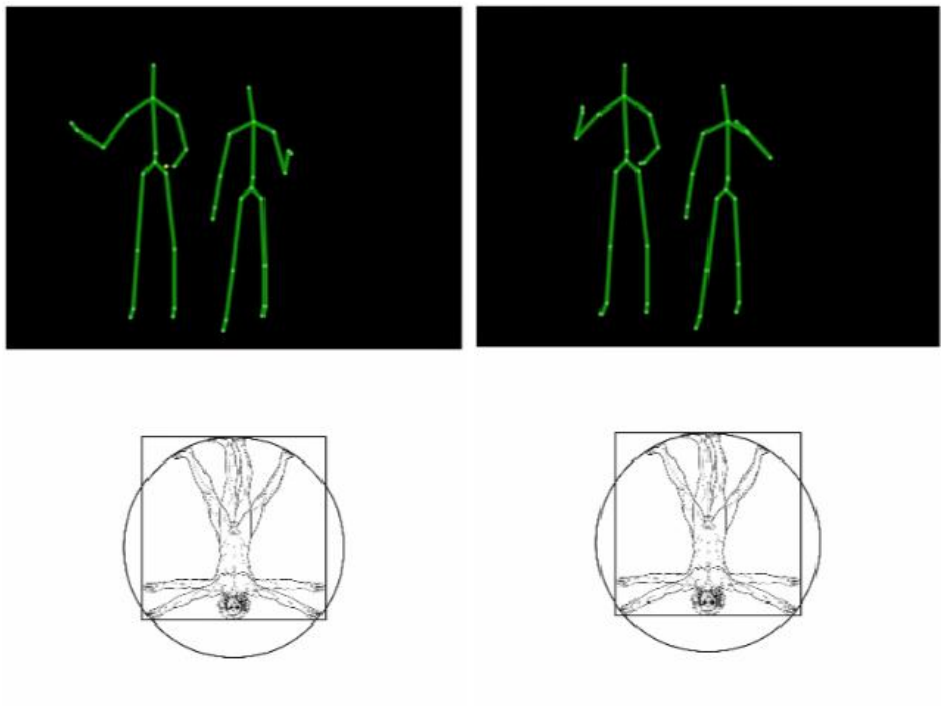

Figura 44.b: Segunda y tercer parte de interacción de dos usuarios

En la Figura 44.c se puede ver que cada uno de los usuarios realizó nuevamente el par de partes del gesto que realizaron en la Figura 44.b. Por lo tanto, cada uno de sus gestos estará esperando que se realice su quinta parte, siendo *RightHandWavePart1* para el *usuario-1* y *LeftHandWavePart1* para el *usuario-2*.

