



Universidad Internacional de La Rioja  
Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Ingeniería de Software y Sistemas  
Informáticos

# Simplificando el desarrollo integrado en un motor de videojuegos de Lenguajes de Dominio Específico

Trabajo fin de estudio presentado por:	González Martínez, Pablo
Tipo de trabajo:	Desarrollo práctico
Director/a:	Barcelona Liédana, Miguel Ángel
Fecha:	Septiembre 2021

## Resumen

Los motores de videojuegos se han beneficiado de un constante incremento de sus capacidades tecnológicas y de su facilidad de uso para el desarrollo de software, con una consecuente expansión de su aplicabilidad hacia sectores diferentes al entretenimiento, como los sectores industria y defensa. Es posible abstraer la implementación de la lógica de comportamiento de sus aplicaciones a través de opciones como el *scripting*, tanto mediante lenguajes en código como de programación visual, así como lógica dirigida por datos modificables mediante opciones de configuración y edición usando herramientas dedicadas. El uso de Lenguajes de Dominio Específico (DSL) es común en el desarrollo de videojuegos, pero rara vez se plantea el propio desarrollo de nuevos DSL como una parte del desarrollo de aplicación. En este trabajo se comprueba el potencial de incluir de manera integrada en un motor de videojuegos la creación, mantenimiento, ejecución y edición de nuevos DSL a través de un conjunto software que permita un desarrollo de lenguajes simplificado, priorizando la facilidad de uso y reduciendo las necesidades de conocimiento de este proceso, exponiéndose como una posible alternativa a las soluciones de desarrollo más predominantes para la creación de lógica, su edición y su configuración, con el objetivo de cumplir con los requisitos más comunes en los nuevos ámbitos de aplicabilidad de la tecnología de videojuegos. El software resultante provee de un entorno integrado en un motor de videojuegos que facilita el desarrollo de lenguajes de alto nivel declarativos con características imperativas con total interoperabilidad con el resto de código de aplicación, ofreciendo un interfaz de programación de uso simple que reduce los requisitos de conocimiento para permitir el desarrollo de lenguajes de alto nivel, e incluye un editor de código integrado en el motor con facilidades de uso para perfiles de baja tecnicidad. A través de una prueba de uso en un proyecto demo se muestra su efectividad, simplicidad, facilidad de uso y capacidad de extender las opciones de edición y configuración, mostrándose como una alternativa viable al desarrollo de herramientas dedicadas y pudiéndose contemplar como una opción integral en el desarrollo de proyectos con tecnología de videojuegos para suplir necesidades de implementación lógica de alto nivel.

**Palabras clave:** Motores de videojuegos – Lenguajes de Dominio Específico - Scripting

## Abstract

Videogame engines have benefited from a constant increase in their technological capabilities and ease of use for software development, with a consequent expansion of their applicability to other sectors other than just entertainment, such as the industrial and defense sectors. In game development, it is common to abstract the implementation of the behavioral logic through such options as scripting with high-level General-Purpose Languages (GPL), both code-based and visual programming, as well as data-driven logic, modifiable through configuration and editing options using dedicated tools. The use of Domain-Specific Languages (DSL) is common in game development, but the development itself of new DSLs is rarely considered as an integral part of application development. This work aims to establish the potential of integrating the creation, maintenance, execution and edition of new DSLs in a video game engine through a software package that allows simplified language development, prioritizing ease of use and reducing the required knowledge for this process, being exposed as a possible alternative to the most prevalent development solutions for the creation of logic, its editing and its configuration, with the aim of complying with the most common requirements in the new fields of applicability of video game technology. The resulting software provides a package integrated into a video game engine that can be used to develop high-level declarative languages with imperative features with full interoperability with the rest of the application code, offering a simple-to-use programming interface that reduces the knowledge requirements to facilitate this development, and includes a code editor integrated into the engine with user-friendly features for low-technical profiles. Through a test of use in a demo project, its effectiveness, simplicity, ease of use and ability to extend the editing and configuration options are showcased, demonstrating this software as a viable alternative for the development of dedicated tools, allowing it to be considered as an integral option in the development of projects with videogame technology to implement high-level logical behaviour.

**Keywords:** Game engine – Domain Specific Languages - Scripting

## Índice de contenidos

Introducción .....	9
1.1. Justificación .....	10
1.2. Planteamiento del problema.....	13
1.3. Estructura del trabajo .....	14
2. Contexto y estado del arte .....	16
2.1. Introducción .....	16
2.2. Estado del arte de Lenguajes de Dominio Específico .....	16
2.3. Estado del arte de lenguajes Integrados para videojuegos .....	18
2.3.1. Scripting de lógica.....	18
2.3.2. Línea de comandos .....	20
2.3.3. Lenguajes de shading.....	23
2.3.4. SQL y declaraciones .....	25
2.3.5. Lenguajes de marcado y estilo .....	27
2.4. Conclusiones.....	28
3. Objetivos concretos y metodología de trabajo .....	31
3.1. Objetivo general .....	31
3.2. Objetivos específicos .....	31
3.3. Metodología del trabajo .....	32
4. Desarrollo específico de la contribución.....	35
4.1. Identificación de requisitos .....	35
4.2. Descripción del sistema software desarrollado .....	39
4.2.1. Introducción .....	39
4.2.2. Declaraciones, invocaciones y estado .....	40
4.2.3. Lenguaje, Scripts y Decllexicon .....	44

4.2.4.	Compilación y ejecución de scripts.....	49
4.2.5.	Complejidad aditiva: <i>Statements</i> .....	53
4.2.6.	Complejidad aditiva: <i>Expressions</i> .....	58
4.2.7.	Implementación para Unity .....	60
4.2.8.	Editor de texto integrado en Unity .....	65
4.2.9.	Prueba de uso: Introducción .....	72
4.2.10.	Prueba de uso: Demo .....	76
4.2.11.	Prueba de uso: Integración.....	79
4.3.	Evaluación .....	85
4.3.1.	Integración en motor, interoperabilidad y dependencias.....	85
4.3.2.	Simplicidad, requisitos de conocimiento de interfaz y facilidad de uso .....	86
4.3.3.	Flexibilidad, modularización y capacidad de adaptación .....	89
4.3.4.	Prueba de integración de uso .....	90
5.	Conclusiones y trabajo futuro .....	91
5.1.	Conclusiones.....	91
5.2.	Trabajo futuro.....	92
	Referencias bibliográficas .....	95
Anexo A.	Artículo .....	99
Anexo B.	Enlace a repositorio.....	107

## Índice de figuras

Figura 1. <i>Segmento de un archivo json de edición deserializable a C#.</i> .....	13
Figura 2. <i>Sistema de programación visual Blueprints.</i> .....	20
Figura 3. <i>Ejemplo de comandos activados en la consola de un videojuego.</i> .....	21
Figura 4. <i>Ejemplo en código C# de una función intérprete de comandos.</i> .....	22
Figura 5. <i>Función en lenguaje OpenGL.</i> .....	24
Figura 6. <i>Grafo de nodos en el editor visual de materiales de Unreal Engine 4.</i> .....	25
Figura 7. <i>Ejemplo de sentencias SQL.</i> .....	26
Figura 8. <i>Ejemplo de UXML.</i> .....	27
Figura 9. <i>Ejemplo de USS.</i> .....	28
Figura 10. <i>Diagrama de iteración en el proceso de desarrollo.</i> .....	32
Figura 11. <i>Delegados de Acolyte.</i> .....	40
Figura 12. <i>Diagrama de clases para palabras declarativas.</i> .....	40
Figura 13. <i>Diagrama de estructura de Lexemas Declarativos Enlazables.</i> .....	41
Figura 14. <i>Código ejemplo de construcción de declaración básica.</i> .....	42
Figura 15. <i>Reglas y grupos desordenados en Xtext.</i> .....	43
Figura 16. <i>Código de ejemplo de palabra tolerada.</i> .....	44
Figura 17. <i>Diagrama de clases de Script, Language y Decllexicon.</i> .....	46
Figura 18. <i>Código de funciones abstractas de Decllexicon.</i> .....	47
Figura 19. <i>Clases anidadas en decllexicon para su gestión modular.</i> .....	47
Figura 20. <i>Fragmento de código con modularización de Decllexicon.</i> .....	48
Figura 21. <i>Diagrama de clases de ejecución de script.</i> .....	49
Figura 22. <i>Código de ejecución dentro de la clase Script.</i> .....	50
Figura 23. <i>Diagrama de actividad del proceso de compilación.</i> .....	51
Figura 24. <i>Fragmento de código con proceso de compilado de tolerancia y literales.</i> .....	52

Figura 25. <i>Diagrama de clases de sentencias Statement</i> .....	53
Figura 26. <i>Fragmento de código con ejemplo de Statement</i> . ....	54
Figura 27. <i>Diagrama de clases de instrucciones</i> .....	54
Figura 28. <i>Fragmento de código con compilación de Statements</i> .....	55
Figura 29. <i>Diagrama de actividad de compilación de Statements</i> . ....	56
Figura 30. <i>Ejemplo de flujo de instrucciones en sentencia IF/ELSE</i> .....	57
Figura 31. <i>Código para el añadido de Statement</i> . ....	57
Figura 32. <i>Fragmento de código con función IF de sentencia IfElse</i> . ....	58
Figura 33. <i>Diagrama de clases de Expressions</i> . ....	59
Figura 34. <i>Fragmento de código con uso de Expressions</i> .....	59
Figura 35. <i>Diagrama de clases para el manejo de scripts como assets</i> . ....	61
Figura 36. <i>Código de generación de script de ScriptAsset</i> .....	61
Figura 37. <i>Código de generalización de ScriptAsset</i> . ....	62
Figura 38. <i>Diagrama de clases de identificación en Unity</i> . ....	63
Figura 39. <i>Imagen de contenedor de objetos en escena de Unity</i> .....	64
Figura 40. <i>Fragmento de código con el uso de UnityIdentifier</i> .....	64
Figura 41. <i>Fragmento de código con análisis de identificador para renderizar</i> .....	67
Figura 42. <i>Diagrama de clases para editar y renderizar un script</i> . ....	68
Figura 43. <i>Editor de texto de Acolyte</i> . ....	69
Figura 44. <i>Diagrama de clases para contextualización en editor</i> .....	70
Figura 45. <i>Ejemplos de contextualización en editor de Acolyte</i> . ....	71
Figura 46. <i>Captura de editor de Acolyte</i> .....	72
Figura 47. <i>Diagrama de clases de entidades y propiedades</i> . ....	73
Figura 48. <i>Diagrama de clases de procedimientos</i> . ....	74
Figura 49. <i>Assets de edición de Procedure y StepAction</i> . ....	75

Figura 50. <i>Fragmento de código con ejecución de acciones de paso.</i> .....	76
Figura 51. <i>Diagrama de clases de interacción en la demo.</i> .....	78
Figura 52. <i>Captura de ejecución de procedimiento demo.</i> .....	78
Figura 53. <i>Captura de interfaz de depuración de procedimientos.</i> .....	79
Figura 54. <i>Fragmento de código con ejecución de acciones de paso usando script.</i> .....	81
Figura 55. <i>Fragmento de código con creación modularizada de Expressions en demo.</i> .....	82
Figura 56. <i>Fragmento de código con declaración de eventos en demo.</i> .....	84
Figura 57. <i>Código ejemplificando StepScript en demo.</i> .....	85
Figura 58. <i>Imagen de aplicación demo final.</i> .....	85



## Introducción

Un motor de videojuegos comprende un elaborado conjunto tecnológico que permite la construcción de complejas simulaciones dinámicas ejecutadas en tiempo real. Aunque enfocados en la creación de videojuegos para el entretenimiento, tienen una creciente relevancia para la creación de aplicaciones en tiempo real usables en otros ámbitos y sectores, mostrando capacidades tecnológicas con un incremento de la accesibilidad y sencillez en el desarrollo, lo que facilita su uso por parte de una gran variedad de usuarios con diferentes perfiles. Research and Markets (2020) indica en su pronóstico el incremento de la demanda de los motores de videojuegos en otras industrias, en gran medida gracias a que múltiples motores tienen como objetivo ofrecerse como productos universales con los que sea posible desarrollar videojuegos y aplicaciones en tiempo real que cumplan con el mayor número de requisitos posible; y menciona sectores como salud, arquitectura, educación, banca, industria del motor, logística, e industria aeroespacial y defensa.

Los motores de videojuegos generalmente se crean y extienden con código fuente implementado con lenguajes de la familia C, de bajo nivel y compilados, siendo C++ el más usado -un uso achacado principalmente a sus requisitos de rendimiento y manejo de memoria-, pero el uso de *scripting* está muy extendido para el desarrollo de la lógica que da funcionamiento a la aplicación construida sobre el motor (Politowski et al, 2020, p. 10).

El proceso de desarrollo de esta lógica depende de los perfiles que se considere óptimo involucrar. La posibilidad de que usuarios con conocimientos reducidos de programación puedan modificar procesos lógicos o incluso crear estos procesos desde cero es posible gracias al uso de programación con *scripts* mediante lenguajes de más alto nivel de abstracción, contando además algunos motores con el uso de lenguajes de programación visual. No obstante, el uso del motor por parte de estos perfiles de menor tecnicidad en los ámbitos de codificación está dominado por las implementaciones software denominadas herramientas; software integrado en el motor que está dedicado a capacitar la creación y modificación del contenido de las aplicaciones desarrolladas requiriendo de una previa implementación de lógica dirigida por datos, que serán el objeto de modificación por parte de dichas herramientas.

En una publicación oficial de desarrollo, Blizzard Entertainment (2018) especifica que *World of Warcraft*, su popular videojuego multijugador masivo de rol (MMORPG) lanzado en 2004 y que continúa con una notable presencia en el mercado a día de hoy, se trata de un juego con una ingeniería muy “*data-driven*”, un término de uso común que puede traducirse como dirigido o impulsado por datos, e indica que esto se fundamenta en código flexible cuyo comportamiento específico está controlado por información contenida en bases de datos<sup>1</sup>, con aspectos de jugabilidad y las reglas que afectan a sus interacciones siendo definidos por diseñadores y artistas mediante datos.

Para permitir este tipo de desarrollo es fundamental que existan fuertes abstracciones entre las complejidades de bajo nivel de abstracción gestionada por los ingenieros de software experimentados y las implementaciones facilitadas al resto de diferentes perfiles de usuarios según su capacitación técnica, por lo que debe diseñarse y desarrollarse el software no sólo de cara a la implementación base que hace uso de los datos, sino a todo el conjunto tecnológico que permita el flujo de trabajo al nivel de abstracción que sea necesario para cada tarea.

### 1.1. Justificación

El desarrollo de videojuegos presenta una naturaleza multidisciplinar con diversas áreas de conocimiento específico, incluyendo toda una gama de aspectos técnicos que conciernen al diseño, arte, sonido, narrativa o jugabilidad. La creación de herramientas para realizar configuraciones y ediciones en este desarrollo es un proceso laborioso, en mayor medida si se quiere obtener un resultado robusto y con un alto grado de usabilidad para sus usuarios del dominio específico.

Podemos identificar en estas herramientas un nivel de especificidad en relación a su aplicabilidad al que podemos asociar un gradiente de posibilidades entre dos opciones: ser de uso generalizado con un alto potencial de reutilización dentro de un dominio, siendo este el principal foco de las herramientas integradas en los motores generalistas; o tener una relación inherente con la construcción de un software concreto sin utilidad fuera de este, ofreciendo

---

<sup>1</sup> El término *data-driven* no implica como requisito que los datos usados se almacenen en bases de datos.

una rentabilidad mucho menor por el tiempo y coste invertido en el desarrollo de la herramienta, pero con mayor efectividad para resolver problemas concretos, en los casos más extremos siendo de utilidad únicamente en un solo proyecto.

Dan Staines publica en 2017 un artículo para IGN dedicado a los «*Tools Programmers*», los programadores de herramientas, cuyo trabajo es la creación, mantenimiento y extensión de herramientas internas para los proyectos como una parte integral de los equipos desarrolladores en los estudios de gran tamaño, lo que sirve para poner de manifiesto la importancia e inversión necesaria para su desarrollo.

Para la creación, configuración y edición<sup>2</sup> del comportamiento de las aplicaciones software en los nuevos sectores de aplicabilidad para la tecnología de videojuegos, se puede estimar que los siguientes puntos, poco comunes en el sector del entretenimiento, tengan un impacto sobre los requisitos establecidos para un proyecto:

- ▶ El software debe estar diseñado en función a dominios complejos de alta tecnicidad que no guardan relación con los dominios específicos que son comunes en los procesos de desarrollo usando tecnología de videojuegos.
- ▶ El software debe proporcionar capacidades de edición y configuración en un dominio concreto a sus usuarios finales, siendo estos personal de diferente capacitación técnica sin ningún conocimiento de tecnología de videojuegos, pero especializados en otras áreas como la ingeniería, manufactura, administración o informática.
- ▶ El software es una parte integrada en un conjunto, desarrollado entre múltiples equipos o empresas colaboradoras. Los programadores de otros equipos o empresas, sin conocimientos en tecnología de videojuegos, requieren de realizar cambios en apartados de funcionamiento lógico para garantizar la integración y su soporte.
- ▶ El software desarrollado tendrá un uso durante un período de tiempo muy superior al que cabe esperar en un videojuego sin que esto implique nuevos contratos de desarrollo ni mantenimiento activo por parte del equipo desarrollador, algo que puede ser común en contratos de algunos sectores, como industria y defensa.

---

<sup>2</sup> Configuración refiere al conjunto de datos preexistentes que determinan el comportamiento del software según un valor variable dentro de unos límites preestablecidos. Edición refiere a la extensión dinámica del comportamiento del software mediante el añadido de conjuntos de nuevos datos interpretables por la implementación nativa. Ambos se refieren a la capacidad de modificar el software mediante datos variables.

- El software requiere la adecuación de sus interfaces gráficas a un diseño determinado por la imagen visual de un cliente intermedio o final, pudiendo imponer también necesidades de usabilidad concretas, aumentando los esfuerzos de comunicación y teniendo un impacto en el coste del desarrollo de cualquier herramienta destinada al uso por el cliente.

