Congreso Internacional en Ingeniería Electrónica. Mem. Electro 2015, Vol. 37, pp 53-57, Chihuahua, Chih. México
http://depi.itchihuahua.edu.mx/display/memorias_electro/MemoriaElectro2015
ISSN 1405-2172

DISEÑO DE UN API DE CONTROL PARA MARIONETAS DIGITALES

Alberto Pacheco González, Elisabet González Juárez, Isidro Robledo Vega
Instituto Tecnológico de Chihuahua
División de Estudios de Posgrado e Investigación
Ave. Tecnológico 2909
Col. 10 de Mayo, Chihuahua, Chih. CP 31310
Tel. (614) 201-2012
{apacheco, egonzalezj, irobledo}@itch.edu.mx

RESUMEN.

Se presenta el diseño de una interfaz de programación de aplicaciones multi-plataforma para manipular marionetas digitales programables basadas en gráficos vectoriales. Dicha API podrá ser implementada en diversos lenguajes de programación, con la intención de soportar el uso de marionetas digitales ya sea dentro de un entorno de programación interactivo con propósitos educativos o dentro de una plataforma de animación vectorial usando uno o más sensores RGBD para la captura de movimiento. Se presentan y analizan las pros y contras de una especificación-ejecutable implementada como protocolo en el lenguaje de programación Swift.

Palabras Clave: marioneta digital, API, UML, mejores prácticas.

ABSTRACT.

We present an API design to support the multi-platform development of vector-based graphics and animations of digital puppets. The aim of this work is to build a library ported to different programming languages, to support a live coding editor and a RGBD fusion platform for vector-based animation streaming. A Swift protocol executable specification is presented evaluating its merits and limitations.

Keywords: digital puppet, API design, UML, best practices.

1. INTRODUCCIÓN

En un sistema de software tradicional es común desarrollar una interfaz de usuario pensando en que dicho sistema será usado por una persona, i.e. interacción hombre-máquina. Sin embargo, tenemos componentes y librerías de software que no terminan siendo utilizadas directamente por personas, sino por otros programas, i.e. interacción máquina-máquina [1]. Cuando se tiene definido un componente de software de acuerdo a un conjunto de funcionalidades independientemente de sus detalles o cambios de implementación, es que tenemos lo que se le conoce como interfaz de programación de aplicaciones (API).

Actualmente existe un creciente interés por el diseño de APIs, particularmente las Web APIs[1], tales como: SOAP con XML orientada a servicios (SOA), REST con JSON orientada a recursos (ROA), WebRTC orientada a procesos (RPC) o transacciones. Debido a la enorme complejidad e interacción de las aplicaciones y servicios en la red que explotan la vasta cantidad de recursos disponibles en Internet, resulta de gran importancia un enfoque de diseño que permita descomponer estos grandes sistemas en micro-componentes más genéricos, confiables, intercambiables y reutilizables [2].

Un API puede considerarse como: a) un componente de software sin interfaz de usuario; b) la especificación de un conjunto de llamadas remotas publicadas para ser consumida, interconectada o extendida por el usuario de dicha API. Existe una serie de consideraciones y principios de diseño de APIs muy afines al diseño de sistemas distribuidos con componentes debilmente acoplados [1], entre las que se pueden mencionar que una API [1-4]: a) abstrae un objeto o problema; b) se diseña teniendo en mente, que su usuario será un programador; c) debe ser modular, consistente, estable, flexible, segura, fácil de adoptar; d) es tan usable como lo sea su documentación. Un API debe incluir el conjunto de definiciones y operaciones a ofrecer detallando la firma de cada una de estas operaciones (descripción, nombre, parámetros, tipos, resultado, condiciones de error, posibles efectos colaterales, sus pre-condiciones y post-condiciones). Por otro lado, un API para un protocolo debe especificar su comportamiento general y detallar como se establece una sesión, si conserva algún estado y cuales son dichos estados, una notación específica para el tipo de mensajes y su secuenciación o manejo de transacciones, detallar el manejo de errores y el re-envio de mensajes en caso de error, niveles de seguridad y autenticación, entre otros [4].