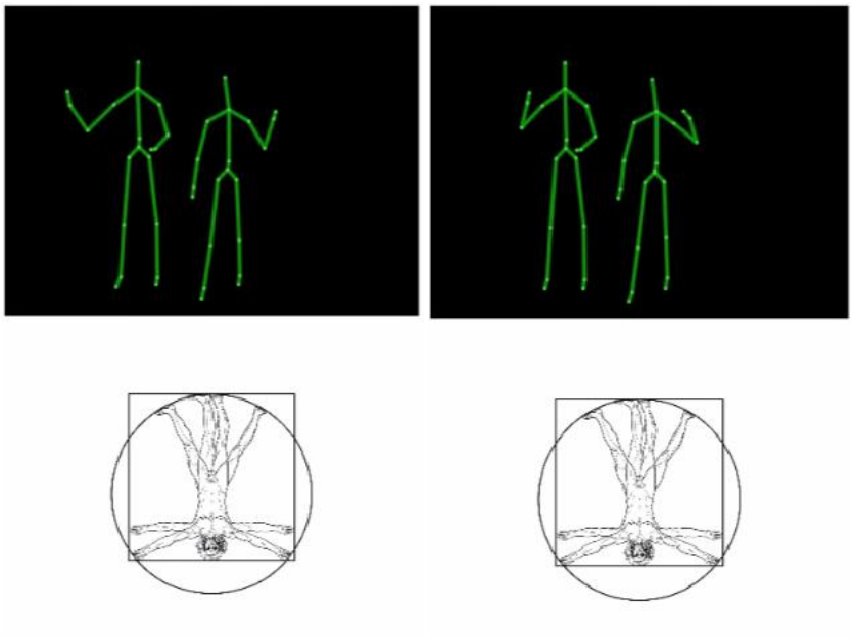

Figura 44.c: Cuarta y quinta parte de interacción de dos usuarios

En la Figura 44.d se puede ver que nuevamente los usuarios realizaron las partes del gesto que realizaron anteriormente (Figura 44.c), provocando así que se completen cada uno de sus gestos. Podemos notar que en consecuencia la imagen rotó hacia la derecha y aumentó de tamaño. Esto se debe a que el *usuario-1* tenía asociado la acción *RotateImageRightAction* a su gesto *RightHandWave*, mientras que el *usuario-2* tenía asociado la acción *ZoomInImageAction* a su gesto *LeftHandWave*. Es decir, los gestos de ambos usuarios en este caso modificaron el display de la imágen.

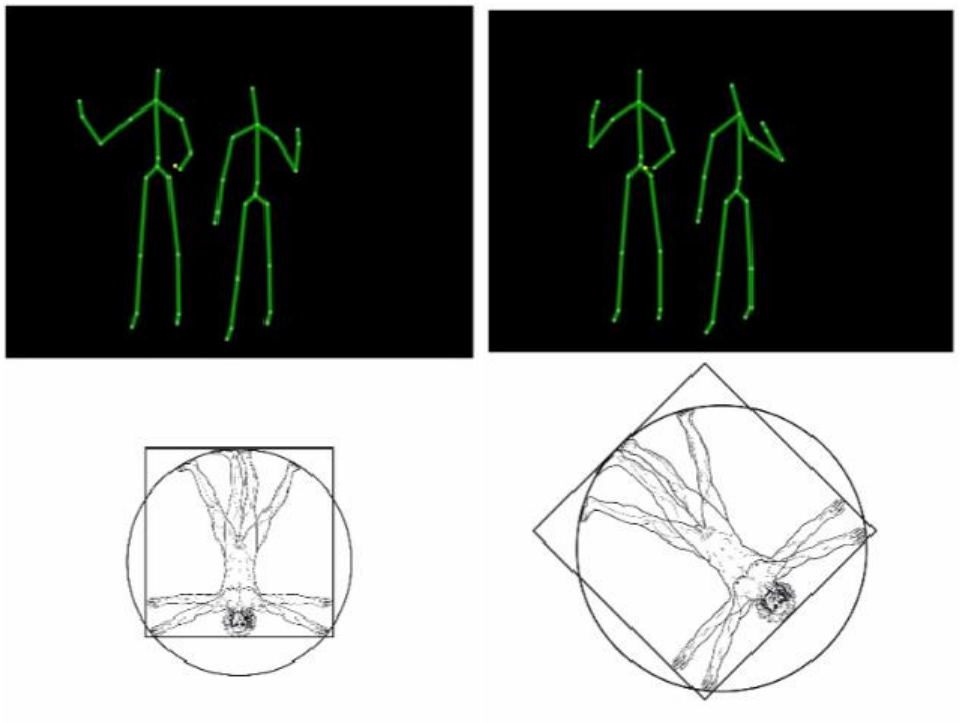

Figura 44.d: Sexta y séptima parte de interacción de dos usuarios

Una vez ejecutada las acciones, los gestos de ambos usuarios vuelven a su estado original para poder ser realizados nuevamente. Cabe destacar que lo anterior es sólo un ejemplo y no es necesario que los usuarios realicen gestos distintos. Cada uno puede realizar un gesto totalmente independiente del otro usuario e incluso realizarlos a destiempo.