Intentar cumplir con requisitos influenciados por estas observaciones a través de proporcionar al resto de *stakeholders* acceso al motor de videojuegos utilizado para el desarrollo es una solución deficiente, ya que tendrá inherentemente una sobrecarga de complejidad, requerirá del uso de un software genérico adicional de terceros si utilizamos un motor comercial generalista, no provee de un conjunto tecnológico integrado, adaptable y flexible con las necesidades concretas del proyecto, y además necesita de realizar de nuevo el proceso de construcción de la aplicación y su actualización en cada dispositivo para ver reflejado cualquier cambio.

Otras opciones como el uso de *scripting* mediante Lenguajes de Propósito General (GPL) interpretados que sean editables después de la compilación son efectivos en eliminar la necesidad de usar el motor y recompilar, pero seguirán teniendo una sobrecarga de complejidad y falta de especificidad y adecuación al uso, por lo que se estima que para la mayoría de los dominios no serán una solución efectiva.

El desarrollo de herramientas integradas en el editor del motor de videojuegos ya puede suponer una inversión de desarrollo significativa a lo largo de un proyecto software, por lo que requerir que estas herramientas estén a disposición fuera del editor del motor puede ser todavía más costoso y complejo al carecer de las utilidades integradas que ofrezca el editor, teniendo esencialmente que replicar estas funcionalidades integradas con nuestro propio código para que estén disponibles en el software ejecutable construido, y algunos proyectos nos dificultarán la reutilización de herramientas propias debido a su especificidad y/o a las necesidades de adecuarse a las decisiones técnicas y de diseño visual impuestas por el cliente.

Para manejar la configuración o edición del software utilizando estas herramientas integradas se utilizan archivos que soporten la serialización de datos a partir de la implementación nativa, lo que suele implicar orientación a objetos y una estructuración de entidades y componentes, que puedan ser fácilmente deserializables al código nativo original, como pueden ser archivos

json o xml. Estos archivos pueden sufrir corrupciones irreversibles y son susceptibles a problemas de retrocompatibilidad ante cualquier refactorización o pequeño rediseño del código, especialmente durante fases de prototipado, y se requerirá de recursos de desarrollo para mitigar los riesgos. La figura 1 muestra un archivo json serializable y deserializable por el código compilado de una aplicación creada con el motor de videojuegos Unity en lenguaje C#, donde se puede observar susceptibilidad ante cambios de refactorización o rediseños.

**Figura 1.** Segmento de un archivo json de edición deserializable a C#.

```
"children": {
  "$type": "DigitalTwin.SceneObject+SerializationData[], Assembly-CSharp",
  "$values": [
    {
      "$type": "DigitalTwin.Compositor+SerializationData, Assembly-CSharp",
      "components": {
        "$type": "DigitalTwin.IDigitalTwinComponentData[], Assembly-CSharp",
        "$values": [
          {
            "$type": "DigitalTwin.VisualBehaviourComponentData, Assembly-CSharp",
            "objectName": "My Object Example"
          }
        ]
      }
    }
  ]
},
```

Fuente: Elaboración propia, 2019.

## 1.2. Planteamiento del problema

Los Lenguajes de Dominio Específico (DSL) son lenguajes de programación adaptados a resolver problemas en un dominio de aplicación concreto en vez de proporcionar funcionalidades de propósito general, permitiendo mayor expresividad y facilidad de uso (Mernik et al, 2005). A lo largo de la historia de los videojuegos, se ha hecho uso de lenguajes de dominio específico para su desarrollo, algunos de ellos originados en el propio ámbito de los videojuegos y otros de uso generalizado para dominios comunes fuera de este.

Sin embargo, no se considera la creación y uso de nuevos DSL integrados como una parte regular del proceso de desarrollo de cada aplicación software, con el objetivo de aliviar las necesidades de desarrollo de herramientas para configuración, edición o lógica de alto nivel, que puede ser especialmente relevante cuando se deban adecuar a dominios que no tengan

relación con el proceso de creación del software, sino con su objetivo funcional una vez construido.

La complejidad del desarrollo de DSL, su necesidad de herramientas externas y soluciones de integración, y la existencia de requisitos que pueden imponer un alto nivel de especificidad y poca reusabilidad en las soluciones hacen que sea complicado para un equipo desarrollador optar por esta vía.

Se propone realizar un trabajo práctico que simplifique e integre la capacidad de desarrollar Lenguajes de Dominio Específico textuales con un alto nivel de abstracción como parte del código de aplicación sobre un motor de videojuegos, sirviendo para especificar la configuración y edición del comportamiento lógico.

Su objetivo es el de transformar el proceso de diseño y desarrollo de DSL en un proceso más sencillo, directamente integrado con el resto de código de aplicación, y accesible para cualquier programador de tecnología de videojuegos reduciendo al mínimo los requisitos de conocimiento para su uso. Se expone como una alternativa ocasional durante el ciclo de vida de desarrollo del software al desarrollo de herramientas dedicadas, al uso de lenguajes de *scripting* de propósito general, o al uso de DSL creados con herramientas externas, permitiendo abstraer la complejidad de los procesos y las implementaciones, facilitar el *scripting* de alto nivel con un alto grado de expresividad y especificidad, mejorar la accesibilidad al desarrollo integrado de lenguajes simples reduciendo los requisitos de conocimiento, y estimando que puede ser de especial utilidad para reducir los recursos de desarrollo necesarios para cumplir con los requisitos de los proyectos software para los sectores de aplicabilidad de la tecnología ajenos al entretenimiento, pudiendo mejorar sus tiempos, costes y tareas de prototipado o validación.

### 1.3. Estructura del trabajo

Se presenta en el capítulo 2 una exploración del estado del arte de DSL y su relación con los videojuegos, finalizando con unas conclusiones aplicables al desarrollo.

El posterior capítulo 3 incluye los objetivos del trabajo, así como la metodología de desarrollo seguida.

En el capítulo 4 se identifican los requisitos y se realiza una descripción del software desarrollado, que culmina en una evaluación final de los resultados.

Se finaliza con la inclusión de las conclusiones y el trabajo futuro a realizar a partir del caso práctico desarrollado en el capítulo 5.

## 2. Contexto y estado del arte

### 2.1. INTRODUCCIÓN

El uso de DSL está presente en la tecnología de videojuegos, siendo algunos de ellos ubicuos en prácticamente cualquier desarrollo moderno, como los lenguajes de *shading* necesarios para los gráficos acelerados por hardware, otros de uso generalizado fuera del ámbito de los videojuegos e integrados en algunos proyectos para tareas concretas, y otros diseñados para facilitar el proceso del propio desarrollo en motores o entornos concretos con el objetivo principal de reducir la necesidad de programadores experimentados para la implementación de la lógica de un videojuego, incrementando la accesibilidad al desarrollo con lenguajes más sencillos y abstractos. Rara vez se contempla el desarrollo de nuevos DSLs como una parte fundamental del ciclo de vida de un videojuego, en gran medida debido a la dificultad que supone esta tarea en un sector que ya sufre de tiempos y costes limitantes, además de grandes necesidades de prototipado e iteración.

### 2.2. ESTADO DEL ARTE DE LENGUAJES DE DOMINIO ESPECÍFICO

En la literatura existente acerca de DSL, se puede observar un punto de vista similar al expuesto en el interés por su utilización. En una revisión sistemática acerca de las fases de evaluación y análisis de dominio en los procesos de desarrollo de DSL, Barisic et al (2015) muestran que, aunque en la literatura rara vez se evalúan los DSL desarrollados durante el análisis de dominio, no reportando inclusión de los usuarios finales o consideraciones de diferentes casos de uso, sí que se especifica el interés en tener como usuario objetivo perfiles no-programadores y contribuir a la facilidad de uso. No obstante, la creación de un nuevo DSL no es una tarea trivial, y requiere del uso de múltiples herramientas dedicadas al proceso de desarrollo y mantenimiento de un lenguaje, incluyendo generadores de código, validadores, comprobadores de modelo y analizadores léxicos, semánticos y sintácticos, y la diversidad de estas herramientas hace que sea difícil establecer qué características son necesarias para construir un lenguaje en un contexto específico (Iung et al, 2020). Mernik et al (2005) indican la complejidad del desarrollo de DSL debido a que requiere conocimientos de dominio y desarrollo de lenguajes, incluye técnicas más variadas que las existentes para GPL, y cuyas



tareas de soporte, estandarización y mantenimiento pueden volverse un problema al que asignar recursos significativos. La elección de desarrollar un DSL puede hacerse evidente únicamente después de haber realizado una inversión considerable en el desarrollo de un software para un dominio. Entre las desventajas de uso de DSL, van Deursen et al (2000) listan, entre otras, su coste de diseño, implementación y mantenimiento.

### **Orientación a lenguajes en el desarrollo**

El enfoque conocido como dirigido por lenguajes (Language-Driven) está centrado en el diseño de DSL adaptado al sistema en construcción como parte del desarrollo, basándose en la obtención de una definición formal del problema a partir de un diseño de dominio, que se utiliza para diseñar su sintaxis, semántica y pragmática (Mauw et al, 2004).

Moreno-Ger et al (2006) muestran un ejemplo de este enfoque aplicado a videojuegos, pero consideran el dominio específico al propio desarrollo de juegos, especificando que la semántica operacional del lenguaje es la base del desarrollo del motor de videojuegos: «La actividad de implementación del lenguaje trata a su vez con la construcción de un motor de juego. Este motor estará basado en la descripción del lenguaje producida durante el diseño de lenguaje» (p. 5). Consideran el lenguaje construido como una herramienta de autoría de contenido para el desarrollo, y notan la necesidad de un enfoque incremental para permitir adaptación a diferentes géneros de videojuegos.

El término Programación Orientada a Lenguajes (LOP) es utilizado por múltiples personas para describir diferentes enfoques de programación, aunque todos ellos con puntos en común. Ward (1994), lo describe como basar el desarrollo de un programa en el desarrollo de un lenguaje especificado formalmente, orientado a dominio, de muy alto nivel, con dos fases independientes en el desarrollo, la implementación del sistema con el lenguaje y la implementación de un compilador, transpilador o interpretador para el lenguaje utilizando tecnología existente. Dmitriev (2004) describe el LOP como la elección de uno o varios DSL existentes, o el desarrollo de uno o varios DSL si es necesario, para llevar a cabo el desarrollo de un programa, a partir del modelado conceptual realizado por el programador, indicando que es una manera mucho más efectiva de programar, ya que las soluciones a los problemas suelen modelarse de una manera no imperativa mediante instrucciones, y los lenguajes de

propósito general requieren traducir conceptos de alto nivel en el dominio en características de programación de bajo nivel.

Geisler et al (2019) desarrollan un DSL de alto nivel con características de programación orientada a aspectos (AOP) destinado a su uso durante el desarrollo de videojuegos con el motor de videojuegos Unreal Engine 4, con el objetivo de reducir la duplicación de código además de proporcionar una plataforma AOP agnóstica del código de juego, pudiendo generar código C++, LUA, Blueprints o Skookum Script, para poder funcionar en el motor. El DSL es creado mediante Xtext, una herramienta que permite el desarrollo de lenguajes de programación y DSL. Xtext provee de un potente entorno con analizador sintáctico, enlazador, comprobador de tipos y compilador, con la creación del lenguaje estando definida en su propio DSL Grammar Language que permite describir sintaxis concreta y cómo está mapeada al modelo semántico (Eclipse, 2021).

Otros estudios notan la ayuda que supone un DSL para el desarrollo de videojuegos orientados a sectores que no priorizan el entretenimiento, como la educación, permitiendo involucrar en mayor medida a los educadores en el desarrollo y mejorar los esfuerzos de comunicación (Zahari et al, 2020).

## 2.3. ESTADO DEL ARTE DE LENGUAJES INTEGRADOS PARA VIDEOJUEGOS

### 2.3.1. Scripting de lógica

El *scripting* de aquella lógica de más alto nivel que carece de gran complejidad técnica y no tiene grandes necesidades de rendimiento o uso de memoria -estando estas características en el código fuente que la soporta a un nivel más bajo de abstracción-, es uno de los principales objetivos en la búsqueda de abstracción proporcionada por lenguajes de alto nivel, teniendo principalmente tres objetivos en el desarrollo con tecnología de videojuegos:

- ▶ Capacitar la programación del comportamiento del software sin requerir una recompilación del código fuente.
- ▶ Acelerar el proceso de desarrollo a través de eliminar complejidades de bajo nivel y mejorar la legibilidad de las implementaciones.
- ▶ Permitir a perfiles con menos conocimientos de programación participar en el desarrollo de esta lógica.

Diversos motores hacen uso de lenguajes textuales para el *scripting*, ya que ofrecen la flexibilidad necesaria para cumplir con las necesidades de las implementaciones y cumplen con los requisitos mencionados. Algunos desarrolladores de motores han creado sus propios lenguajes con este fin, como UnrealScript para el motor Unreal Engine o GameMaker Language para el motor GameMaker que tienen algunas características similares a GPL, pero contienen implementaciones dedicadas en exclusiva al dominio de desarrollo de videojuegos, mientras que otros hacen uso de GPL preexistentes que son integrados para su ejecución dentro del motor, añadiendo capacidades de interoperabilidad con el código fuente, siendo Lua uno de los lenguajes más utilizados con este fin.

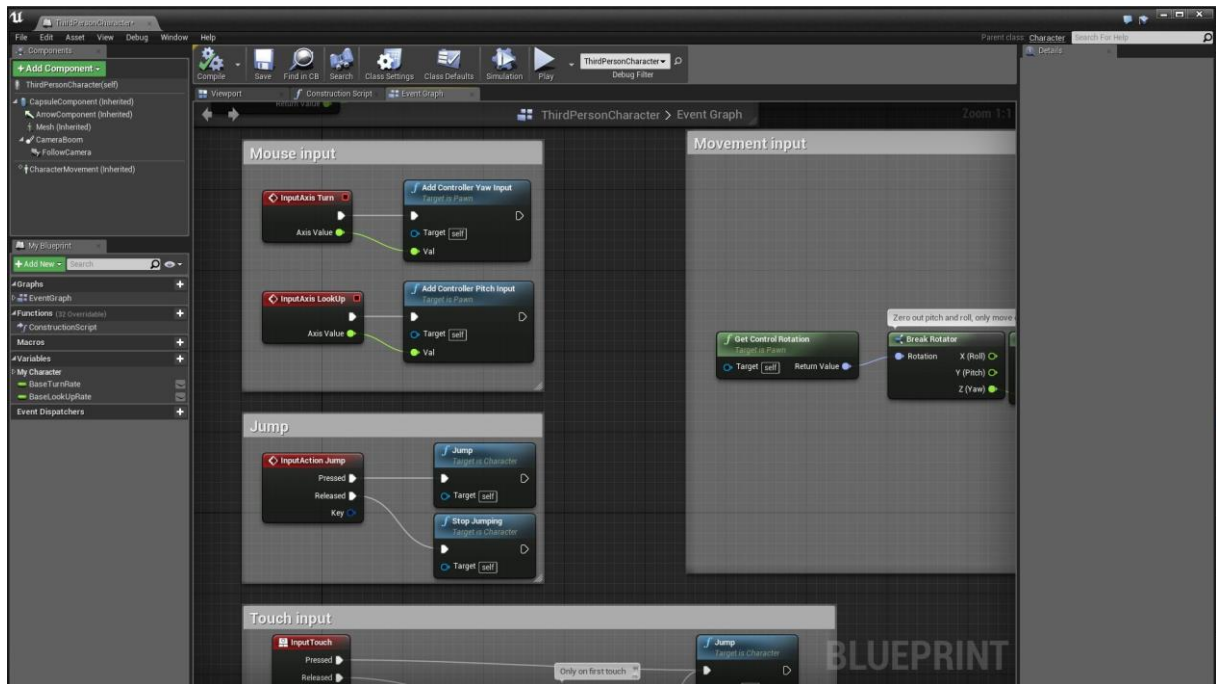
Existen otros ejemplos del uso de DSL también centrados en facilitar el proceso de desarrollo y hacerlo más accesible, aunque su uso está centrado en proporcionar modelos independientes a partir de los que generar el juego, mediante un DSL que, a través de un motor de transformación utilizando plantillas para cada plataforma objetivo, permite obtener el videojuego en código nativo (Nuñez-Valdez et al, 2013; González García et al, 2019).

En algunos casos, se ha considerado que estos lenguajes textuales de *scripting* no satisfacen adecuadamente los objetivos buscados con su implementación, optándose por proporcionar herramientas de programación visual.

UnrealScript es un lenguaje nativo de *scripting* para Unreal Engine, obsoleto en versiones modernas, que fue diseñado para la programación de juegos simple y de alto nivel. Se considera similar a Java, orientado a objetos, aunque es un lenguaje interpretado. El motor Unreal Engine descartó en su versión 4 el lenguaje UnrealScript a favor de su sistema de programación visual basada en nodos denominada Blueprints (UnrealWiki, 2021).

Blueprints es un sistema de programación visual basado en nodos que permite capacidades muy avanzadas de implementación lógica, pudiendo actuar como un sustituto de la programación en C++ durante el desarrollo (Epic Games, 2021a).

**Figura 2.** Sistema de programación visual Blueprints.



Fuente: Epic Games, 2021a.

### 2.3.2. Línea de comandos

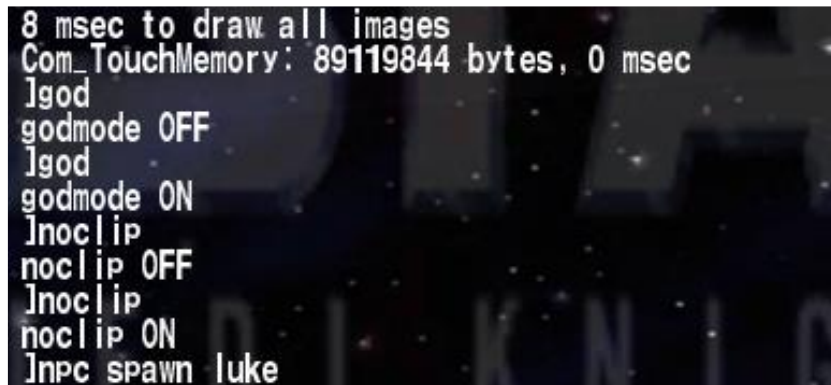
Algunos videojuegos realizan implementaciones que permiten interpretar líneas de comandos textuales, comúnmente a través de una consola dedicada que forma parte de la aplicación construida, en algunos casos accesible para los usuarios finales. Las consolas de línea de comandos actúan esencialmente como un tipo adicional de input para el software, pero sin estar ligado a ninguna limitación física derivada del uso de periféricos, permitiendo una interpretación sin límites de una cadena de caracteres escrita.