Con respecto al dominio de aplicación, podemos referir, que si bien existen diversos trabajos sobre el desarrollo de marionetas digitales [5-7], no existe una especificación de un API que permita lograr la independencia no sólo de la aplicación, sino de su plataforma, de su soporte tecnológico (e.g. sensores RBGD) y de los lenguaje a usar para su implementación. El presente diseño de API para una marioneta digital programable forma parte de una propuesta para una plataforma de fusión de sensores RGBD para la animación de marionetas digitales (FSA-MD) [8], específicamente el API aquí propuesto establece la funcionalidad requerida para operar una marioneta digital ya sea con la información obtenida a través de la captura

[1] A mediados del 2015, el sitio de la Web Programable reportaba casi 14,000 APIs disponibles (http:// http://www.programmableweb.com).

de movimiento con un sensor RGBD [9] o a través de código escrito en un editor en-línea interactivo [10].

2. API DE MARIONETA DIGITAL PROGRAMABLE

2.1. Antecedentes.

En un trabajo previo [9], se logró manipular directamente una marioneta digital representada con imágenes vectoriales SVG, capturando en tiempo real los movimientos corporales por medio de un sensor RGB de bajo costo, de rango medio (0.8-3.5m), resolución VGA 640x480 @30fps (**fig. 1**), bajo Apple OS X 10.9 con el driver OpenKinect/libfreenect, usando el lenguaje de programación Processing de MIT y la librería OpenNI.

Figura 1. Sensor RGBD PrimeSense Carmine 1.08

La arquitectura monolítica original de dicha aplicación fue luego separada en módulos (**fig. 2**): a) un módulo de captura; b) un módulo de animación; c) un módulo de manejo de la interfaz de usuario a través de un panel de control y de visualización; d) un conjunto de imágenes SVG; e) un módulo para representar una marioneta digital, denominada interfaz de la marioneta digital programable (API-MDP).

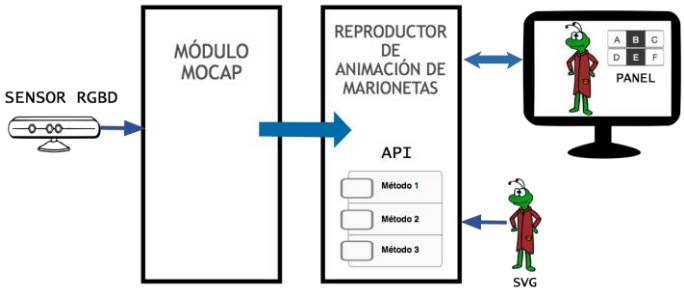

Figura 2. Arquitectura modular del sistema de captura de movimiento basado en un sensor RGBD.

Otro proyecto de código abierto consistió en modificar el entorno de programación interactivo de la plataforma de Khan Academy (**fig. 3**), para representar y manipular una marioneta digital usando el API-MDP implementado en el lenguaje de programación ProcessingJS de dicho entorno [10, 11].

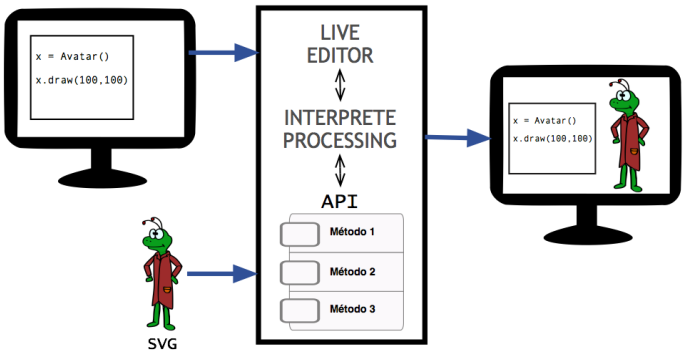

Figura 3. Arquitectura del entorno interactivo de programación basado en la plataforma de Khan Academy.