Los ejemplos mostrados en este capítulo permiten apreciar como usando Kinect se pueden detectar los gestos de los usuarios, y usando el modelo propuesto en esta tesis se pueden ejecutar diferentes acciones automáticamente.

6. Conclusiones y Trabajos Futuros

En el transcurso de esta tesis se presentaron diferentes modelos, primero se presentó un modelo de gestos general y de fácil extensibilidad en la Sección 3.2. Este modelo ya cumple con uno de los objetivos buscados que era la necesidad de contar con un módulo de gestos que, sin importar el sensor que se esté utilizando, se puedan definir diferentes gestos y asociarlos a distintas acciones dinámicamente. El problema que contaba este modelo es que el concepto de esqueleto queda integrado en todas las clases del modelo, generando así un acoplamiento del mismo. Esto generaba que el mecanismo de observación de los cambios del esqueleto fuera comportamiento de las clases, haciendo que las mismas estuvieran sobrecargadas de comportamiento, teniendo responsabilidades que no le eran propias. Para poder desacoplar el concepto de esqueleto de las clases relacionadas a los gestos, se presentó un nuevo modelo de contexto en la Sección 3.3 capaz de proveer comunicación entre diferentes módulos de una aplicación cualquiera sin que los mismos cuenten con un conocimiento explícito de los demás. Esta capa permite observar los cambios en un modelo de dominio y accionar acorde a estos, sin que dicho dominio conozca este comportamiento explícitamente. Es decir, queda representada una capa de contexto independiente del dominio con la cual se la integre. Finalmente, en la Sección 3.4 se combinaron los dos modelos anteriores para tener el modelo de gestos general independiente de los mecanismos de observación de cambios de los esqueletos. Esto permite que el modelo de gestos defina los gestos relevantes junto a las acciones asociadas para cada usuario, y el mecanismo de adaptación acorde a los cambios en el movimiento del usuario sean manejados por la modelo de contexto.

El modelo integrado presentado en la Sección 3.4, presenta grandes ventajas a los desarrolladores que vayan a hacer uso del mismo. Entre ellas se destacan las posibilidades de definir sus propios gestos y sus propias acciones, lo que permite que este módulo sea fácilmente adaptable a cualquier dominio. A su vez, los desarrolladores pueden implementar sub-clases de *EventHandler* para poder reaccionar a ciertos eventos que ocurran dentro de su aplicación y así generar nuevas funcionalidades de manera sencilla, incluso pueden agregar funcionalidades que requieran comunicación con módulos anteriores sin que afecte el comportamiento o el código ya escrito de los mismos. Además de los *EventHandlers,* la capa de contexto cuenta con múltiples opciones donde los desarrolladores puedan definir sus propias sub-clases para definir un comportamiento más específico, ya sea creando una nueva clase de *MatchingPolicy* para crear el filtro que se desee o sub-clasificando el *AdaptationEnvironment* para tener otra manera de notificar los eventos, entre otras. Aunque el modelo permite todas estas modificaciones para crear comportamiento más específico, creemos que el modelo propuesto en esta tesis es suficientemente genérico y simple de usar como para abarcar gran parte de los requerimientos que puedan tener las aplicaciones que utilizan gestos.

Sin embargo, todo esto no se encuentra libre de desventajas. Para la utilización de este modelo integrado (*Modelo de Gestos combinado con la Capa de Contexto*) es necesario entender

su funcionamiento, cuestión que consume tiempo y distrae al desarrollador del objetivo principal de su aplicación. A su vez, como es de esperarse en un framework, el desacoplamiento y tratar cuestiones generales de aplicaciones conlleva una sobrecarga de ineficiencia debido a la cantidad extra de clases, instancias y mensajes que se deben realizar para lograr el funcionamiento esperado. A pesar de estas desventajas creemos que este módulo es útil debido a que ahorra mucho trabajo y tiempo para el desarrollo, y las cuestiones de eficiencia son despreciables a menos que se esté desarrollando un sistema que necesita una gran velocidad de respuesta.