De cara a los usuarios finales, han estado tradicionalmente relacionados con dos usos principales: activar “trampas” que afectan al comportamiento del juego de una manera no intencionada por el diseño de juego, y permitir modificaciones adicionales de características del juego que no están presentes en los menús de interfaz de configuración, ya que los desarrolladores estiman que no tienen suficiente utilidad para el jugador medio como para justificar su inclusión en los menús.

En caso de implementar comandos, no obstante, su objetivo principal será el de actuar como una pieza clave del proceso de desarrollo en sí, ya que se utilizarán para los procesos de

pruebas manuales, siendo estos el objetivo de las “trampas” que podrán opcionalmente ponerse a disposición de los jugadores.

**Figura 3.** Ejemplo de comandos activados en la consola de un videojuego.



```
8 msec to draw all images
Com_TouchMemory: 89119844 bytes, 0 msec
lgod
godmode OFF
lgod
godmode ON
lnoclip
noclip OFF
lnoclip
noclip ON
lnpc spawn luke
```

Fuente: Captura del videojuego Star Wars: Jedi Knight II (LucasArts, 2002).

Algunos ejemplos de comandos clásicos son el *godmode*, para hacer al personaje jugador invencible, o *noclip*, para permitir al jugador moverse libremente por todo el mapa jugable sin ser afectado por colisiones, ambos dos ejemplos claros de su utilidad en los procesos de pruebas.

Es común que los comandos puedan interpretar también input de forma literal, ejecutando una función en el código a partir de una variable introducida por texto por el usuario, como un comando *npc spawn x*, siendo *x* el nombre asignado a un personaje-no-jugable.

La principal diferencia entre un lenguaje específico y un sistema de línea de comandos radica principalmente en los siguientes puntos:

- ▶ Los comandos están orientados a ser ejecutados opcionalmente durante la ejecución del juego. Nunca permiten ningún tipo de guardado ni posibilidad de compilación, mientras que los *scripts* son fácilmente usables a través de archivos textuales escritos previamente a su uso y pudiendo compilarlos para una ejecución más óptima. Se considera válido denominar *script* a un archivo que junta comandos para su ejecución secuencial.
- ▶ Los comandos están pensados para ser una ejecución de paradigma funcional, unitaria e independiente, mientras que un script puede depender de un estado dentro de su propia ejecución, otorgándole mucha mayor flexibilidad y expresividad.

- Un comando suele implementarse con grandes limitaciones en su capacidad de procesar el texto, siendo un mapeado directo de texto a función, opcionalmente con la posibilidad de permitir interpretar una variable con la que alimentar a la función (Figura 4).

En la wiki de Valve Software (2021) se muestra un registro de todos los comandos disponibles para su motor de videojuegos, Source Engine. El motor es capaz de interpretar cientos de comandos, desde configuraciones de usuario, jugabilidad, valores de interfaz gráfica o incluso variables de uso para la inteligencia artificial.

**Figura 4.** Ejemplo en código C# de una función intérprete de comandos.

```
Dictionary<string, ConsoleCommand> commands;

public void InterpretCommand(string command)
{
    if(string.IsNullOrEmpty(command))
    {
        EmptyCommand();
        return;
    }

    string[] split = command.Split(' ');

    if(commands.TryGetValue(split[0], out ConsoleCommand command))
    {
        command.Execute(split.Length == 2 ? split[1] : null);
    }
    else
    {
        InvalidCommand(command);
    }
}
```

Fuente: Elaboración propia.

El uso de líneas de comando como método para utilizar software es común en dominios específicos de alta complejidad, y en la literatura se pueden encontrar ejemplos de la limitación que suponen debido a la falta de familiaridad de sus usuarios finales con el uso requerido. Morais et al (2018, p.2) destacan el entrenamiento intensivo requerido para utilizar de manera efectiva los interfaces de líneas de comandos en el ámbito de la bioinformática, debido en gran parte a ser desarrollados para UnixShell, y especifican al respecto que «la

bioinformática ha sido una causa de ‘dolor de cabeza’ para muchos científicos, incluso aquellos que han crecido en la era de las computadoras».

### 2.3.3. Lenguajes de shading

El módulo para el renderizado de gráficos en tiempo real por computador es uno de los aspectos fundamentales que caracterizan a un motor de videojuegos.

Los lenguajes de *shading* son lenguajes de programación de gráficos adaptados para la programación de efectos de *shaders* -pequeños programas encargados del renderizado en GPU-, entre los que se encuentran principalmente GLSL para OpenGL, HLSL para DirectX, MSL para la API Metal, PSSL para PlayStation, así como variantes y derivados, siendo lenguajes con una sintaxis similar a C/C++, y para los que existen herramientas de compilación de un lenguaje a otro, frecuentemente pudiendo reutilizarse un mismo código generado en el desarrollo con un motor de videojuegos para construir las aplicaciones a diferentes plataformas. Estos lenguajes proporcionan abstracciones sobre el ámbito de los gráficos por computador, permitiendo un uso de funcionalidades construidas en el propio lenguaje para favorecer un desarrollo de más alto nivel, pero aun así con un nivel de complejidad elevado debido a las necesidades no sólo de conocimientos sobre el dominio, sino de matemáticas, física y programación de bajo nivel -especialmente en relación al uso de memoria y optimización de rendimiento-, además de los conocimientos concretos relacionados con las características del lenguaje en sí.

En la documentación oficial de la librería de gráficos OpenGL, se observa que GLSL ofrece soporte para características comunes como bucles *for* o condiciones *if*, pero se diferencia de C/C++ en el uso de funciones estándar, variables predefinidas, inclusión de otros tipos para variables relevantes para el renderizado, calificadores de tipos y bloques de interfaz (Khronos Group, 2021).

La programación con lenguajes de *shading* se considera un ámbito con una complejidad técnica muy alta, siendo lenguajes que no están pensados para mejorar la accesibilidad a la escritura de código para computación de gráficos por perfiles menos técnicos, sino de proporcionar un entorno de edición para los programas *shader* y facilitar el trabajo de programadores experimentados.

**Figura 5.** *Función en lenguaje OpenGL.*

```
float GPURnd(inout vec4 state)
{
    const vec4 q = vec4(1225, 1585, 2457, 2098);
    const vec4 r = vec4(1112, 367, 92, 265);
    const vec4 a = vec4(3423, 2646, 1707, 1999);
    const vec4 m = vec4(4194287, 4194277, 4194191, 4194167);

    vec4 beta = floor(state / q);
    vec4 p = a * mod(state, q) - beta * r;
    beta = (sign(-p) + vec4(1)) * vec4(0.5) * m;
    state = (p + beta);

    return fract(dot(state / m, vec4(1, -1, 1, -1)));
}
```

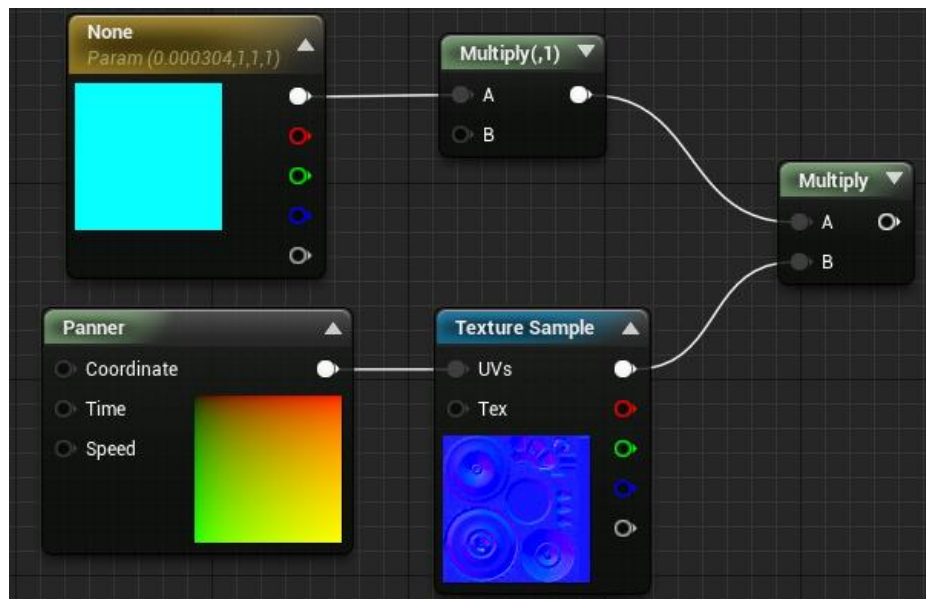
Fuente: Hachisuka, 2015.

Algunos motores integran herramientas de edición visual que son capaces de generar código en uno de los lenguajes de *shading* compatibles a partir de un grafo de nodos con el objetivo para facilitar el proceso de creación de *shaders*, volviendo su creación accesible a usuarios de perfil artístico-técnico que no posean conocimientos de programación avanzados, permitiendo trabajar de manera autónoma en la autoría de materiales para su renderizado. Estas herramientas suponen un coste de desarrollo elevado y debido a esto no forman parte de todos los motores generalistas, aunque algunos motores disponen de *plugins* desarrollados por terceros.

La autoría de *shaders* es no obstante un contexto de edición que se reserva al desarrollo interno y no se proporciona en el software final, utilizándose otras opciones como librerías de materiales e interfaces de edición de los datos que alimentan sus variables cuando se requiere que los usuarios finales puedan modificar visualmente la aplicación, ya que incluso con las abstracciones proporcionadas por las herramientas visuales crear *shaders* sigue implicando conocimientos avanzados del dominio específico de gráficos por computador.



**Figura 6.** Grafo de nodos en el editor visual de materiales de Unreal Engine 4.



Fuente: Epic Games, 2021b.

#### 2.3.4. SQL y declaraciones

Como muchas otras aplicaciones software, algunos videojuegos muestran extensas necesidades de almacenamiento y manejo de datos.

*Structured Query Language* (SQL) es un DSL declarativo para administrar bases de datos relacionales que ha sido usado con frecuencia en videojuegos con características *online* y sistemas de juego que requieren un uso extensivo de datos, como el género MMORPG. El juego World of Warcraft hace uso de bases de datos relacionales para casi todos los sistemas de juego, incluyendo datos de *ítems*<sup>3</sup>, datos de criaturas, datos de personajes jugables, datos de inteligencia artificial, datos para el *spawning*<sup>4</sup>, entre otros (Blizzard Entertainment, 2018). El alcance de SQL incluye peticiones de datos, manipulaciones de datos, definiciones de datos y control de acceso a datos. Contiene múltiples sentencias, siendo esencialmente un lenguaje declarativo con algunos elementos procedurales y su fundación teórica está basada en el álgebra relacional y el cálculo relacional basado en tuplas.

<sup>3</sup> Se denomina ítems a los objetos usables que forman parte del inventario en los juegos RPG.

<sup>4</sup> Se denomina *spawn* al proceso ejecutable de generar contenido, como un personaje-no-jugable, en algún punto del mundo virtual.

El lenguaje SQL proporciona un buen ejemplo de la capacidad de abstracción que es posible obtener a través de lenguajes de dominio específico con características declarativas, siendo posible para usuarios no-técnicos intuir las operaciones realizadas por el código cuando estas son sencillas, aunque su relevancia como una pieza fundamental del desarrollo de software usando bases de datos relacionales implica que regularmente se requiere un uso del lenguaje para sentencias complejas, habiendo evolucionado a través de diferentes estándares que lo sitúan con una complejidad alta y siendo utilizado por desarrolladores experimentados que requieren de conocimientos avanzados de lenguaje.

**Figura 7.** Ejemplo de sentencias SQL.

```
SELECT
    employee.id,
    employee.first_name,
    employee.last_name,
    SUM(DATEDIFF("SECOND", call.start_time, call.end_time)) AS call_duration_sum
FROM call
INNER JOIN employee ON call.employee_id = employee.id
GROUP BY
    employee.id,
    employee.first_name,
    employee.last_name
ORDER BY
    employee.id ASC;
```

Fuente: Emil Drkusic, 2020.

Su evolución también muestra como ha sido capaz de añadir capacidades imperativas a un lenguaje que originalmente era sólo declarativo, como el flujo de control mediante palabras clave como [BEGIN/END](#), [BREAK](#), [CONTINUE](#), [GOTO](#), [RETURN](#), [IF/ELSE](#), [TRY/CATCH](#), [THROW](#), [WAITFOR](#) y [WHILE](#) (Microsoft Corporation, 2017).

La referencia de lenguaje SQL proporcionada por Oracle Corporation (2016) contiene más de 1500 páginas con documentación sobre el lenguaje, incluyendo tipos de datos, reglas de comparación, literales, modelos de formato, objetos de base de datos, sintaxis para objetos de esquema para sentencias, operadores, funciones, expresiones, condiciones, cláusulas de definición de datos, peticiones y subpeticiones, así como información para el uso de diversas sentencias más complejas, poniendo de manifiesto las necesidades de aprendizaje que implica dominar el lenguaje pese a la intuitiva expresividad de las declaraciones.

### 2.3.5. Lenguajes de marcado y estilo

Los DSL de marcado y estilo son algunos de los lenguajes más conocidos y utilizados, como HTML y CSS. El motor de videojuegos Unity incorpora en su paquete para la creación de interfaces gráficas *UI Toolkit* dos lenguajes creados específicamente para adecuarse a las necesidades del motor en la creación de interfaz: UXML (Unity Extensible Markup Language) y USS (Unity Style Sheets).

UXML define la estructura del interfaz de usuario, y se inspira en HTML, XAML y XML. Las características de UXML están definidas en código nativo C#, y funcionalidades como añadir nuevos elementos o atributos disponibles requieren de programación C# por parte de los desarrolladores (Unity Technologies, 2021a).

**Figura 8.** *Ejemplo de UXML.*

```
<?xml version="1.0" encoding="utf-8"?>
<UXML
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="UnityEngine.UIElements"
  xsi:noNamespaceSchemaLocation="../UIElementsSchema/UIElements.xsd"(...)>
  <Label text="Select something to remove from your suitcase:"/>
  <Box>
    <Toggle name="boots" label="Boots" value="false" />
    <Toggle name="helmet" label="Helmet" value="false" />
    <Toggle name="cloak" label="Cloak of invisibility" value="false"/>
  </Box>
  <Box>
    <Button name="cancel" text="Cancel" />
    <Button name="ok" text="OK" />
  </Box>
</UXML>
```

Fuente: Unity Technologies, 2021a.

USS permite configurar estilos que asignar a elementos visuales en interfaz. Está inspirado en el lenguaje CSS, pero contiene personalizaciones para trabajar con Unity (Unity Technologies, 2021b).

**Figura 9.** *Ejemplo de USS.*

```
selector {  
  property1:value;  
  property2:value;  
}  
  
Button {  
  width: 200px;  
}
```

Fuente: Unity Technologies, 2021b.

Estos lenguajes han sido desarrollados de manera posterior y como sustituto a las funcionalidades de creación y edición de interfaces de usuario integradas en el motor siguiendo su marco de entidades y componentes de Unity, que permite la creación de jerarquías anidadas de objetos en la escena y componentes que describen los elementos de interfaz, con sus datos y comportamiento, y proporcionando en la API de Unity el control sobre los mismos mediante código C#. Se puede estimar por tanto que los desarrolladores del motor lo han considerado una opción más efectiva para el desarrollo de interfaces gráficas.

## 2.4. CONCLUSIONES

Los ejemplos existentes de uso de DSL en la literatura a menudo enfocados en considerar como su dominio objetivo el propio desarrollo de videojuegos, pero no en usarse como una parte integrada del desarrollo cuando los objetivos funcionales de una aplicación requieren de manejar otros dominios específicos. El enfoque orientado a lenguajes es de interés, pero su existencia en el ámbito de los videojuegos sufre del mismo punto de vista con respecto al desarrollo como dominio. Los ejemplos existentes de uso de DSL para favorecer el desarrollo de dominios específicos ajenos al propio dominio de desarrollo de videojuegos son una fuente útil para demostrar la utilidad y el potencial del uso de DSL, pero no ofrecen soluciones a la capacidad de desarrollar DSL cumpliendo con requisitos de maximizar la simplicidad en el desarrollo, integración con el propio motor de videojuegos, y el propio proceso de creación de DSL como una parte regular del desarrollo.

La programación orientada a lenguajes ofrece una base conceptual útil, aunque ambigua según el autor que utilice el término, pero por un lado está muy centrada en utilizar DSL como

base para el desarrollo de todo el software, y parece enfocar un uso demasiado formal del desarrollo de lenguajes como para poder garantizar la simplicidad y facilidad de uso buscado, dando como resultado un enfoque de programación que muy probablemente sería ignorado por la gran mayoría de desarrolladores dada la necesidad de iteración, prototipado y las limitaciones de recursos, tiempos y costes en los proyectos.

El desarrollo de DSL es un proceso complejo, incluyendo la necesidad de conocimiento de herramientas externas donde crear los lenguajes, y su posterior integración con un motor de videojuegos, que puede hacerse posible mediante generación de código compatible con el código del motor. Esta complejidad y la falta de integración e interoperabilidad por defecto en el propio desarrollo es una de las principales barreras, ya que no parece factible para intentar reducir recursos de desarrollo y facilitar el prototipado, permitiendo una construcción sencilla, integrada e incremental de lenguajes de alto nivel para configurar y editar las funcionalidades aplicables a un dominio concreto construidas en una aplicación.