2.2. Diseño y especificación del API-MDP.

El reuso en ambos proyectos del API-MDP y el interés por portar dichas aplicaciones a diversas plataformas móviles (iOS, Android y Windows) ha tenido como consecuencia natural que se defina como objetivo central de este trabajo, una especificación formal, genérica y reutilizable del API-MDP para lograr controlar una marioneta digital de manera independiente al hardware, plataforma y lenguaje de implementación.

Dentro de la metodología seguida para especificar el API-MDP, primero se representó su diseño en notación UML 2.5 [12]. Para verificar dicha especificación se realizaron diversas implementaciones en distintos lenguajes de programación, tales como Javascript, Processing, C# y Swift. Sin embargo, cada implementación resultó ser en gran medida dependiente de cada lenguaje utilizado, particularmente en lo que se refiere a los tipos de datos nativo de cada lenguaje. Buscando una mejor correspondencia con la representación de diseño y un modelo de datos más abstracto, luego de revisar y analizar las implementaciones efectuadas, se encontró que se obtuvieron mejores resultados a través del lenguaje de programación Swift. Para lograr un prototipo más ágil y genérico se incorporó el uso de un *playground* en Xcode y la declaración de un protocolo de Swift. De esta forma fue posible mejorar con rapidez tanto la especificación en UML como su implementación, de forma muy próxima a una especificación-ejecutable [13], aspecto que se detalla a continuación.

Como muestra el diagrama UML para la especificación del API-MDP (**fig. 4**), sólo se utilizaron los tipos de datos primitivos declarados en UML 2.5 (**`Integer`**, **`Real`**, **`String`**, **`Boolean`**). Al momento de especificar la clase central **`Avatar`** se requirió la definición de la enumeración **`JointID`** (para identificar cada articulación de una marioneta) y los datos primitivos **`Frac`** (para representar una fracción entre 0.0 y 4.0) y **`Angle`** (una valor dentro del rango de -180 a 180 grados).

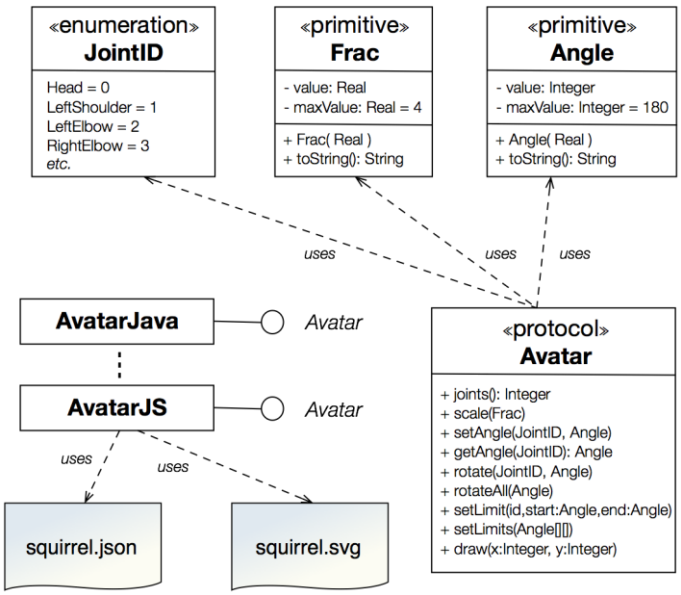

Figura 4. Representación del API MDP en UML 2.5.

Como parte de la especificación del API tenemos que para crear una marioneta digital en una plataforma en particular se debe implementar el protocolo **`Avatar`** (**fig. 4**). Como ejemplo, se muestra como la implementación particular **`AvatarJS`** en Javascript requiere a su vez de dos documentos donde se resguardan los metadatos (**`squirrel.json`**) y una representación gráfica en 2D (**`squirrel.svg`**) de la marioneta de una ardilla (**fig. 5**).