Como comentábamos anteriormente, las clases del modelo integrado pueden ser sub-clasificadas fácilmente para tener funcionalidades más específicas. Así, podemos pensar que se podrían desarrollar sub-módulos que engloben el comportamiento específico para cierto tipo de aplicaciones. Por ejemplo, sub-módulos que contengan gestos muy usados en variedad de aplicaciones, otros que contengan las implementaciones específicas de ciertos sensores, o incluso definir las acciones que suelen aparecer en aplicaciones que compartan ciertas funcionalidades.

Con el avance y desarrollo de cada vez más sensores de movimiento, se abre una variada gama de aplicaciones relacionadas. A fin de explicar más ventajas relacionadas con el modelo propuesto, combinando el *Modelo de Gestos* con la *Capa de Contexto*, creemos relevante presentar más detalles de los usos actuales de estos sensores. En [Bratitsis and Kandroudi, 2014] se discuten los diferentes usos de los tres sensores principales de movimiento en educación. En este paper primero hacen un estudio de aplicaciones desarrolladas haciendo uso de estos sensores de movimiento para asistir en la educación de personas con ciertas discapacidades. En el mismo se encuentra una tabla que lista los distintos usos que se le dieron a cada sensor. Esta misma tabla se puede apreciar en la Figura 45.

Authors	Console	Study/ Results
Lang et al., 2012	XBOX Kinect	Kinect for supporting sign language learning.
Aimaiti & Yan,2011	XBOX Kinect	Powerful tool for computers in order to understand human body language and thus providing opportunities in sign language or medical research.
Butler & Willet,2010	Wii	Benefits of Wii Balance Board in patients' rehabilitation regarding gross motor co-ordination, balance and strength.
DePriest &Barilovits, 2011	XBOX Kinect	Valuable applications for physical therapy and home rehabilitation exercises.
Hammond et al.,2013	Wii	Effectiveness in motor skills of children with Developmental Co-ordination disorder.
Kee, 2009	PlayStation Move	Case study research with autistic children regarding racing car video games for learning physics.
Pyfers,2012	XBOX Kinect	Kinect technology could be a useful asset for educating deaf people.
Rahman, 2010	Wii	Balance improvement for children with Down syndrome.
Shih et al., 2010	Wii	Nintendo Wii Balance Board on two children with multiple disabilities helped them to adjust their abnormal standing posture.
Urturi et al.,2012	XBOX Kinect	Exploitation of XBOX Kinect for disabled people, in wheel chairs indicated an increase in their motivation although the game concerned physical exercises.
Wuang et al.,2011	Wii	Wii virtual reality improved motor proficiency, visual-integrative abilities, and sensory integrative functions in children with Down Syndrome.
Zafrulla et al.,2011	XBOX Kinect	Kinect can be a viable option for sign verification.

Figura 45: Proyectos educativos con uso de sensores de movimiento [Bratitsis and Kandroudi, 2014]

Se puede apreciar en la figura anterior que ha habido una gran cantidad de autores investigando sobre los posibles usos de distintos sensores de movimiento. Cada uno de ellos ha logrado un avance o aporte a la mejor educación de personas discapacitadas. El modelo integrado propuesto en esta tesis podría ser usado por cualquiera de estos proyectos para desarrollar sus implementaciones y facilitar el trabajo de los autores dado que el mismo puede ser extendido para soportar cualquiera de los sensores utilizados.

Además, en [Bratitsis and Kandroudi, 2014] se presenta el uso de los sensores en educación en sentido general, no sólo para personas con discapacidades. En el mismo se

caracterizan las aplicaciones o juegos según en cómo ayudan a los cuatro tipos de educación: física, cognitiva, social y emocional. Esta caracterización se hace en particular considerando juegos que hacen uso del Kinect. Dicha caracterización se puede apreciar en la Figura 46.