La implementación de consolas de comandos puede servir como inspiración para una implementación integrada más elaborada. Su simplicidad para el desarrollo es muy atractiva para cualquier equipo de desarrolladores, ejemplificado por su uso extendido entre la tecnología de videojuegos, pero supone enormes limitaciones para garantizar la expresividad y flexibilidad de funcionamiento a través de las abstracciones proporcionadas que serían deseables en un DSL, incluso aunque se implementara para procesar *scripts* de comandos, en vez de ser un proceso de ejecución de líneas de comando.

Algunos de los ejemplos ponen de manifiesto la dificultad de producir DSL que sean realmente simples y sencillos para los usuarios con perfiles de baja tecnicidad, de ahí que en algunos casos algunos desarrolladores hayan optado por desarrollar herramientas de programación visual como vía alternativa. Algunas soluciones que favorecen la simplicidad de implementación para los programadores no son capaces de proveer un entorno que facilite su uso a los perfiles del dominio específico que harán uso de la implementación, por lo que resulta difícil encontrar una solución que ofrezca facilidad y simplicidad de implementación,

tiempos y coste de desarrollo reducidos, y capacidad de adaptarse a las abstracciones de alto nivel necesarias para favorecer la facilidad de uso para los perfiles en función de su nivel de tecnicidad.

El código de lenguaje SQL se encuentra entre los más legibles gracias a sus características declarativas. El uso de declaraciones para definir el comportamiento deseado parece una de las implementaciones más útiles para garantizar la abstracción del DSL y permitir que la complejidad de implementación sea subyacente. Así mismo, SQL muestra las posibilidades incrementales de un lenguaje, pudiendo añadir características imperativas cuando se vuelve necesario, lo que es sumamente útil para mantener el proceso básico de desarrollo de DSL lo más sencillo posible y garantizar la posibilidad de añadir complejidad de manera aditiva.

## 3. OBJETIVOS CONCRETOS Y METODOLOGÍA DE TRABAJO

### 3.1. Objetivo general

Diseñar e implementar un conjunto software integrado en un motor de videojuegos que permita crear, mantener, editar y ejecutar Lenguajes de Dominio Específico textuales de alto nivel que afecten al comportamiento de la lógica de ejecución de las aplicaciones construidas sin necesitar herramientas externas, sirviendo como fuente de configuración y edición, priorizando la simplicidad y facilidad de uso de todas las funcionalidades proporcionadas.

### 3.2. Objetivos específicos

Desarrollar un paquete software integrado en un motor de videojuegos que permita crear, modificar y ejecutar DSL como parte del código compilado de aplicación a través de *scripts* de cadenas de caracteres utilizando su API<sup>5</sup>.

Iterar para optimizar el proceso de creación y mantenimiento de los lenguajes desde un punto de vista de legibilidad, facilidad de uso e interoperabilidad con el resto de código de aplicación y adecuación a posibilitar la extensión del software fuera del alcance del trabajo con un enfoque de complejidad aditiva para asegurar su ajuste a los requisitos básicos de alto nivel.

Desarrollar un paquete software sobre el anterior que proporcione un entorno de interfaz integrado en el motor, accesible tanto desde su editor como desde cualquier aplicación construida en el mismo, que permita gestionar, consultar y editar los *scripts* para los lenguajes creados.

Desarrollar un paquete software para el motor, independiente de los anteriores, que muestre necesidades de requisitos asociados a los nuevos sectores de aplicabilidad de la tecnología de videojuegos y provea de un entorno donde comprobar las capacidades de integración del software desarrollado sobre implementaciones software de aplicación preexistentes.

---

<sup>5</sup> Interfaz de programación de aplicaciones (Application Programming Interface).

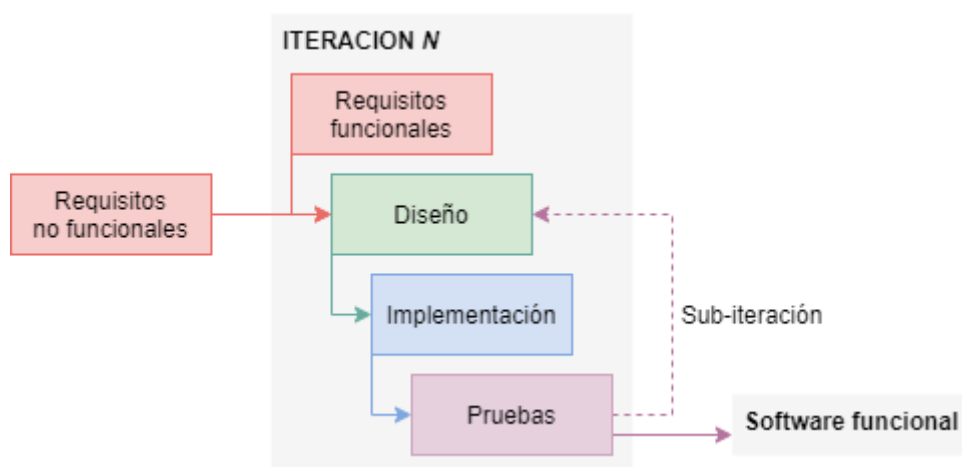
### 3.3. METODOLOGÍA DEL TRABAJO

#### Proceso de desarrollo

El trabajo se realiza de manera iterativa e incremental, con una división en iteraciones a partir de la división del software a desarrollar en conjuntos funcionales cohesivos. Cada iteración incluye fases de especificación de requisitos, diseño, implementación y pruebas. Debido a las necesidades del desarrollo propuesto, la integración se considera como una parte inherente de algunas iteraciones y no como una posible fase. Los requisitos de cada iteración son requisitos de más bajo nivel estipulados a partir de identificar un posible resultado funcional de las necesidades de desarrollo identificadas por los requisitos de alto nivel. Según las necesidades funcionales, cada iteración puede tener dependencia en una o más implementaciones del software en iteraciones anteriores, incluyendo dentro de sí las tareas de revisión, refactorización y extensión del código previo para que sea posible cumplir con sus objetivos funcionales y realizar mejoras observables que permitan su evolución de cara a una mejor adecuación a los requisitos no funcionales.

Debido a la necesidad de experimentar y prototipar para cumplir con los objetivos del trabajo, se considera que las fases de diseño, implementación y pruebas se adecúen a la posibilidad de tener sus propias iteraciones, pudiendo repetirse para mejorar la adecuación al objetivo funcional.

**Figura 10.** Diagrama de iteración en el proceso de desarrollo.



Fuente: Elaboración propia.



El diseño del software se lleva a cabo mediante pseudo programación utilizando el propio entorno de desarrollo. Debido a la falta de necesidad de comunicación durante el proceso de desarrollo con ningún *stakeholder*, no se realizan diagramas o elementos de descripción de diseño similares. El diseño se realiza escribiendo código en el IDE usando el lenguaje de programación final, utilizando las funcionalidades ofrecidas por dicho IDE para navegar y comprender el diseño propuesto, para posteriormente proceder a la extensión de su contenido en la implementación.

Se realizan diagramas como medida de documentación para la comprensión del desarrollo producido.

### **Complejidad funcional aditiva**

A partir de considerar los requisitos más fundamentales de alto nivel del trabajo, se denomina complejidad funcional aditiva al principio guiado por tres reglas que es aplicable a todos los ámbitos del desarrollo orientados al usuario<sup>6</sup>, desde la división de iteraciones al diseño y detalles de implementación del software:

- ▶ La complejidad funcional es inversamente proporcional a la prioridad.
- ▶ Capacitar el añadido de características más complejas de manera limpia, sencilla, cohesiva y desacoplada es más importante que el propio añadido de características.
- ▶ El añadido de mayor complejidad nunca debe romper la adecuación a requisitos de simplicidad, legibilidad y facilidad de uso preexistentes.

### **Requisitos de conocimiento de interfaz**

Debido a que el software desarrollado tiene en gran medida el objetivo de ser utilizado por otros programadores a través de su API, se utiliza como principio para los detalles de implementación la evaluación de los requisitos de conocimiento de interfaz. En este trabajo se denomina requisitos de conocimiento de interfaz de un conjunto software a la necesidad

---

<sup>6</sup> No se considera relevante la complejidad de implementación del software proporcionado, únicamente la complejidad asociada a su uso -que puede incluir implementaciones de terceros usando su interfaz público-.

de conocer información concreta sobre su interfaz público para llevar a cabo una tarea de programación utilizando dicho software.

## 4. DESARROLLO ESPECÍFICO DE LA CONTRIBUCIÓN

### 4.1. IDENTIFICACIÓN DE REQUISITOS

El conjunto software se desarrolla integrado en el motor de videojuegos Unity, en su versión 2020.3.12f, y se implementa al completo en lenguaje C# (versión 8.0, compilador Roslyn, .NET 4.6).

Unity es un versátil motor de videojuegos generalista que permite el desarrollo de videojuegos multiplataforma y que cuenta con implementaciones y una filosofía de diseño que son de gran utilidad para el desarrollo de aplicaciones en otros sectores ajenos al entretenimiento. En una entrevista con Buisness Insider, el propio CEO de Unity Technologies, John Riccitiello, indica el interés de diversas industrias en este motor, y enfatiza el enfoque a largo plazo que tiene la compañía para el desarrollo fuera de los videojuegos, llegando a afirmar que en un plazo de 20 años considera que será un porcentaje mayoritario de sus clientes (Wolverton, 2018).

Unity posee diversas características técnicas que lo hacen ideal como entorno de implementación frente a otros competidores:

- ▶ *Scripting* mediante código C# compilable, que mediante su *scripting backend* il2cpp permite una conversión óptima a C++, actuando como lenguaje por defecto para la programación de código fuente. En comparación con C++, C# permite reducir la verbosidad del código, no requiere de manejo de memoria ni punteros, posee de múltiples utilidades gracias al entorno .NET y tiene funcionalidades construidas por defecto en el lenguaje que permiten reducir las dependencias en librerías de terceros o en el código proporcionado por el motor. El uso de C# compilado también permite dar soporte fácilmente a múltiples librerías de código C# que simplemente pueden añadirse y ser compiladas sin ningún proceso adicional para ser usadas por el código de aplicación.
- ▶ Posee un sistema de paquetes enfocado en modularizar el desarrollo incluyendo manejo de dependencias, un proceso que en otros motores puede ser engorroso con la inclusión de plugins.
- ▶ Tiene soporte para todas las principales plataformas, incluyendo dispositivos de realidad virtual y realidad aumentada, así como dispositivos holográficos orientados al entorno empresarial como las gafas Hololens 1 y 2.

- ▶ Es uno de los motores generalistas con mayor cuota de mercado, teniendo colaboración estrecha con grandes compañías entre las que destaca Microsoft; junto con el entorno .NET y la fácil implementación de código en lenguaje C# o C++, Unity favorece que la mayoría de las librerías de código orientadas a los motores generalistas estén disponibles en Unity y se priorice su desarrollo por encima de otros motores.

Se utiliza Visual Studio 2019 Community como IDE de programación, dada la integración con Unity de la que dispone, y ser un IDE con capacidades avanzadas para la programación en el lenguaje C#.

### **Requisitos de alto nivel del software**

Para el paquete software para la creación, mantenimiento y ejecución de nuevos DSL integrados se establecen los siguientes requisitos de alto nivel:

- ▶ Permitir la creación de un nuevo lenguaje de dominio específico de alto nivel reduciendo la complejidad del proceso al máximo, sin necesitar conocimientos de desarrollo de lenguajes, lenguajes adicionales al usado por el motor de videojuegos, ni conocimientos de cualquier otra herramienta externa o adicional.
- ▶ Código desarrollado al completo de forma nativa sobre el motor junto con el resto de código de aplicación, proporcionando una API integrada usable desde cualquier parte del código fuente, proporcionando interoperabilidad directa.
- ▶ Priorizar el desarrollo de lenguajes sencillos, declarativos y con un alto nivel de abstracción orientados a la configuración, edición o prototipado de lógica frente a capacidades imperativas avanzadas similares a las encontradas en código de propósito general.
- ▶ Priorizar la complejidad funcional aditiva, pudiendo añadir características de mayor complejidad sin que esto impacte en la simplicidad de uso requerido mínimo para crear un nuevo lenguaje, minimizando la curva de aprendizaje para usar el software por programadores.
- ▶ Poseer capacidades de modularización, que permitan extender un lenguaje según qué paquetes de software se han decidido incluir.
- ▶ Entorno extremadamente flexible para permitir que los lenguajes creados puedan cumplir con las necesidades de los desarrolladores para modificar el comportamiento de las

aplicaciones y posibilitando la comunicación con el código de aplicación a través de un contexto para la ejecución semántica encapsulado en un objeto con estado, donde un programador puedan elegir libremente cómo interactuar con su propio código.

- ▶ Priorizar la legibilidad y la facilidad de uso del código proporcionado frente a rendimiento o memoria, no estando diseñado para utilizarse en entornos de rendimiento crítico.
- ▶ Diseño orientado a poder proporcionar guías y ayudas a la escritura del código para los lenguajes desarrollados, facilitando su uso por perfiles de diversa tecnicidad, pudiendo reducir la complejidad y la necesidad de conocimiento y minimizando la curva de aprendizaje para usar los lenguajes por parte de sus usuarios finales.

Para reducir los requisitos de conocimiento de interfaz, teniendo en cuenta la tecnología y lenguaje utilizados, se siguen las siguientes reglas para la API del software:

- ▶ Minimizar la cantidad de espacios de nombres, clases, funciones o variables que es necesario conocer para llevar a cabo una tarea de programación usando la API.
- ▶ Minimizar la visibilidad de todo el contenido dentro de un alcance a no ser que sea estrictamente necesario, incluyendo la visibilidad protegida en las clases no selladas.
- ▶ Forzar funciones abstractas que permitan comprender el funcionamiento de una clase abstracta y sus necesidades de uso al ser de obligatorio cumplimiento ante herencia siempre que sea posible.
- ▶ Minimizar la cantidad de *assets* que es necesario crear para implementar las funcionalidades principales ofrecidas por el software, incluyendo *scripts C#* o cualquier objeto en el editor del motor.
- ▶ Minimizar la cantidad de clases de las que es obligatorio heredar para implementar las funcionalidades principales ofrecidas por el software.
- ▶ Eliminar la necesidad de especificar genéricos<sup>7</sup> en las implementaciones por herencia a no ser que se considere la solución más legible para la especificación de tipos.

---

<sup>7</sup> En C#, se conoce como genéricos al uso de parámetros de tipo para clases o métodos, que permiten no especificar uno o más tipos hasta que la clase o método es declarado e instanciado por el código, permitiendo reutilizar el código para diferentes tipos de manera automática según su uso.

- ▶ Sellar clases cuando sea necesario para dejar constancia explícita de las consideraciones de diseño en cuanto a la herencia.

El diseño de la implementación será guiado por el propio diseño de la API, siguiendo un diseño *user-first* que permita diseñar software orientado a la simplicidad y facilidad de uso, evitando realizar implementaciones según ideas preconcebidas del dominio de aplicación (desarrollo de lenguajes) que puedan no satisfacer los requisitos propuestos.

El código se implementa siguiendo las convenciones de codificación oficiales de C# documentadas por Microsoft (Microsoft Corporation, 2021).

## 4.2. DESCRIPCIÓN DEL SISTEMA SOFTWARE DESARROLLADO

### 4.2.1. Introducción

El núcleo del conjunto software se basa en añadir la capacidad de crear, mantener y ejecutar DSL a través de un paquete software de *scripts* C# usable por cualquier código de aplicación desarrollada en el motor de videojuegos Unity.

A este paquete software se le ha denominado Acolyte, debido a la relevancia del uso de funciones de primera clase a través de delegados<sup>8</sup> en su funcionamiento, los cuales están contruidos como parte del lenguaje C# y contienen una función *Invoke()* para su ejecución.

La prioridad de Acolyte es proporcionar un entorno para la construcción de lenguajes declarativos con un alto nivel de abstracción, utilizando la complejidad funcional aditiva para proporcionar características más avanzadas, incluyendo lógica imperativa.

En Unity, cualquier archivo con extensión '.cs' que contenga código C#, añadido a la carpeta *Assets*<sup>9</sup> será considerado un script que se compilará, siempre y cuando sea compatible con la versión del lenguaje y .NET disponibles en Unity, y tras su compilación, el contenido C# se considera parte del proyecto y cualquier interfaz público aparecerá disponible para cualquier otro script creado en el mismo proyecto.

Acolyte está estructurado en dos carpetas: Core y Unity; con el objetivo de mantener el código principal de manera cohesiva y desacoplado del entorno de uso, pero proporcionando las implementaciones necesarias para proceder a su uso integrado en el motor Unity.

La carpeta Core contiene código C# sin ningún tipo de dependencia a excepción de la API básica del entorno .NET. La carpeta Unity es dependiente en Core y en la API de Unity, y contiene las implementaciones específicas para el uso de Acolyte en el motor.

---

<sup>8</sup> En C#, un delegado es un tipo que representa referencias a métodos con una lista de parámetros y un tipo de retorno, pudiendo usarse para tratar a los métodos como de primera clase.

<sup>9</sup> Assets es la carpeta raíz por defecto con el contenido de un proyecto en Unity.

#### 4.2.2. Declaraciones, invocaciones y estado

El proceso declarativo de Acolyte se basa en especificar palabras, reconocibles como cadenas de caracteres, a las que se les enlaza una función encapsulada. Al procesar líneas de texto en un script e identificar las expresiones, Acolyte ejecutará cada función.