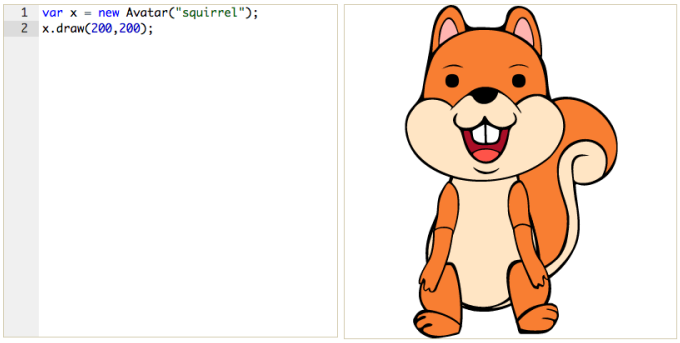

Figura 5. Ejemplo de una marioneta digital programable en el entorno de programación interactivo.

3. RESULTADOS OBTENIDOS

3.1. API MDP implementada en Swift.

A continuación se detalla como se llevó a cabo la implementación de cada dato primitivo presente en la representación de diseño en UML. Para conservar los mismos datos primitivos de UML, sólo fue necesario definir en Swift las siguientes equivalencias:

```
typealias Integer = Int
typealias Real = Float
```

Para implementar la enumeración **`JointID`** se usó:

```
enum JointID: Integer {
case Head = 0
case LeftShoulder = 1
case LeftElbow = 2
case RightShoulder = 3
// add more here...
}
```

Como puede observarse, existe un mapeo casi directo entre el diagrama UML y el lenguaje Swift. Los datos primitivos **`Frac`** y **`Angle`** fueron definidos de forma muy similar. Como se demuestra abajo para el tipo primitivo **`Angle`**, la estrategia fue definir un nuevo tipo entero[2] basado en: a) un atributo **`value`**; b) la constante del valor límite **`MaxValue`**; c) el protocolo para aceptar un valor literal entero (constructor a partir de un dato **`Integer`**); d) el protocolo para imprimir un valor (método **`description`** de sólo lectura), de forma tal que tenemos:

```
/// Angle degrees from -180º to 180º
struct Angle :
        IntegerLiteralConvertible,
        Printable
{
   var value: Integer
   static let MaxValue: Integer = 180

   var description: String {
      get {
         return String(value)
      }
   }
   init(integerLiteral v: Integer) {
      value = v % Angle.MaxValue
   }
}
```

Con estos elementos es posible soportar la especificación completa del protocolo Avatar en Swift, como se demuestra a continuación:

```
protocol Avatar {
  func joints() -> Integer
  func scale(size:Frac)
  func setAngle(id:JointID, angle:Angle)
  func getAngle(id:JointID) -> Angle
```

[2] En Swift un tipo entero es una clase que implementa diversos protocolos. Para más detalle vea http://swiftdoc.org/type/Int/hierarchy

```
    func rotate(id:JointID, angle:Angle)
    func rotateAll(angle:Angle)
    func setLimit(id:JointID,
                  start:Angle, end:Angle)
    func setLimits(list:[[Angle]])
    func draw(x:Integer, y:Integer)
}
```

3.2. Ventajas y Limitantes.

Gracias al mecanismo de protocolos, sinónimos de tipos, derivación de nuevos tipos primitivos y la edición interactiva de los *playgrounds* del lenguaje Swift fue posible implementar y validar de manera muy ágil la especificación representada en UML del API-MDP. Esta forma de diseño y validación inmediata de una especificación permitió perfeccionar de manera iterativa e interactiva el modelo propuesto, proceso que permitió refinar y depurar una gran cantidad de errores y omisiones que existían en la especificación original, gracias a que la especificación-ejecutable en Swift fue revisada sintácticamente de manera completa y exhaustiva por el propio intérprete del lenguaje.

Otro beneficio adicional fue el poder documentar de manera estructurada la implementación del API mediante la adopción de un formato particular para comentar código en Swift, mismo que el entorno Xcode reconoce y despliega como ayuda (*quick help tool*) cuando se coloca el cursor y se pulsa la tecla Alt ⌥ sobre el nombre de una definición de clase, protocolo, estructura o función, tal y como se demuestra en la **fig. 6**.