GAME	Physical develop-ment	Cognitive develop-ment	Emotional develop-ment	Social develop-ment
Body and brain Connection	X	X		
Kinect sports	X			X
Kinectimals			X	X
The Fantastic Pets			X	X
Kinect Adventures	X		X	X
Disneyland Adventures	X		X	X
Sesame street: Once upon a monster			X	X
Kinect FunLabs		X		X

Figura 46: Estudio sobre juegos y su aporte a cada tipo de educación [Bratitsis and Kandroudi, 2014]

Los juegos listados en la Figura 46 podrían fácilmente ser desarrollados utilizando el modelo integrado propuesto. Las acciones correspondientes a cada uno de ellos podría ser implementada y asociada a uno de los gestos existentes de manera sencilla como se ha mostrado durante el desarrollo de esta tesis. A su vez, dado que todos hacen uso del Kinect, estos juegos podrían usar el mismo código base para capturar los movimientos y hasta usar el mismo conjunto de gestos.

A continuación se listarán distintos trabajos futuros que podrían realizarse usando como base el modelo integrado propuesto. Estos trabajos pueden ser tanto a nivel de modelado como de implementación:

- Extender el modelo con un conjunto de gestos que sean usados en la mayoría de las aplicaciones. Cada uno de estos gestos podría estar implementado para varios sensores de movimiento.
- Extender el modelo con un conjunto de acciones usuales para ciertos dominios de aplicaciones.
- Extender el modelo de la capa de contexto para proveer comportamiento más específico. Por ejemplo, a través de la creación de nuevos *MatchingPolicy* o *AdaptationEnvironments.*

- Desarrollar mecanismos de sensado para determinar los usuarios que interactúan con el sistema. Por ejemplo, el uso de QRs u otro tipo de sensores de identificación. Estos mecanismos serán complementos de los sensores de movimiento.

- Analizar el funcionamiento de distintos sensores de movimiento para detectar similitudes y diferencias.

- Implementar la captura de esqueletos desde diferentes sensores de movimiento. Esto permitiría la simultaneidad de uso de diferentes mecanismos de sensado.

- Analizar y diseñar gestos que requieran la interacción con múltiples sensores de movimiento. Es decir, que ciertos conjuntos de partes de gestos sean tomados de sensores diferentes.

- Diseñar estrategias que utilicen múltiples sensores para reducir los errores de detección de los esqueletos. Por ejemplo, utilizar dos Kinects enfrentados para evitar la obstrucción de los usuarios por parte de otros u otros objetos.

- Realizar prototipos para dominios particulares para detectar funcionalidades específicas aún no soportadas por el modelo, y así enriquecer el mismo.

- Diseñar una herramienta que permita a usuarios no expertos crear gestos personalizables a través del registro de los esqueletos que se toman del sensor de movimiento y a su vez asociarles acciones específicas a cada uno de ellos.

Bibliografía

[Binbin Xu, 2014] XU, B. (2014). Kinect enabled motion based touchless drawing tool.

[Bond and Curran, 2014] Bond, A., & Curran, K. (2014). A Camera-Based System for Determining Hand Range of Movement Measurements in Rheumatoid Arthritis. *Recent Advances in Ambient Intelligence and Context-Aware Computing*, 39.

[Bratitsis and Kandroudi, 2014] Bratitsis, T., & Kandroudi, M. (2014). Motion sensor technologies in education.

[Brown et al, 1997] Brown, P. J., Bovey, J. D., & Chen, X. (1997). Context-aware applications: from the laboratory to the marketplace. *Personal Communications, IEEE*, *4*(5), 58-64.

[Brown, 1995] Brown, P. J. (1995). The stick-e document: a framework for creating context-aware applications. *ELECTRONIC PUBLISHING-CHICHESTER-*, *8*, 259-272.