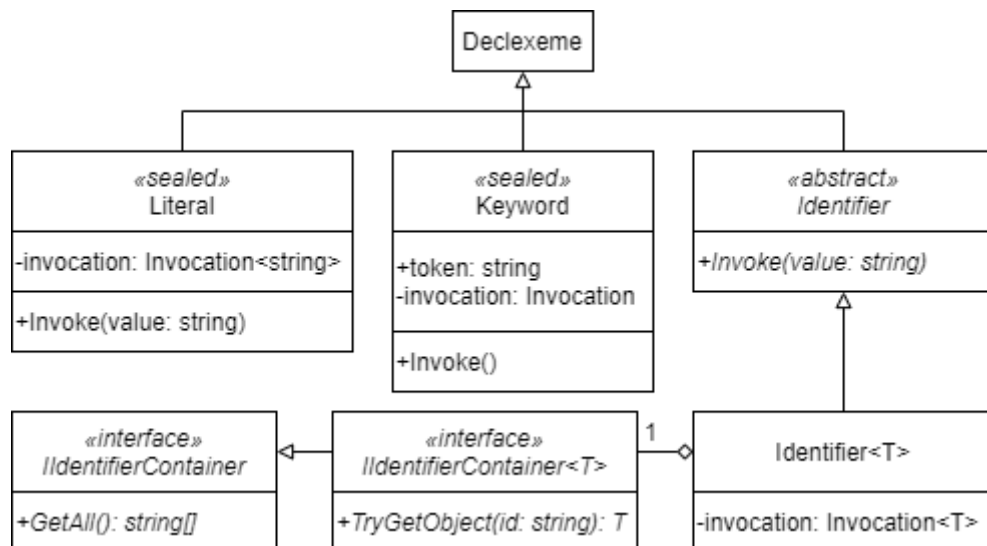
Este funcionamiento declarativo se sostiene en el uso de funciones que contengan todo tipo de complejidad lógica que los desarrolladores necesiten a través de su encapsulamiento en delegados invocables. Acolyte provee de un delegado denominado *Invocation*, incluyendo una opción genérica para un solo parámetro (Figura 11). Si en algún caso fuera necesario el uso de más de un parámetro, se deberán encapsular en una clase o estructura.

**Figura 11.** Delegados de Acolyte.

```
public delegate void Invocation();
public delegate void Invocation<T>(T parameter);
```

Fuente: Elaboración propia.

**Figura 12.** Diagrama de clases para palabras declarativas.



Fuente: Elaboración propia.



Las declaraciones se construyen a partir de una clase *Declexeme* que permite describir una estructura de datos en forma de árbol a partir de enlazar 'decllexemas', denominados Lexemas Declarativos Enlazables (LDE).

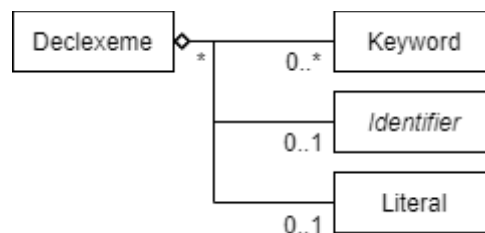
Heredando de esta clase base *Declexeme*, se especifican las clases que describen los diferentes tipos de LDE que es posible utilizar en las declaraciones (Figura 12).

Los tipos de LDE disponibles en Acolyte son:

- ▶ Literales (*Literal*): encapsulan una invocación con parámetro tipo *string*. Se utilizan para permitir el uso de cadenas de caracteres literales como parámetros en la función encapsulada. No se permite herencia de esta clase.
- ▶ Palabras clave (*Keyword*): palabras reconocidas mediante una cadena de caracteres *token* que encapsulan una invocación. No se permite herencia de esta clase.
- ▶ Identificadores (*Identifier*): encapsulan una invocación implementando un genérico. Son similares a un literal, pero su parámetro de invocación será uno o varios objetos de un tipo concreto, que deberán ser identificados por procesos lógicos internos a partir del string proporcionado en el método *Invoke(...)*. Para usar una clase de tipo identificador debe primero definirse por parte del software que va a hacer uso de Acolyte una clase derivada de un interfaz *IdentifierContainer*, el cual permite obtener objetos mediante un string.

Un decllexema agrega las variables que permiten crear la estructura de árbol; se enlaza cada uno con otros LDE subsecuentes de manera tipificada. Como se observa en la figura 13, sólo se pueden especificar un identificador y un literal subsecuentes, pero existen un número indeterminado de palabras clave subsecuentes, ya que estos últimos se reconocen en el texto mediante palabras concretas diferentes entre sí a través de su *token* asignado.

**Figura 13.** Diagrama de estructura de Lexemas Declarativos Enlazables.



Fuente: Elaboración propia.

Debido a la herencia, cada uno de los decllexemas subsecuentes tendrá la misma estructura, creando así un árbol de recorrido unidireccional, que permite representar las posibilidades de declarativas de izquierda a derecha.

Para crear una declaración, la clase *Decllexeme* contiene un método *Then(..)* el cual mediante polimorfismo facilita el añadido tipificado de los LDE subsecuentes para un LDE dado.

**Figura 14.** Código ejemplo de construcción de declaración básica.

```
Printer printer = null;

void SetError() => printer = new ErrorPrinter();
void SetWarning() => printer = new WarningPrinter();
void Print(string message)
{
    printer.message = message;
    printer.Print();
}

var printError = new Keyword("error", SetError);
var printWarning = new Keyword("warning", SetWarning);
var message = new Literal(Print);

printError.Then(message);
printWarning.Then(message);

// Ejemplo: "error Hello-World!"
```

Fuente: Elaboración propia.

En la figura 14 se observa un ejemplo de árbol de declaración básica, en el que tenemos dos posibles palabras clave iniciales, *error* y *warning*, y un literal subsecuente, con el objetivo de hacer un log del literal introducido, y tipificarlo en función de la palabra clave inicial. En un caso tan simple las capacidades de control de estado que se muestran parecen demasiado engorrosas, habiendo creado un objeto con un parámetro de manera aparentemente innecesaria, pero son útiles para permitir abstraer lógica más compleja de manera simple y garantizando una interoperabilidad muy flexible con el resto del código. Se podrían utilizar las palabras claves para construir el lenguaje, pero sin asignar invocación, funcionando por tanto de manera similar a una consola de comandos pero pudiendo estructurar las declaraciones. El

proceso de creación es extremadamente sencillo, simplemente preparando las funciones deseadas y añadiéndolas en la creación de cada palabra. Para preparar el árbol, hacemos uso de la función *Then(...)*, y se puede observar que, dado que cada palabra es una instancia agregada a su padre en el árbol, es posible reutilizar la instancia *message*. A través de la estructura LDE, a diferencia de la identificación de *tokens* por parte de otros lenguajes que permiten grupos desordenados con elementos reconocibles con diferentes reglas, las declaraciones en Acolyte están pensadas para tener un orden de declaración estricto, y el uso de la misma palabra en distintos órdenes tendrá que ser especificado explícitamente mediante el método *Then(...)*. Esta decisión de diseño está relacionada con los requisitos del trabajo, ya que dada la orientación de alto nivel de abstracción para el uso de los lenguajes, es importante poder proveer de guías y ayudas al usuario. Forzar un orden implica que se puede analizar qué palabras deben ir cuándo, pudiendo mostrarse de manera estructurada mientras se escribe el código.

Como se muestra en la figura 15, la herramienta Xtext para la creación de lenguajes proporciona la posibilidad de indicar una secuencia de *tokens* desordenados, que se consideran válidos en función a reglas específicas.

**Figura 15.** Reglas y grupos desordenados en Xtext.

```
Modifier:
static?='static'? & final?='final'? & visibility=Visibility;
enum Visibility
PUBLIC='public' | PRIVATE='private' | PROTECTED='protected';

Ejemplos de grupos:
public static final
static protected
final private static
public
static final static           // ERROR: static appears twice
public static final private  // ERROR: visibility appears twice
final                        // ERROR: visibility is missing
```

Fuente: Eclipse Foundation (2021).

Proporcionar guías para un lenguaje de estas características implica que rápidamente las ayudas a la escritura se llenarían de todo tipo de palabras que es posible utilizar en todo tipo de casos, y no sería posible dejar claro las reglas que les afectan para su correcta escritura.

### Tolerancia de palabras

La clase *Declexeme* (LDE) también incluye una lista de palabras a las que se denomina *tolerated*. Una cadena de caracteres puede añadirse a un declexema mediante el método *Tolerate(string word)*, lo que facilita mayor expresividad cuando se pretenda un uso algo más cercano al lenguaje natural, necesitando algunas expresiones declarativas el uso de palabras como conjunciones o preposiciones. En el procesamiento de una línea declarativa, el análisis sintáctico ignorará una palabra tolerada y pasará a la siguiente.

Por ejemplo, si se requiere de dos posibles declaraciones para indicar una acción sobre un objeto, una colocación o una interacción simple:

*place object*

*interact object = interact **with** object*

**Figura 16.** Código de ejemplo de palabra tolerada.

```
place.Then(selectObject);  
interact.Then(selectObject);  
interact.Tolerate("with");
```

Fuente: Elaboración propia.

#### 4.2.3. Lenguaje, Scripts y Declexicon

Para manejar la creación de un lenguaje se implementan la clase abstracta *Language*, la cual contiene el conjunto de datos que definen a un lenguaje concreto para su uso a lo largo del código. Cuando una aplicación desee utilizar Acolyte para crear un lenguaje, deberá crear una clase que herede de *Language* y definir su contenido.

La clase sellada *Script* define cualquier objeto que contenga dentro de sí un script ejecutable, permitiendo referenciar una cadena de caracteres leída en un archivo de texto a lo largo del

código de manera unívoca. Cada instancia de *Script* requiere de especificar una referencia al *Language* al que pertenece. Tanto *Language* como *Script* hacen uso de una tercera clase: *Declexicon*.

## **Declexicon**

El declexicon es una clase cuyo propósito es definir el contenido declarativo del lenguaje, y actuar como el contenedor del estado de ejecución de las declaraciones. Está compuesto por un objeto *Declexeme* que actúa como raíz del árbol de Lexemas Declarativos Enlazables. Debido a la simplificación propuesta por los LDE, que componen esencialmente un mapeo a una función, el declexicon actúa por tanto como la fuente de especificación de la sintaxis del lenguaje, y como el entorno semántico para el mismo.

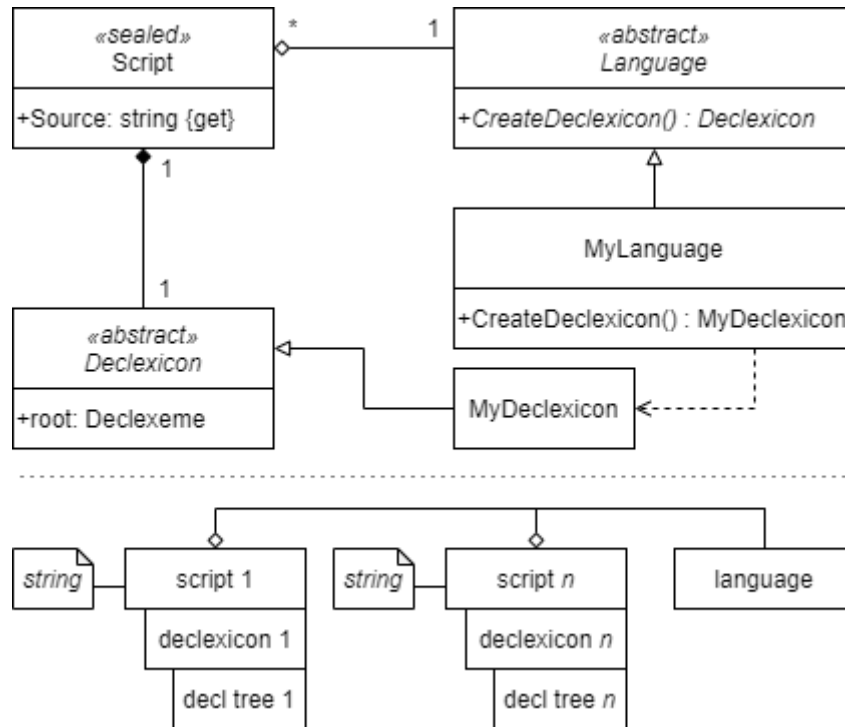
Dada la necesidad de su uso como objeto con estado interno, cada instancia de *Script* contiene su propio declexicon, permitiendo ejecutar correctamente múltiples scripts de manera simultánea para un mismo lenguaje. Como se especifica para los LDE, las funciones están orientadas a ser invocadas en el ámbito de su descripción para gestionar el estado, por lo que cada árbol de palabras debe instanciarse de nuevo, permitiendo instanciar cada función en el alcance adecuado. Esto significa que una clase que derive de *Declexicon* se utilizará como clase para crear la sintáctica del lenguaje y mapear la semántica, al definir su árbol de LDE, y se instanciará para cada *Script* utilizándose como objeto instanciado para controlar esta semántica especificada en el alcance de estado.

En la clase *Language*, se obliga mediante una función abstracta a especificar un método factoría para crear una instancia del declexicon de ese lenguaje, siendo esta factoría utilizada por cada instancia de *Script* en construcción para crear su instancia de declexicon.

Como se observa en la figura 17, una implementación de nuevo lenguaje para Acolyte implica crear un nuevo tipo de declexicon y lenguaje heredando de las clases correspondientes, y utilizar la factoría de *Language* para especificar como crear el tipo de declexicon apropiado. Cada instancia de *Script* es creada a partir de un código fuente en

forma de cadena de caracteres, el cual tiene agregada la instancia del lenguaje que debe utilizarse para interpretarse.

**Figura 17.** Diagrama de clases de *Script*, *Language* y *Declexicon*.



Fuente: Elaboración propia.

La instancia de *Declexicon* contenida por cada instancia de *Script*, será utilizada por otras partes del código de Acolyte para saber cómo procesar declarativamente el código fuente asignado al script.

Para ayudar a su funcionamiento con estado, la clase base *Declexicon* facilita tres métodos, uno que notifica del comienzo de ejecución de un script, y permite pasar como parámetro un objeto que podrá ser convertido al tipo deseado, otro que es llamado cada vez que se procesa un fin de línea en el texto de un script, y otro, con implementación de genérico, que se llama al finalizar la ejecución del script y facilita un delegado para realizar *callback* con el tipo indicado por el llamador (Figura 18).

**Figura 18.** Código de funciones abstractas de Declexicon.

```
public class MyDeclexicon : Declexicon
{
    public override void EndOfLine() {}

    protected override void HandleExecutionStart(object context) {}

    protected override void HandleExecutionCompletion<T>
        (Action<T> callback) {}
}
```

Fuente: Elaboración propia.

La clase base *Declexicon* implementa con anidación privada un interfaz y una clase genérica derivada de este, *IDeclexemeFactory* y *DeclexemeFactory<T>*, que contiene un delegado que permite invocarse con una instancia de declexicon como parámetro y retornar un array de LDE (Figura 19).

**Figura 19.** Clases anidadas en declexicon para su gestión modular.

```
private interface IDeclexemeFactory
{
    bool Invoke(Declexicon declexicon, out Declexeme[] declexemes);
}

private class DeclexemeFactory<T> : IDeclexemeFactory where T : Declexicon
{
    public Func<T, Declexeme[]> factoryMethod;

    public DeclexemeFactory (Func<T, Declexeme[]> factoryMethod)
    {
        this.factoryMethod = factoryMethod;
    }

    public bool Invoke(Declexicon declexicon, out Declexeme[] declexemes)
    {
        if(declexicon is T castedDeclexicon)
        {
            declexemes = factoryMethod.Invoke(castedDeclexicon);
            return true;
        }
    }
}
```

Fuente: Elaboración propia.

Contiene también una lista estática de instancias de esta clase, y un método protegido estático que permite añadir una nueva factoría. Utilizando clases parciales, cualquier código que implemente la clase de forma parcial puede proporcionar un método factoría para el árbol de palabras correspondiente al módulo (Figura 20), mientras se aprovecha la parcialidad de la clase para mantener un único objeto con estado, pero separar las funciones mapeadas para su organización, legibilidad y desacoplamiento modular.

**Figura 20.** Fragmento de código con modularización de Declexicon.

```
public partial abstract class Declexicon
{
    ...
    private static List<IDeclexemeFactory> declFactories;

    public Declexicon()
    {
        foreach(var declexemeFactory in declFactories)
            if(declFactories.Invoke(this, out Declexeme[] declexemes))
                AddDeclexemes(declexemes);
    }
    ...
    protected static void AddDeclFactory<T>(Func<<T>, Word[]> factoryMethod)
        where T : Declexicon
    {
        declFactories.Add(new WordFactory<T>(factoryMethod));
    }
}

-----

public partial class MyDeclexicon : Declexicon
{
    private static void InitializeSomeModule()
    {
        AddDeclFactory((MyDeclexicon declexicon) =>
        {
            return declexicon.GenerateSomeTree();
        });
    }

    private Declexeme[] GenerateSomeTree()
    {
        Keyword example = new Keyword("example", null);
        return new Declexeme[] { example };
    }
}
```

Fuente: Elaboración propia.



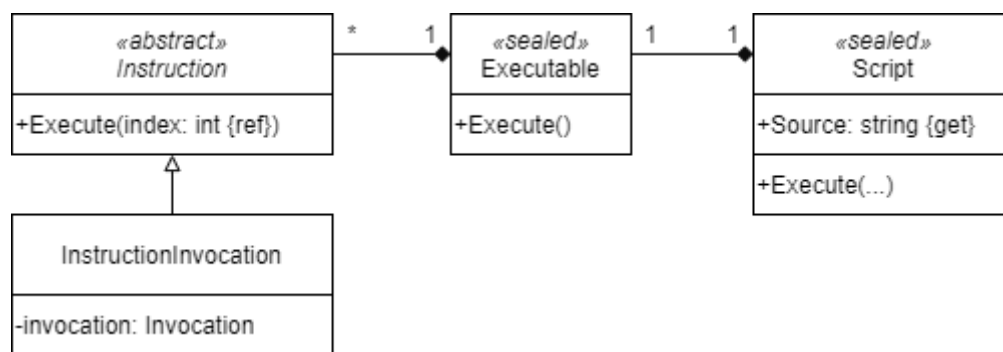
#### 4.2.4. Compilación y ejecución de scripts

Si bien no entra en los requisitos del software priorizar la optimización, se ha optado por realizar un proceso de compilación para los scripts en vez de un interpretador, dando como resultado un conjunto de datos ejecutables que permiten no tener que hacer una interpretación sintáctica del contenido textual de un script por cada ejecución.