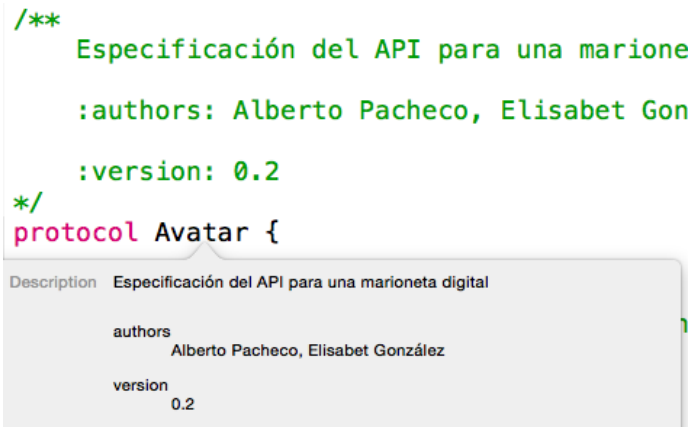

Figura 6. Ayuda emergente en Xcode derivada a partir de los comentarios que anteceden a la definición del protocolo.

Como se muestra en la **fig. 7**, es posible detallar cada parámetro de una función, una breve descripción y opcionalmente un código de ejemplo de uso. Aplicando este método se logró documentar de manera detallada, completa y ágil la especificación implementada en Swift [14].

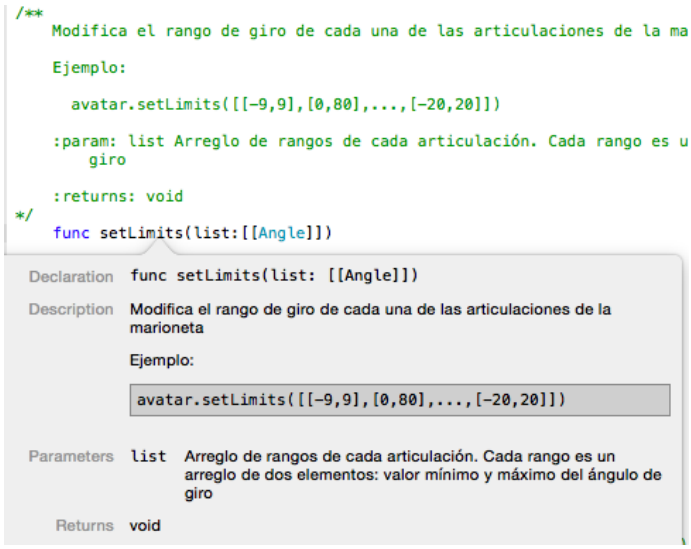

Figura 7. Ayuda rápida para un método de clase incluyendo descripción, un ejemplo, parámetros y valor de retorno.

Existen sin embargo, algunos aspectos a mejorar que corresponden al propio lenguaje y su entorno. A pesar de la facilidad que ofrece el lenguaje para extender o sintetizar tipos de datos primitivos, es conveniente contar con un mecanismo más simple o de alto nivel para mapear valores restringidos a un determinado rango, tal como fue el caso de **`Frac`** (un valor real limitado a un rango de 0.0 a 4.0) y **`Angle`** (un valor entero entre -180 a 180). Una posible propuesta sería algo similar al tipo subrango de Pascal (**`Type byte = 0…255`**), que en caso de Swift pudieran ser:

```
typedef  Frac: Real    =  0.0...4.0
typedef Angle: Integer = -180...180
```

Otro problema se refiere al entorno Xcode y el uso de *playgrounds*, mismo que ofrece un formato de comentarios (`//:`) para convertir a texto enriquecido usando la sintaxis de la especificación Markdown [15], misma que es incompatible con la notación para documentar las declaraciones usando otro formato de comentarios (`/**`). Otra limitante fue la ausencia de una opción del entorno Xcode para exportar en un archivo toda la documentación especificada (esta funcionalidad será incluida en la próxima versión de Xcode).