[C. Emmanouilidi et al, 2013] Emmanouilidis, C., Koutsiamanis, R. A., & Tasidou, A. (2013). Mobile guides: Taxonomy of architectures, context awareness, technologies and applications.*Journal of Network and Computer Applications*, *36*(1), 103-125.

[Fortier et al, 2007] Fortier, A., Challiol, C., Rossi, G., & Gordillo, S. (2007). Physical Hypermedia: a Context-Aware approach. In *Proceedings of the CAiSE* (Vol. 7, pp. 499-513).

[Dey et al, 2001] Dey, A. K., Abowd, G. D., & Salber, D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications.*Human-computer interaction*, *16*(2), 97-166.

[Dey, 1998] Dey, A. K. (1998, March). Context-aware computing: The CyberDesk project. In*Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments*(pp. 51-54).

[Dey, 2000] Dey, A. K. (2000). *Providing architectural support for building context-aware applications* (Doctoral dissertation, Georgia Institute of Technology).

[Franklin and Flachsbart, 1998] Franklin, D., & Flaschbart, J. (1998, March). All gadget and no representation makes jack a dull environment. In *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments* (pp. 155-160).

[Gallo et al, 2011] Gallo, L., Placitelli, A. P., & Ciampi, M. (2011, June). Controller-free exploration of medical image data: Experiencing the Kinect. In *Computer-based medical systems (CBMS), 2011 24th international symposium on* (pp. 1-6). IEEE.

[Gamma et al, 1994] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

[Pascoe, 1998] Pascoe, J. (1998, October). Adding generic contextual capabilities to wearable computers. In *Wearable Computers, 1998. Digest of Papers. Second International Symposium on* (pp. 92-99). IEEE.

[Schilit and Theimer, 1994] Schilit, B. N., & Theimer, M. M. (1994). Disseminating active map information to mobile hosts. *Network, IEEE*, *8*(5), 22-32.

[Schilit et al, 1994] Schilit, B., Adams, N., & Want, R. (1994, December). Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on* (pp. 85-90). IEEE.

[Schmidt, 2013] Schmidt, A. (2013). Context-aware computing: context-awareness, context-aware user interfaces, and implicit interaction. *The Encyclopedia of Human-Computer Interaction, 2nd Ed.*

[Shehzad et al, 2004] Shehzad, A., Ngo, H. Q., Pham, K. A., & Lee, S. Y. (2004, September). Formal modeling in context aware systems. In *Proceedings of the First International Workshop on Modeling and Retrieval of Context*.

[Stowers et al, 2011] Stowers, J., Hayes, M., & Bainbridge-Smith, A. (2011, April). Altitude control of a quadrotor helicopter using depth map from Microsoft Kinect sensor. In*Mechatronics (ICM), 2011 IEEE International Conference on* (pp. 358-362). IEEE.

[Ward et al, 1997] Ward, A., Jones, A., & Hopper, A. (1997). A new location technique for the active office. *Personal Communications, IEEE*, *4*(5), 42-47.

[Yau et al, 2002] Yau, S. S., Karim, F., Wang, Y., Wang, B., & Gupta, S. K. (2002). Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, *1*(3), 33-40.

Anexo A: Uso de la capa de contexto

En este anexo entraremos en detalle en los pasos que debiera seguir el desarrollador para implementar una nueva funcionalidad en su aplicación utilizando la capa de contexto. Por funcionalidad nos referimos a todo el mecanismo de eventos descripto anteriormente en la Sección 3.3. Suponiendo que el desarrollador ya cuenta con un objeto que realiza cierta acción, describiremos los pasos que tiene que tomar para que se notifique un evento de dicha acción y, otro objeto u otro módulo, pueda enterarse sin la necesidad de una referencia explícita al objeto anterior.

La funcionalidad presentada en este anexo no está relacionada a los gestos, permitiendo de esta manera observar el comportamiento de la Capa de Contexto para otro dominio diferente. Tener en cuenta que todo el código que se muestre en este anexo está escrito en C#, lenguaje con el que se desarrolló el prototipo.