Para proceder a la ejecución, se ha creado una clase abstracta *Instruction* que simula la unidad básica de ejecución. La instrucción principal creada para Acolyte es *InstructionInvocation*, que simplemente encapsula un delegado *Invocation* para invocar su función al ejecutarse. Otras implementaciones de instrucción pueden ser usadas para comportamientos más complejos. Se hace uso del parámetro *int* por referencia en la función *Execute(ref int index)* de *Instruction* para permitir a una instrucción realizar acciones como saltos en la ejecución, para la implementación de características imperativas como bucles o condicionales.

Una clase sellada *Executable* encapsula un array de instrucciones, pudiendo ser ejecutadas en orden en un bucle *for* con una variable *int* que se envía al método *Execute(...)*, permitiéndole así a una instrucción controlar el flujo.

**Figura 21.** Diagrama de clases de ejecución de script.



Fuente: Elaboración propia.

Un array de instrucciones, a través de un ejecutable, es lo que genera la clase *Compiler*. Esta clase está referenciada únicamente por la clase *Script* a lo largo de todo el código, por lo que se ha construido como una clase privada anidada dentro de la clase *Script* para mantener la API de Acolyte menos verbosa. La clase *Compiler* se ocupa de procesar sintácticamente el contenido de un *Script* y producir las instrucciones requeridas en el orden adecuado según

su interpretación de cada línea de texto. Realiza su compilación a partir de conocer el lenguaje, el declexicon y el código fuente. Para las características declarativas, hace un fuerte uso de consultar el declexicon para comprobar las palabras existentes en el árbol de LDE, y es el compilador quien añade también instrucciones

**Figura 22.** Código de ejecución dentro de la clase *Script*.

```
public partial class Script
{
    ...
    public void Execute<T>(object context, Action<T> callback)
    {
        if(!IsCompiled)
            Compile();

        declexicon.StartExecution(context);
        executable.Execute();
        declexicon.CompleteExecution(callback);
    }

    private void Compile()
    {
        executable = Compiler.Compile(language, declexicon, Source);
    }
}

-----
script.Execute(new MyCustomContext(someParameters), (MyResult[] results) =>
{
    // ...
});
```

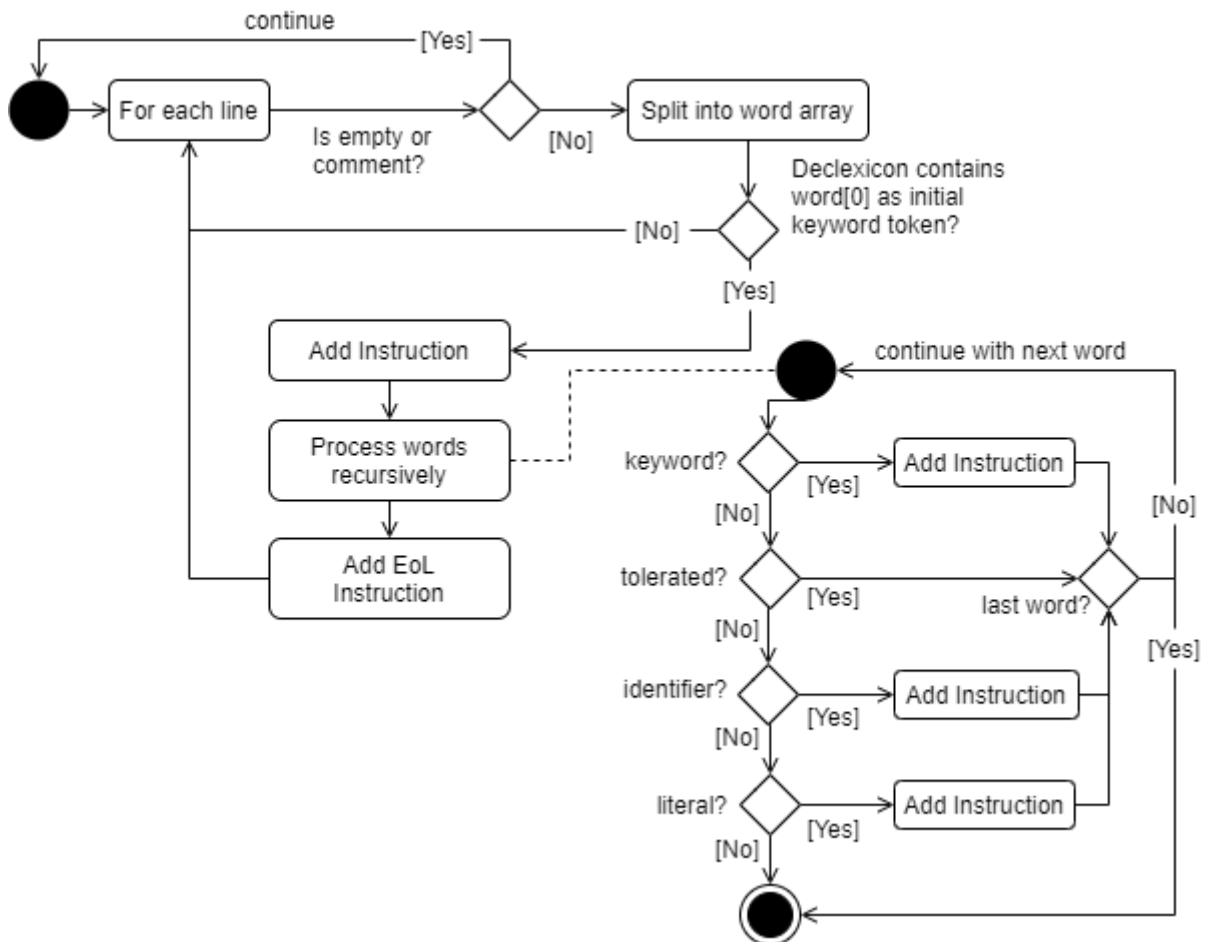
Fuente: Elaboración propia.

La figura 22 muestra un fragmento de código de la clase *Script*, que contiene el método público para su ejecución, permitiendo que el uso de la API de Acolyte sea sencilla desde el exterior. Tan sólo hay que tener una referencia al script deseado e invocar este método *Execute(...)*, proporcionando los parámetros adecuados teniendo en cuenta el manejo de tipos que hemos especificado en nuestro declexicon. Cada script se compila automáticamente cuando se ejecuta si no se ha compilado previamente, aunque puede también compilarse opcionalmente en su construcción.

El procesamiento de declaraciones realizado por el compilador se ocupa de analizar cada línea del código fuente de un script y comprobar su adecuación sintáctica al contenido del

declexicon. Cuando se encuentran correspondencias, se obtiene la invocación encapsulada por la palabra y se añade una instrucción a su lista de instrucciones, que será retornada al completar la compilación en forma de ejecutable.

**Figura 23.** Diagrama de actividad del proceso de compilación.



Fuente: Elaboración propia.

La figura 23 muestra el proceso realizado por el compilador para producir instrucciones a partir de las declaraciones. Procesa línea a línea el código del *script* proporcionado y se hace una partición de las palabras en un array utilizando el separador del lenguaje (por defecto un espacio). Para considerar una línea como una posible declaración, en Acolyte es obligatorio que la primera palabra sea una palabra clave. Si se cumple esta regla, se procesarán todos los *strings* del array comprobando su coincidencia con las posibles palabras subsecuentes a el último LDE procesada a través de una función recursiva.

Esta comprobación se realiza con el siguiente orden de prioridad:

1. Palabra clave - ¿Coincide con alguna de las palabras clave subsecuentes del LDE?
2. Palabra tolerada - ¿Es el *string* tolerado por el LDE?
3. Identificador - ¿Tiene el LDE un identificador subsecuente? Usar el *string* como valor.
4. Literal - ¿Tiene el LDE un literal subsecuente? Usar el *string* como valor.
5. No se encuentra coincidencia. Se procesará la declaración hasta este punto.

**Figura 24.** Fragmento de código con proceso de compilado de tolerancia y literales.

```
private void ProcessRecursively(Declexeme current, string[] words, int i)
{
    ...
    else if(current.IsTolerated(words[i]))
    {
        next = current;
    }
    else if(current.SubsequentLiteral != null)
    {
        AddLiteral(current.SubsequentLiteral, words[i]);
        next = current.SubsequentLiteral;
    }
    ...
    ProcessWordRecursively(next, words, ++i);
}

private void AddLiteral(Literal literal, string value)
{
    AddInstruction(() => { literal.Invoke(value); });
}

private void AddInstruction(Invocation invocation)
{
    instructions.Add(new InstructionInvocation(invocation));
}
```

Fuente: Elaboración propia.

El método recursivo del compilador contiene un *if..else* para manejar las prioridades una por una. Cuando se cumple una condición en función de las prioridades, simplemente encapsula la invocación del *Invocation* contenido en el LDE y lo añade a la lista de instrucciones creando una nueva instrucción de tipo invocación (Figura 24).

Se puede observar una diferencia estructural fundamental entre dos tipos de LDE:

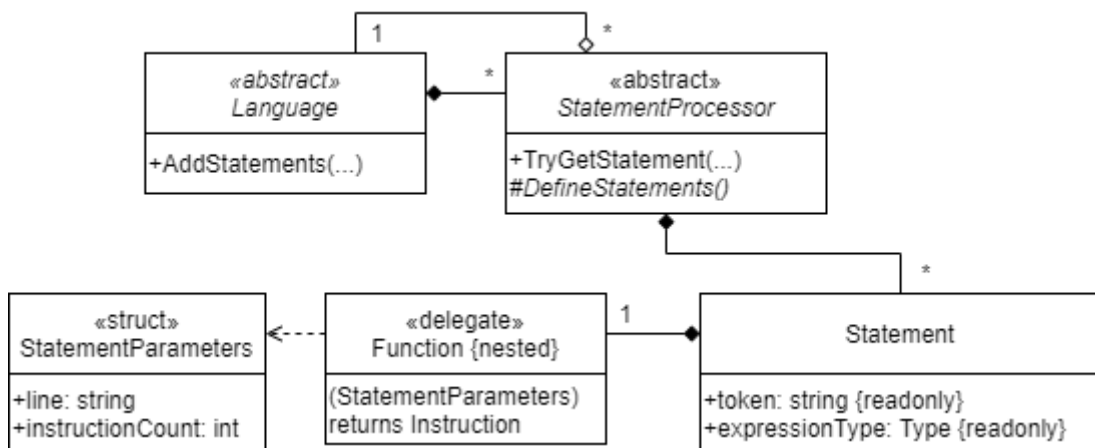
- ▶ Las palabras clave y toleradas consultan si el valor del *string* actual se encuentra entre los registrados en el LDE para determinar la condición en el proceso. Ninguno requiere de parámetros.
- ▶ Los identificadores y literales comprueban si el LDE los soporta como subsecuentes para determinar la condición en el proceso, usando una comparación con *null*, y utilizan el valor del *string* para el parámetro de su invocación.

#### 4.2.5. Complejidad aditiva: *Statements*

Para proporcionar mayores capacidades a Acolyte ajenas al procesamiento de líneas declarativas, Acolyte posee la capacidad de procesar sentencias para manejar lógica imperativa a través de la clase *Statement*.

Las sentencias permiten procesar una línea de código de manera personalizada, y son tratadas con mayor prioridad que las palabras clave de las declaraciones durante el análisis sintáctico. Para implementar sentencias, Acolyte provee de la clase abstracta *StatementProcessor* que genera y gestiona como objeto con estado uno o varios *Statement*.

**Figura 25.** Diagrama de clases de sentencias *Statement*.



Fuente: Elaboración propia.

Cualquier sentencia creada se podrá añadir a un lenguaje, permitiendo al resto del código conocer qué procesos de sentencias existen para gestionar el análisis sintáctico.

Un *Statement* contiene su token identificador y la función que realiza el proceso de análisis de una línea de código y su lógica asociada, retornando una *Instruction* resultante, que puede ser *null* si una sentencia no la produce (Figura 26).

**Figura 26.** Fragmento de código con ejemplo de *Statement*.

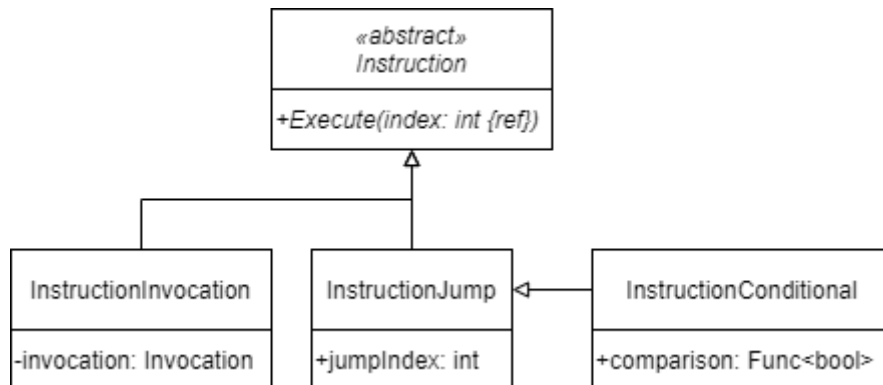
```
public class ExampleStatementProcessor : StatementProcessor
{
    protected override Statement[] DefineStatements()
    {
        return new Statement[]
        {
            new Statement("example", ExampleFunction)
        };
    }

    private Instruction ExampleFunction(StatementParameters parameters)
    {
        return null;
    }
}
```

Fuente: Elaboración propia.

Para facilitar el funcionamiento de *Statements*, Acolyte hace uso de instrucciones adicionales a la instrucción de invocación utilizada en declaraciones, como *InstructionJump* y su generalización *InstructionConditional* (Figura 27).

**Figura 27.** Diagrama de clases de instrucciones.



Fuente: Elaboración propia.

Estas instrucciones permiten cambiar el índice de la ejecución actual de instrucciones, que es pasado como parámetro referencia cuando se ejecutan, para realizar un salto, siendo el salto de *InstructionConditional* determinado por el resultado de un delegado que retorna *bool*.

El compilador añade ahora un análisis previo al proceso declarativo que, para cada línea, consulta uno a uno los *StatementProcessor* del lenguaje del script compilado e intenta procesar el contenido de la línea. En caso de tener éxito, añadirá la instrucción, retornada por la función del *Statement* encontrado, si esta existe, e ignorará el subsecuente proceso declarativo (Figura 28).

**Figura 28.** Fragmento de código con compilación de *Statements*.

```
private bool ProcessStatements(string line)
{
    if(statementProcessors == null) return false;

    foreach(var statementProcessor in statementProcessors)
    {
        if(statementProcessor.TryGetStatement(line, out var statement))
        {
            Instruction instruction = statement.function.Invoke
            (
                new StatementParameters(line, instructions.Count)
            );

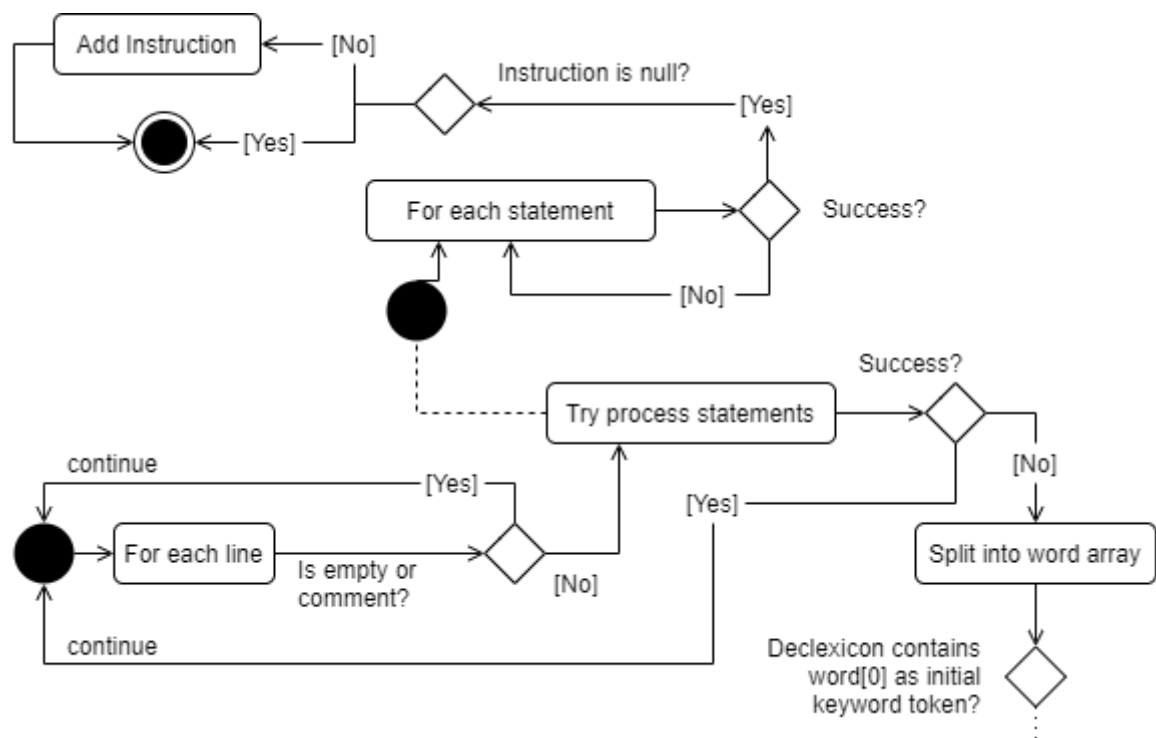
            if(instruction != null)
                instructions.Add(instruction);
            return true;
        }
    }
    return false;
}
```

Fuente: Elaboración propia.

Esta comprobación se realiza usando el token de cada *Statement* definido utilizando el método de *string.StartsWith(...)* en la línea proporcionada.

Esta priorización en la compilación por las sentencias imperativas implica que si se utiliza la misma cadena de caracteres para una palabra clave inicial declarativa que para un token de *Statement*, la declaración nunca será procesada.

**Figura 29.** Diagrama de actividad de compilación de Statements.



Fuente: Elaboración propia.

La figura 29 muestra el cambio en el diagrama de actividad de compilación, donde se añade una nueva actividad de procesamiento de sentencias imperativas antes de la separación de las palabras y su proceso declarativo.