4. CONCLUSIONES.

Definir la especificación de un API neutral a la plataforma y bien documentada ha demostrado ser una tarea importante y no trivial si se busca un diseño neutral, modular y reusable. Sin embargo, resulta complicado lograr una correspondencia y una validación entre el diseño representado en la notación visual de UML y cualquiera de los lenguajes de programación que dan

pie a muy diversas implementaciones muy dependientes de cada uno de dichos lenguajes. Aunque sin lograr la perfección, la especificación de protocolos del lenguaje Swift ha demostrado ser una valiosa herramienta para representar casi de manera directa una especificación basada en UML 2.5, misma que ofrece una rigurosa verificación sintáctica y la posibilidad de generar la importante documentación del API [3]. Con esta nueva API-MDP se espera desarrollar una plataforma de animación vectorial basada en la fusión de múltiples flujos de sensores RGBD [8] y también ampliar un entorno interactivo de programación para controlar y manipular marionetas digitales programables [10].

Referencias.

[1] Biehl, M. API Architecture: The Big Picture for Building APIs, 1st edition. CreateSpace Independent Publishing Platform, 2015.

[2] M. Reddy, *API Design for C++*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[3] Niño, J., "Introducing API Design Principles in CS2," *J. Comput. Sci. Coll.*, vol. 24, no. 4, pp. 109–116, Apr. 2009.

[4] Fielding, R. & Taylor, R. "Principled Design of the Modern Web Architecture," *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002.

[5] Leite, L. & Orvalho, V. Anim-actor: understanding interaction with digital puppetry using low-cost motion capture. In *Proc. of the 8th Int. Conf. on Advances in Computer Entertainment Technology* (p. 65). ACM, 2011. Retrieved from http://dl.acm.org/citation.cfm?id=2071505

[6] Shum, H. & Ho, E. S. L. Real-time Physical Modelling of Character Movements with Microsoft Kinect. In *Proc. of the 18th ACM Symposium on Virtual Reality Software and Technology* (pp. 17–24). New York, NY, USA: ACM, 2012. http://doi.org/10.1145/2407336.2407340

[7] Zhang, Y., Han, T., Ren, Z., Umetani, N., Tong, X., Liu, Y., Cao, X. BodyAvatar: Creating Freeform 3D Avatars Using First-person Body Gestures. In *Proc. of the 26th Annual ACM Symposium on User Interface Software and Technology* (pp. 387–396). New York, NY, USA: ACM, 2013. http://doi.org/10.1145/2501988.2502015

[8] Pacheco, A. & González, E. Plataforma de fusión de sensores RGBD para la Animación de Marionetas Digitales. *En Memorias del Encuentro Nac. de Ciencias de la Computación (ENC),* Ensenada, B.C., UABC, 2015.

[9] Pacheco, A., Ramírez, M., Guzmán, C., González, E. *Marionetas Digitales: Tecnología Emergente para Narrar Historias con Personajes Animados mediante la Captura Digital de Movimiento*, en Tecnologías Emergentes en la Educación de Cruz, R., López, G., Pacheco, A., Pearson, México, 2015.

[10] Pacheco, A., Robledo, I., González, E. Marioneta Digital Programable: Un Entorno Interactivo en Línea para la Introducción a la Programación. *En Memorias del Encuentro Nacional de Ciencias de la Computación (ENC)*, Ensenada, B.C., UABC, 2015.

[11] Pacheco, A. "Demo del entorno de programación interactivo para manipular marionetas digitales programables" [Online]. Disponible (2015): http://podcast.itch.edu.mx/live-editor

[12] Object Management Group. "Unified Modeling Language™ (UML®)," [Online]. Disponible (2015): http://www.omg.org/spec/UML/

[13] Anderson, A. & Shaw, G. "Executable Requirements and Specifications," *J. VLSI Signal Process. Syst.*, vol. 15, no. 1–2, pp. 49–61, Jan. 1997.

[14] Pacheco, A. "Especificación del API para Marioneta Digital Programable usando Swift" [Online]. Disponible (2015): http://bit.ly/1EWVsSJ

[15] GitHub. "Mastering Markdown," [Online]. Disponible (2015): https://guides.github.com/features/mastering-markdown