Supongamos que tenemos una clase *User* y que cada instancia representa a un usuario diferente. Supongamos también que cada instancia contiene una variable que representa si el usuario está actualmente dentro del sistema y que ya se encuentra implementado todo lo referente a la actualización de ese estado. El código de los métodos de la clase *User* que involucran lo anteriormente descripto se especifican en el Código 1.

```
public void LogIn()
{
    loggedIn = true;
}

public void LogOut()
{
    loggedIn = false;
}
```

Código 1: Definición de los métodos LogIn y LogOut

Ahora supongamos que el desarrollador quiere que se notifique a todo objeto interesado del ingreso o salida de un usuario al sistema. Para ello, lo primero que debe hacer es sub-clasificar la clase *ContextEvent* para que exista un evento que represente dicho ingreso. Llamaremos a esta sub-clase *UserStateChangeEvent* donde el objeto disparador del evento será la instancia de *User* y el objeto valor que representa el cambio será el booleano *True* o *False*. Su constructor queda bastante simple, dado que su super-clase *ContextEvent* ya almacena estos dos objetos en su constructor:

```
public UserStateChangeEvent(User user, Boolean boolValue): base(user, boolValue){
}
```

Código 2: Definición del método UserStateChangeEvent

Es necesario, a su vez, sub-clasificar la clase *ContextEventFactory* para que exista un objeto capaz de crear el evento que acabamos de codificar. Llamaremos a la clase *UserStateChangeEventFactory* y su código se puede apreciar en el Código 3. Notar que en este paso se podría utilizar la clase *GeneralEventFactory* introducida en la Sección 4.3 pero se decidió implementar una sub-clase para dar un ejemplo y mostrar que no es demasiado compleja su implementación.

```
public class UserStateChangeEventFactory extends ContextEventFactory
{
    public override ContextEvent GetEvent(Object triggerer, Object objectChanged)
    {
        If(triggerer is User && objectChanged is Boolean)
        {
            return new UserStateChangeEvent((User) triggerer, (Boolean) objectChanged);
        }
        else
        {
            throw new InvalidCastException();
        }
    }
}
```

Código 3: Definición de la clase UserStateChangeEventFactory

El método `GetEvent`() es el método abstracto de la clase *ContextEventFactory* y es el mismo que va a llamar el *ContextFeature* cuando le pida al factory un nuevo evento. Podemos ver que el método es bastante simple (ver Código 3), lo único que tenemos que tener en cuenta es verificar los tipos de los objetos que recibimos dado que nuestro evento sólo puede recibir un objeto *User* y un *Boolean*. Además, como la clase abstracta *ContextEventFactory* fue implementada para que sea lo más general posible, el método debe recibir dos *Object* (como se puede apreciar en el Código 3). Cualquier otro tipo más específico estaría restringiendo nuestra posibilidad de crear eventos.

El siguiente paso sería crear la acción que el desarrollador quiere que se ejecute cuando se dispara el evento. Para ello debe sub-clasificar la clase *EventHandler* e implementar el método `DoHandle`(). Llamaremos a esta clase *NotifyFriendsAboutLoggingHandler* y su responsabilidad será la de tomar a todos los amigos del usuario y notificarles que el mismo ha ingresado o salido del sistema. A su vez, a esta sub-clase le interesará sólo el evento de que un usuario cambió su estado, por lo que puede definir en su constructor el tipo de *MatchingPolicy* que utilizará. El código de la clase se puede apreciar en el Código 4.

```
public class NotifyFriendsAboutLoggingHandler extends EventHandler
	{
		public NotifyFriendsAboutLoggingHandler() :
			base(new ClassBasedMatchingPolicy(typeof(UserStateChangedEvent)))
		{
		}

		protected override void DoHandle(ContextEvent event, AdaptationEnvironment env)
		{
			UserStateChangedEvent ev = (UserStateChangedEvent) event;
			User user = ev.User;
			foreach(User u in user.GetFriends())
			{
				If(ev.LoggedIn)
				{
					u.FriendLoggedIn(user);
				}
				else
				{
					u.FriendLoggedOut(user);
				}
			}
		}
	}
```