## IF/ELSE

Para comprobar y demostrar el uso de sentencias imperativas, Acolyte implementa una generalización de *StatementProcessor* para sentencias de tipo IF/ELSE a través de la clase *IfElse*. Esta clase define tres *Statement* con sus tres funciones respectivas y hace uso de una pila de instrucciones (no confundir con la lista de instrucciones compiladas) para manejar el estado según las llamadas a las funciones.

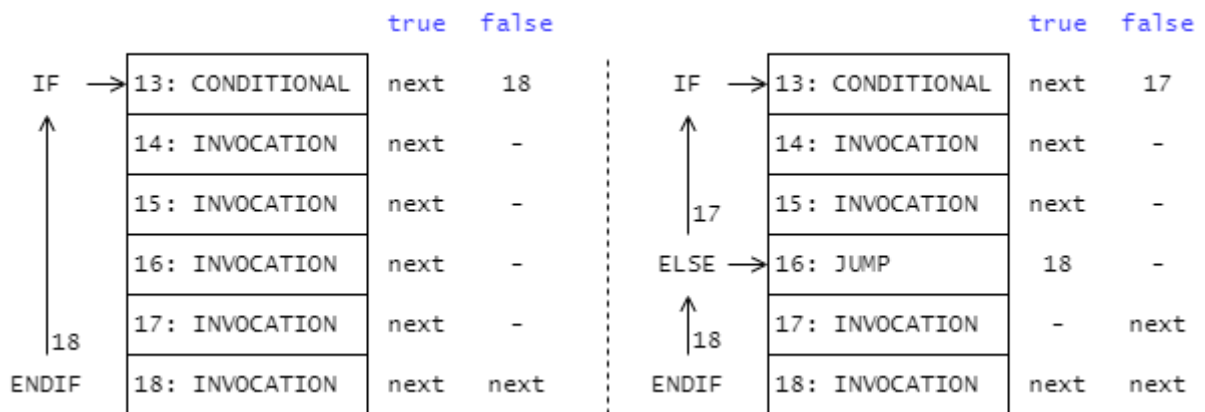
- ▶ La función IF crea una *InstructionConditional* definiendo su delegado de condición, la añade a la pila del procesador y la retorna para añadirse a la lista de compilación.



- ▶ La función ELSE retira la última instrucción de la pila del procesador, le añade el conteo de instrucciones compiladas actuales como índice de salto, y crea una *InstructionJump* que añade a la pila y retorna.
- ▶ La función ENDIF obtiene la última instrucción de la pila del procesador y le añade el índice de instrucciones compiladas actual (conteo - 1) como índice de salto.

Esto permite que el compilador añada instrucciones de salto que, a partir de la interpretación del delegado *bool* de la instrucción condicional creada por el primer IF, puedan realizar saltos de instrucción que procesen u omitan las instrucciones intermedias (Figura 30).

**Figura 30.** Ejemplo de flujo de instrucciones en sentencia IF/ELSE.



Fuente: Elaboración propia.

El añadido de la capacidad de procesar sentencias IF/ELSE a un nuevo lenguaje es extremadamente simple, tan sólo se debe utilizar la función *AddStatements(...)* en la construcción de nuestra clase heredera de *Language* e indicar la instancia de sentencia con nuestros parámetros deseados de *token* (Figura 31).

**Figura 31.** Código para el añadido de Statement.

```
AddStatements(new IfElse("if", "else", "endif"));
```

Fuente: Elaboración propia.

La figura 32 muestra un fragmento de código de la clase *IfElse* donde se observa la función IF que produce la instrucción de salto condicional y la añade a la pila local. Para la construcción del delegado *bool* que necesita la instrucción condicional, se encapsula una consulta utilizando el resto del *string* de la línea analizada con otra de las implementaciones de complejidad aditiva en Acolyte: las expresiones.

**Figura 32.** Fragmento de código con función IF de sentencia *IfElse*.

```
protected override Statement[] DefineStatements()
{
    return new Statement[]
    {
        new Statement(ifToken, IfFunction, typeof(bool)),
        ...
    };
}

private Instruction IfFunction(StatementParameters parameters)
{
    string expression = parameters.line.Substring(ifToken.Length).Trim();
    var conditional = new InstructionConditional
    {
        comparison = () =>
        {
            Language.TryGetExpression(expression, out bool value);
            return !value; // Jump when comparison is false
        }
    };
    instructions.Push(conditional);
    return conditional;
}
```

Fuente: Elaboración propia.

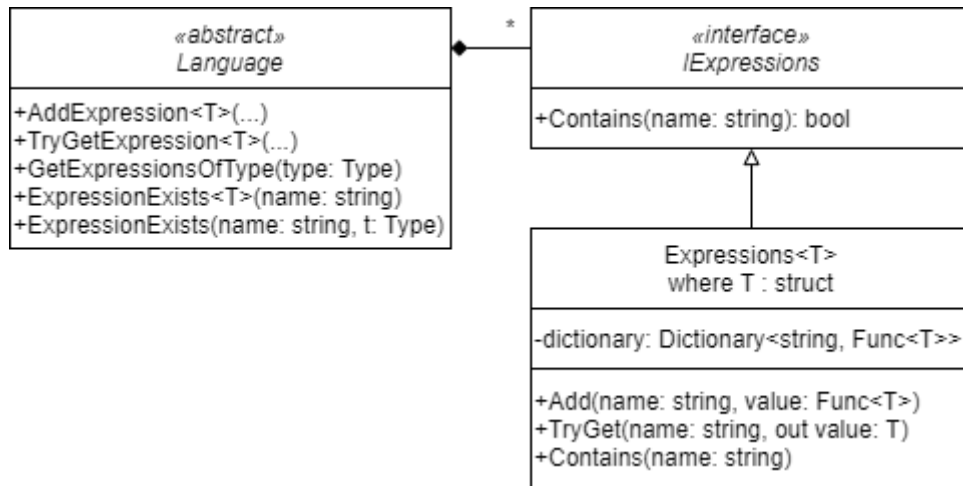
#### 4.2.6. Complejidad aditiva: *Expressions*

Acolyte permite interpretar cadenas de caracteres como expresiones que resuelven a una variable de un tipo primitivo concreto a través de la clase genérica *Expressions<T>*, que implementa el interfaz *IExpressions* para facilitar el uso de la clase.

Una instancia de *Expressions* está formada por un diccionario que mapea un *string* a una función que retorna el valor deseado. Se utiliza una función para poder realizar las operaciones lógicas necesarias para determinar la correctitud del valor según las necesidades de lógica de implementación. Las *Expressions* son añadidas a un lenguaje, que las añade a

diccionarios divididos por tipo y permite la consulta de los valores de cada expresión en cada *Expressions* del diccionario.

**Figura 33.** Diagrama de clases de *Expressions*.



Fuente: Elaboración propia.

Para añadir expresiones se utiliza la función *AddExpression(...)* de la clase *Language*, y la obtención de su valor se consulta con *TryGetExpression<T>(...)* (Figura 34).

**Figura 34.** Fragmento de código con uso de *Expressions*.

```
public MyLanguage()
{
    AddExpression("profile is expert", IsProfileExpert);
}

private bool IsProfileExpert() => return User.Current.Profile.isExpert;
-----
language.TryGetExpression("profile is expert", out bool value);
```

Fuente: Elaboración propia.

Las sentencias imperativas IF/ELSE con las expresiones permiten a un lenguaje definir una condición escribiendo *{if token} {expression string}*.

#### 4.2.7. Implementación para Unity

Si bien el núcleo de Acolyte se encuentra desacoplado y funciona de manera independiente, su objetivo es ser utilizado de manera integrada con el motor Unity. Gracias a estar programado en C#, la integración interoperable con el código de aplicación viene dada por defecto, tras compilar el código. La carpeta Unity del paquete software Acolyte provee de algunos objetos de ayuda para su correcto uso en el entorno de Unity.

##### **ScriptAsset**

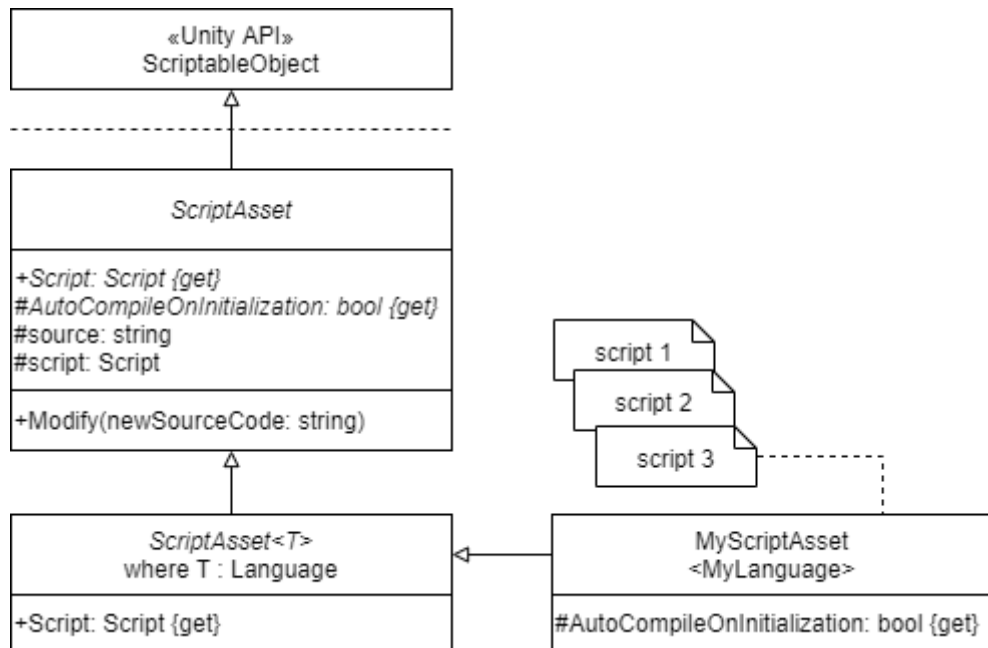
En un motor de videojuegos, se suele denominar *asset* a cualquier conjunto de datos, en forma de archivo, que forma parte del contenido de la aplicación. Los scripts son fundamentalmente cadenas de caracteres cuyo contenido representa su código fuente, pero para poder manejarlos con la mayor comodidad posible en el motor Unity, Acolyte dispone de una clase abstracta *ScriptAsset* que permite manejar un *string* encapsulado como un asset.

*ScriptAsset* hereda de la clase de Unity *ScriptableObject*, la cual es utilizada para poder tratar como assets a objetos en código que deben usarse como conjuntos de datos, a diferencia de otros objetos que podrán estar en la escena 3D (Unity Technologies, 2021c), y se utilizan dos clases *ScriptAsset* para definir los scripts en Acolyte como assets. Se utiliza la clase no-genérica para permitir su uso como interfaz para cualquier otra clase, ya que no se pueden tener referencias a variables de tipos genéricos sin especificar el tipo, y la clase genérica permite especificar el tipo de lenguaje de una implementación de *ScriptAsset* explícitamente al declarar la clase (Figura 35). A través de conocer este lenguaje, la clase genérica realiza un *override* al método de obtención del script que permite crearlo con inicialización *lazy*<sup>10</sup>, especificando automáticamente el lenguaje de dicho script, el cual es requerido por su constructor. De esta manera, cuando cualquier parte del código haga referencia a un *ScriptAsset* y pida su *Script*, se generará automáticamente con los datos correctos si no se ha creado previamente (Figura 36).

---

<sup>10</sup> La inicialización lazy se refiere a esperar al primer uso de un objeto antes de su creación.

**Figura 35.** Diagrama de clases para el manejo de scripts como assets.



Fuente: Elaboración propia.

**Figura 36.** Código de generación de script de ScriptAsset.

```
public abstract class ScriptAsset<T> : ScriptAsset where T : Language, new()
{
    public override Script Script
    {
        get
        {
            if(script == null)
            {
                if(!languages.TryGetValue(typeof(T), out Language lang))
                {
                    lang = new T();
                    languages.Add(typeof(T), lang);
                }
                script = new Script
                    (source, lang, AutoCompileOnInitialization);
            }
            return script;
        }
    }

    private static readonly Dictionary<Type, Language> languages;
```

Fuente: Elaboración propia.

En la figura 37 se implementa un ejemplo de *ScriptAsset* utilizando el atributo *CreateAssetMenu* de Unity, el cual permite añadir la creación de un asset al menú contextual del editor.

**Figura 37.** Código de generalización de *ScriptAsset*.

```
[CreateAssetMenu(fileName = "MyScript", menuName = "TFM/My Script Asset")]  
public sealed class MyScriptAsset : ScriptAsset<MyLanguage>  
{  
    protected override bool AutoCompileOnInitialization => true;  
}
```

Fuente: Elaboración propia.

## Identificador de objetos

Los declexemas de tipo identificador necesitan de una implementación concreta del interfaz *IdentifierContainer* como contenedor de objetos identificados.

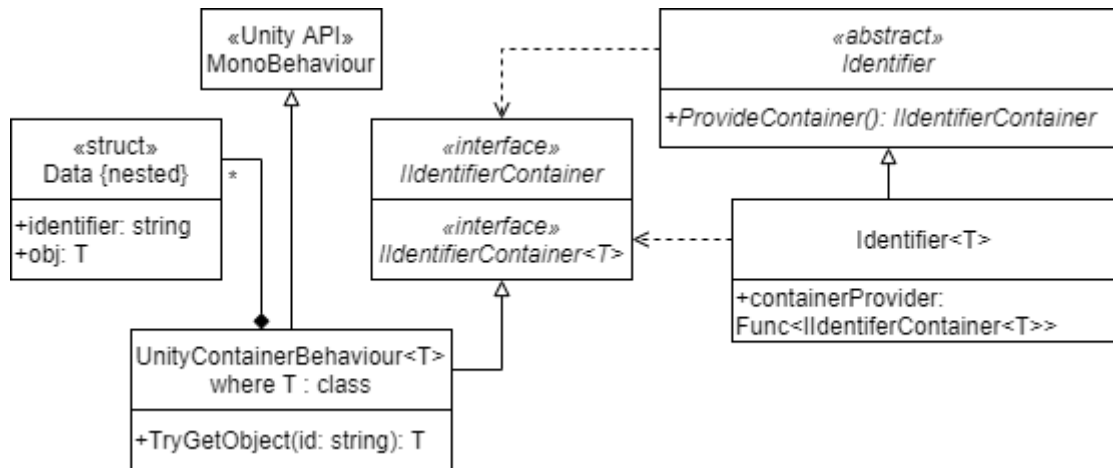
Unity está construido como un *framework* de entidades y componentes. En una escena hay una jerarquía de objetos, denominados *GameObjects*, a los que es posible añadir múltiples componentes que determinan su comportamiento (Unity Technologies, 2021d), como el componente de transformación *Transform*, que facilita sus datos de posición, rotación y escala en escena, componentes de renderizado, componentes de física, o componentes creados por los desarrolladores a través de crear clases que hereden de la clase de Unity *MonoBehaviour*.

El funcionamiento en Acolyte se basa en utilizar un contenedor usable en escena como componente, *UnityContainerBehaviour<T>*<sup>11</sup>, que permite serializar objetos e identificarlos, incluyendo objetos como referencias a *GameObjects* o cualquier componente de Unity.

---

<sup>11</sup> Unity no permite añadir a escena componentes con genéricos, por lo que es necesario crear una nueva clase con cada tipo deseado, como *MyContainer : UnityContainerBehaviour<GameObject>* si se desea serializar en escena. Si la serialización se va a realizar por código, se puede utilizar el genérico.

**Figura 38.** Diagrama de clases de identificación en Unity.



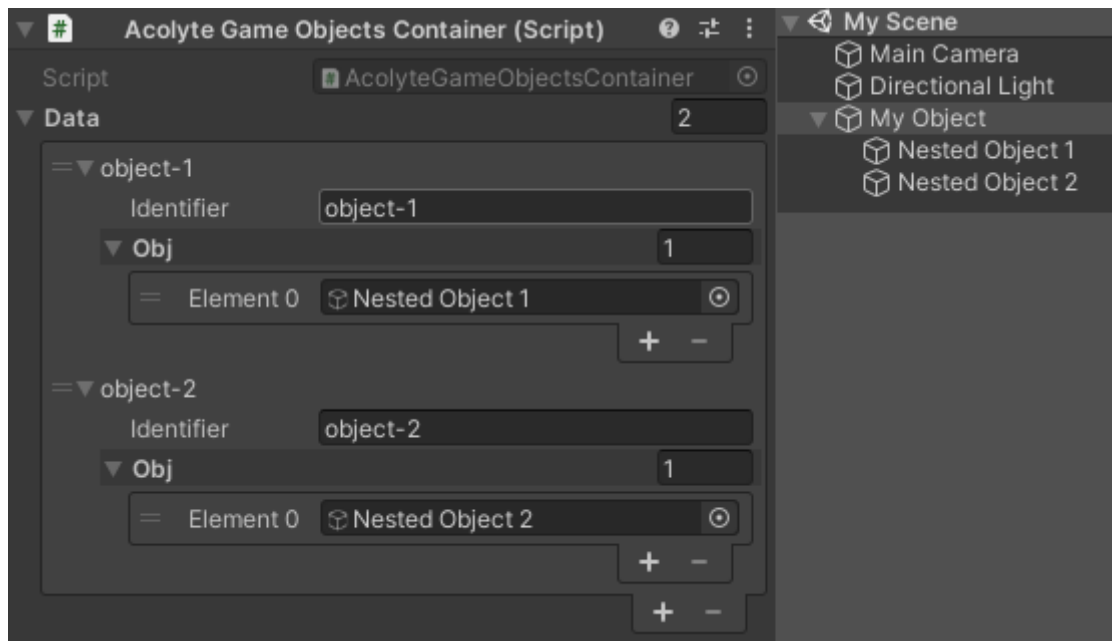
Fuente: Elaboración propia.

Cuando se crea un declexema de tipo *Identifier*, se usa el genérico para especificar el tipo que desea buscar (debe ser una clase referenciable, no se considera válido a los primitivos), y se le proporciona de un delegado del que obtener el contenedor adecuado. Cuando se invoca, utiliza el valor *string* de invocación para buscar en el contenedor un objeto identificado con dicho valor, e invocar su *Invocation* enviando ese objeto como parámetro. Se debe utilizar un delegado para poder asegurar que se puede obtener siempre la referencia al contenedor adecuada a través de usar una función, ya que si se utiliza una referencia directa, esta se asignará al identificador durante la compilación y puede no ser la adecuada durante su ejecución (o incluso no existir todavía).