Código 4: Definición de la clase NotifyFriendsAboutLoggingHandler

Es posible que la clase *User* notifique ante el cambio de diferentes aspectos. Para poder diferenciarlos utilizaremos un enumerativo definido dentro de la clase donde cada valor representa a aspectos distintos. Por ahora la clase tiene un solo aspecto, por lo que el enumerativo podría definirse como se puede observar en el Código 5.

```
enum Aspect
{
	UserStateChanged
};
```

Código 5: Definición del aspecto a notificar del usuario

Luego el desarrollador deberá definir en qué momento se notifica el cambio de qué aspecto. Para ello puede o convertir a la clase *User* en una sub-clasificación de la clase *ContextObservable* provista por la capa de contexto introducida en la Sección 4.3, o implementar la interfaz *IObservable* y todos los métodos que conlleva. Vamos a suponer que no existe problema en heredar de la clase *ContextObservable* y que la notificación se hará en los métodos `LogIn` y `LogOut` definidos anteriormente, en el Código 6 se puede apreciar la modificación realizada.

```
public void LogIn()
{
    loggedIn = true;
    Notify(Aspect.UserStateChanged, true);
}

public void LogOut()
{
    loggedIn = false;
    Notify(Aspect.UserStateChanged, false);
}
```

Código 7: Redefinición de los métodos LogIn *y* Log*Out*

Lo que resta hacer es construir las instancias de las clases que interactúan en todo este proceso en el momento en que se crea un usuario nuevo. El desarrollador puede optar por varias alternativas para dónde insertar el código que realice esta tarea, puede ser dentro de una clase que se encarga de manejar los usuario como *UserManager*, o utilizando el paradigma orientado a aspectos (*Aspect Oriented Programming, AOP*[6]) para interceptar la creación de los objetos, o si el sistema es una aplicación web puede insertarlo en la capa de servicios en el mismo método que se encarga de crear un nuevo usuario, etc. Sea cual sea la manera que el desarrollador elija tendrá un código similar al visualizado en el Código 8.

```
{
…
ContextEventFactory factory = new UserStateChangedEventFactory();
ContextFeature contextFeature = new ObserverContextFeature(
        "StateChange", factory, user, User.Aspect.UserStateChanged);

AwareObject awareObject = new AwareObject(user);
awareObject.AddContextFeature(contextFeature);

GetGlobalEnvironment().AddAwareObject(awareObject);
…
}
```

Código 8: Instanciación de las clases definidas

Algunos comentarios del Código 8:

- *user* es una variable que contiene al usuario recién creado.
- Se crea un *ObserverContextFeature* en lugar de un *ValueHolderContextFeature* porque la clase encargada de disparar el evento es la clase User. Recordar que la clase *ValueHolderContextFeature* se encarga de representar datos contextuales o dinámicos que no se encuentran representados dentro de las clases del dominio de la aplicación.

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., & Irwin, J. (1997). *Aspect-oriented programming*. Springer Berlin Heidelberg.

- `GetGlobalEnvironment()` es un método ficticio que sirve sólo a los propósitos del ejemplo y se encarga de recuperar el *AdaptationEnvironment* que representa al canal de difusión de todo el sistema. El desarrollador debería elegir el o los *AdaptationEnvironment* a los que quiere suscribir al usuario y encargarse de tener una referencia a ellos en el momento en que se ejecuta el código anterior.

- Una instancia de la clase *NotifyFriendsAboutLoggingHandler* debería estar presente en alguno de los *AdaptationEnvironment* en los que el desarrollador suscribirá a los nuevos usuarios. La misma tendría que ser agregada en el momento en que se crea el *AdaptationEnvironment*.