En la figura 39 se observa el editor de Unity con un componente derivado de *UnityContainerBehaviour* aplicando un array de *GameObject* como parámetro genérico.

En la figura 40 un ejemplo en código muestra cómo se puede especificar un árbol declarativo que interprete una línea *"clone object-1"* o *"destroy object-2"* para clonar o destruir los objetos identificados.

**Figura 39.** Imagen de contenedor de objetos en escena de Unity.



Fuente: Elaboración propia.

**Figura 40.** Fragmento de código con el uso de `UnityIdentifier`.

```
public class Identifier<T> : Identifier where T : class
{
    private readonly Func<IIdentifierContainer<T>> containerProvider;
    ...
    public override void Invoke(string value)
    {
        if(containerProvider.Invoke().TryGetObject(value, out T obj))
            invocation.Invoke(obj);
        else
            invocation.Invoke(null);
    }
    ...
}

var setCloning = new Keyword("clone", SetCloning);
var setDestruction = new Keyword("destroy", SetDestruction);
var selectObj = new Identifier<GameObject[]>(SelectObjects,
() => { return gameObjectsContainer; });

setCloning.Then(selectObj);
setDestruction.Then(selectObj);
```

Fuente: Elaboración propia.



Para que un declexicon tenga las referencias adecuadas a objetos como el contenedor, se debe recordar que es posible utilizar el parámetro *object context* del método *ExecutionStart()*.

#### 4.2.8. Editor de texto integrado en Unity

Aunque los assets de tipo *ScriptAsset* pueden modificarse desde el editor de Unity, es fundamental permitir la edición de scripts desde la aplicación construida. Los requisitos de Acolyte implican que la simplicidad y facilidad de uso no están sólo orientados al desarrollo de DSL, sino a su uso final también.

Acolyte incluye un paquete software que permite la edición de scripts durante la ejecución de una aplicación en Unity, permitiendo abrir cualquiera de los scripts existentes y modificar su contenido contando con ayudas visuales mediante coloración de tipos y guías contextuales del lenguaje del script.

El editor de Acolyte para Unity se ha desarrollado utilizando la implementación de interfaces gráficas de objetos y componentes de Unity. Para el renderizado de texto, se utiliza el paquete *TextMeshPro* que forma parte del motor (Unity Technologies, 2021e).

#### Datos de renderizado

Para poder realizar el renderizado del texto, Acolyte provee de una encapsulación de datos *RenderUnit* que constituye una unidad de renderizado de texto y permite renderizar un contenido *string* junto con cualquier otro dato necesario asociado al mismo. La estructura contiene un método estático público que permite obtener un array de dos dimensiones con la distribución horizontal y vertical de unidades de renderizado que permitan construir los componentes de interfaz que muestran el texto del script.

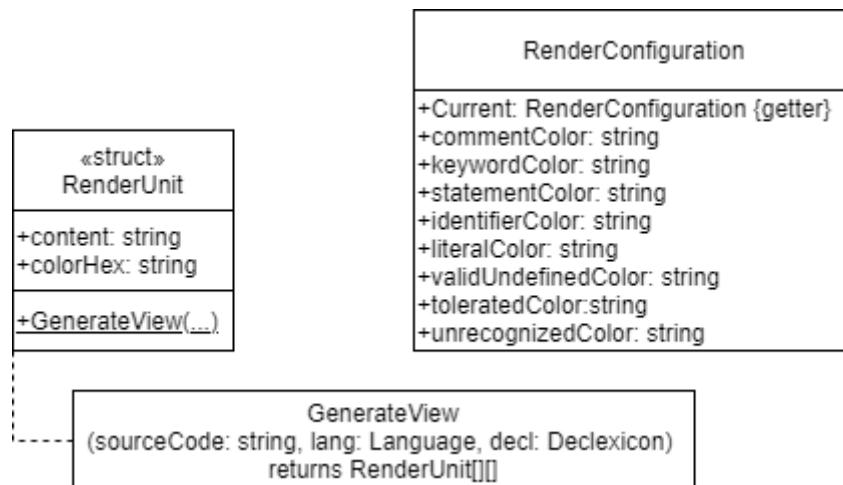
Esta función estática utiliza funciones locales para realizar un análisis sintáctico del código fuente del script similar al análisis del compilador, pero añadiendo unidades de renderizado para completar el array bidimensional. Su principal objetivo es identificar a qué tipo pertenecen las palabras de cada línea del código según el lenguaje y el declexicon, y les asigna un color según la interpretación.

El análisis sintáctico para el renderizado identifica de manera independiente y unitaria:

- ▶ Líneas que son un comentario
- ▶ Tokens de *Statements*.
- ▶ Expresiones tras un *Statement*, si se ha indicado un tipo en su definición en el procesador.
- ▶ Declexemas que construyen una declaración.
  - Palabras clave.
  - Palabras toleradas
  - Literales, dando por válido cualquier contenido literal.
  - Identificadores. Si se obtiene un contenedor del identificador, el análisis valida si el contenido del *string* coincide con algún objeto identificado en el contenedor.
- ▶ Cualquier otra palabra se considera no válida.

El método con el análisis sintáctico asignará un color concreto a cada encapsulación unitaria en función del tipo encontrado. Para consultar los colores, una clase *RenderConfiguration* con patrón *singleton* permite serializar cada color como hexadecimal en *string*.

**Figura 40.** Diagrama de clases para unidades de render de script.



Fuente: Elaboración propia.

En la figura 41 se muestra un ejemplo de código del proceso durante el análisis sintáctico de la función generadora de *RenderUnit* donde se comprueba la validez del identificador y se añade un nuevo *RenderUnit* a la lista de la línea actual en función de los resultados. Si no se consigue un contenedor del identificador, se dará por válida la palabra, si hay contenedor, la palabra analizada deberá coincidir con alguno de sus IDs.

**Figura 41.** Fragmento de código con análisis de identificador para renderizar.

```
...
else if(current.SubsequentIdentifier != null)
{
    bool validIdentifier = true;

    var container = current.SubsequentIdentifier.ProvideContainer();

    if(container != null)
    {
        validIdentifier = false;
        foreach(var id in container.GetAllIdentifiers())
            if(id == word)
            {
                validIdentifier = true;
                break;
            }
    }

    if(validIdentifier)
        AddIdentifier(current.SubsequentIdentifier, word);
    else
        AddUnrecognized(word);

    current = current.SubsequentIdentifier;
}
...
```

Fuente: Elaboración propia.

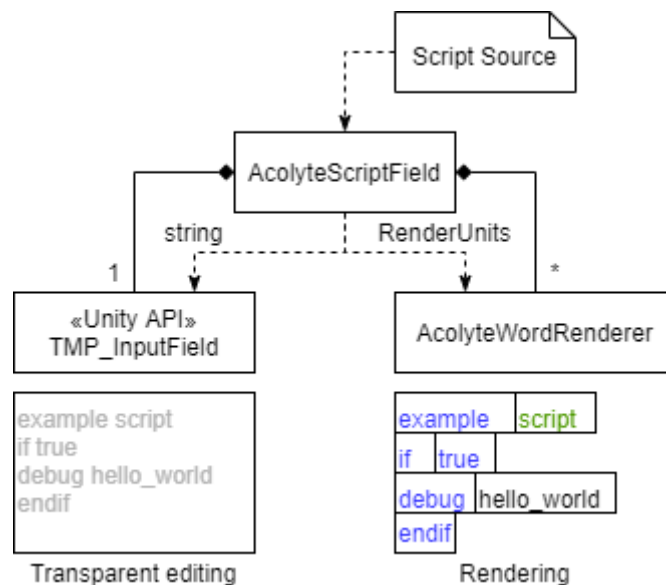
## Edición y renderizado de texto en Unity

Acolyte provee de varios componentes contruidos sobre la API de UI de Unity para permitir la edición integrada de texto. Por un lado, se utiliza el componente *TMP\_InputField* del paquete *TextMeshPro* de Unity, el cual renderiza un componente de texto y permite la introducción y manipulación del texto, pero debido a algunas limitaciones en su funcionamiento, Acolyte lo renderiza transparente y lo usa sólo para el control de *input*, y utiliza su propio sistema para la visualización del texto en sí, que es mostrado por encima estructurando componentes que renderizan cada *RenderUnit* obtenido en el array bidimensional.

Para ello, una clase *AcolyteScriptField* define el elemento de interfaz que está compuesto por un *InputField* y por una lista dinámica de *AcolyteWordRenderers*, otra clase creada que

contiene un componente de texto y son capaces de interpretar un *RenderUnit*. Cuando un script se asigna al *ScriptField*, se envía al completo al *InputField* para permitir su edición. Ante cada cambio del contenido de *InputField*, *ScriptField* genera el array bidimensional para renderizar el texto, rellenando cada *WordRenderer* con el contenido resultante y estableciendo las posiciones y tamaños de estos para reflejar adecuadamente la misma posición y tamaño que tendrían si estuvieran siendo renderizados por el *InputField*, actuando por tanto como su superposición con componentes individuales por cada *RenderUnit* (Figura 42).

**Figura 42.** Diagrama de clases para editar y renderizar un script.



Fuente: Elaboración propia.

Debido a la constante necesidad de analizar los cambios producidos en el *InputField* ante las modificaciones realizadas por un usuario, se hace uso de la lista dinámica de *WordRenderers* mediante *object pooling*<sup>12</sup> para garantizar la reutilización de componentes y asegurar el rendimiento. Para guardar un script, se utiliza el valor de texto del *InputField*, el cual es un

<sup>12</sup> Object Pooling es un patrón que implica no destruir objetos que se usan regularmente, sino mantenerlos desactivados cuando no se requieren para volver a utilizarlos si vuelven a necesitarse, evitando los costes de rendimiento por su construcción y destrucción constante.

reflejo de todas las ediciones que haya realizado el usuario hasta el momento, aunque este las haya estado visualizando a través de los *WordRenderers* sin saberlo.

Adicionalmente, una clase *AcolyteWordSelector* se ocupa de analizar la selección actual dentro del *InputField* de uso, para controlar sobre qué *WordRenderer* está el usuario en cada momento. El *ScriptField* utiliza esta información para mostrar una raya blanca subrayando el *WordRenderer* actual.

La figura 43 muestra el resultado del renderizado, con *Statements* (if, endif), *Expressions* (profile is expert), palabras clave (interact, using), palabra tolerada (with), identificador (AssemblyParts2), literal (herramienta) y una palabra no reconocida (abcdefg). Se muestra al identificador “AssemblyParts2” como seleccionado, con el signo de intercalación de la edición después de su primera “s”.

**Figura 43.** Editor de texto de Acolyte.

```
1  if profile is expert
2  interact with AssemblyParts2 using herramienta
3  abcdefg
4  endif
```

Fuente: Elaboración propia.

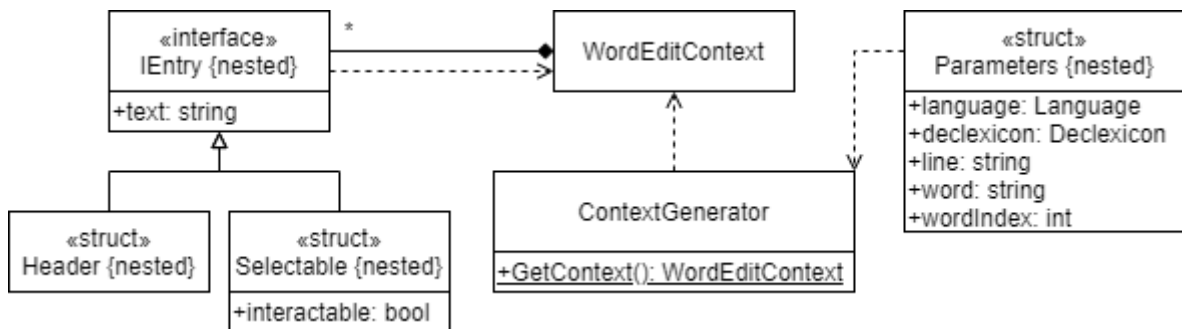
El uso de la selección de la unidad de renderizado actual tiene un uso fundamental para cumplir los requisitos asociados al uso de Acolyte, ya que esta selección se utiliza para mostrar el contexto asociado.

### Contextualización

Acolyte realiza un análisis sintáctico adicional en el editor únicamente de la línea en la que se encuentra el usuario durante la edición de texto con el propósito de contextualizar qué es posible escribir de manera válida. Este contexto permite fácilmente conocer el lenguaje y guía al usuario sobre qué es posible introducir a medida que edita el texto palabra por palabra.

Una clase *WordEditContext* compuesta por un array de entradas *IEntry* con información es creado a partir de este análisis de una línea, con el propósito de mostrar cada entrada en el interfaz como guía para el usuario. La clase *ContextGenerator* se ocupa de proporcionar una función estática pública para realizar el análisis sintáctico, devolviendo el *WordEditContext* adecuado (Figura 44).

**Figura 44.** Diagrama de clases para contextualización en editor.



Fuente: Elaboración propia.

Se utiliza información como la palabra seleccionada y su índice entre las unidades de la línea para saber cómo analizar el *string* y rellenar la lista de entradas con las que se creará el *WordEditContext* a retornar, pudiendo listar las siguientes entradas de información:

Para el índice 0:

- ▶ Indicación línea ignorada por comentario (*<Comment>*).
- ▶ Lista de posibles token de *Statements*.
- ▶ Lista de posibles palabras clave iniciales del declexicon.

Para índices posteriores, en función del análisis de las palabras anteriores:

- ▶ Lista de posibles *Expressions*, sólo si se ha indicado tipo en su definición.
- ▶ Lista de posibles palabras clave subsecuentes.
- ▶ Lista de posibles identificadores de objeto según el tipo del genérico del identificador.
- ▶ Indicación de posibilidad de introducir un literal (*<Literal>*).
- ▶ Indicación de palabra tolerada, a partir del declexema anterior (*<Tolerated>*).

Cualquier otro resultado aparecerá con una entrada *<Unrecognized>*.

Una clase para elemento de interfaz *AcolyteContextualizer* se ocupa de interpretar un *WordEditContext* y generar el contenido de la lista en función de sus entradas.

El componente *ScriptField* tiene una referencia al *AcolyteContextualizer* usado, y cuando detecta una nueva selección por parte del *WordSelector*, obtiene los parámetros necesarios y los envía al contextualizador.

En la figura 45 se observan tres ejemplos de contextualización: el primero mostrando las posibles introducciones si nos encontramos en la primera palabra, el segundo si nos encontramos en una línea teniendo previamente un “if”, mostrando por tanto la lista de expresiones booleanas, y el último cuando la palabra anterior es un declexema que tiene un identificador de objetos subsecuente.

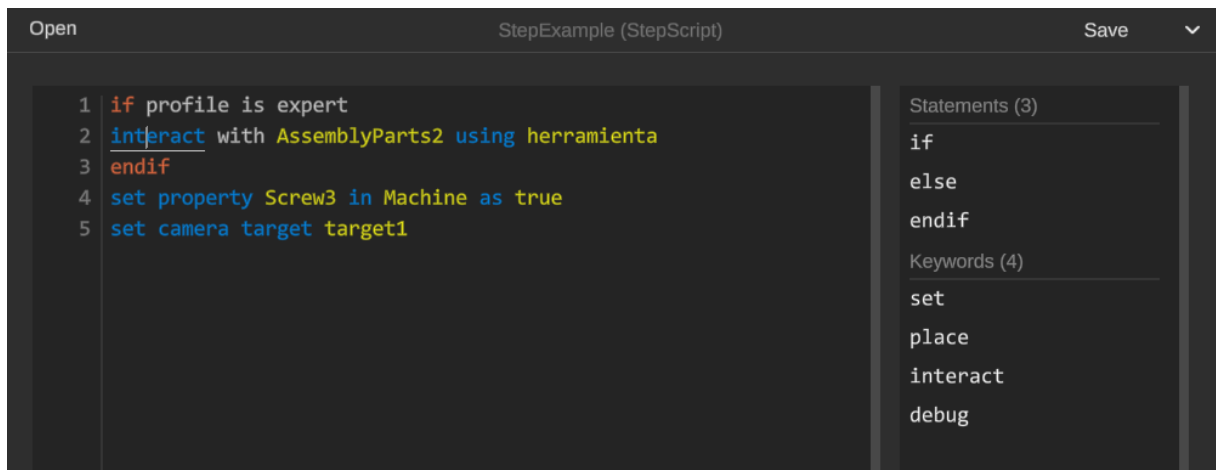
**Figura 45.** Ejemplos de contextualización en editor de Acolyte.

Statements (3)	Expression: boolean	Object identifiers (3)
if	profile is expert	Machine
else	booleanExample	AssemblyParts1
endif	true	AssemblyParts2
Keywords (4)	false	
set		
place		
interact		
debug		

Fuente: Elaboración propia.

La figura 46 muestra el resultado final del editor de Acolyte utilizando un script como ejemplo de prueba, donde se pueden observar también los botones de apertura de script, que permite abrir una lista con todos los scripts existentes en el proyecto, y el botón de guardado, que sobrescribe el código fuente del script y lo recompila si es necesario.

**Figura 46.** *Captura de editor de Acolyte.*



Fuente: Elaboración propia.

#### 4.2.9. Prueba de uso: Introducción

Para probar Acolyte se desarrolla un paquete software independiente que integra diferentes características de proyectos ajenos al sector del entretenimiento. El software de prueba consiste en un sistema que permite el desarrollo de procedimientos paso a paso enfocados en el aprendizaje, entrenamiento o apoyo al mantenimiento de procesos en sectores como la industria, defensa o salud. Se ha denominado a este software ProceUnity, y proporciona una base sobre la que crear la lógica concreta de cada proyecto según sus necesidades, ofreciendo modularidad y flexibilidad de adaptación.

Este software está originalmente preparado para tener lógica dirigida por datos modificables desde el editor de Unity, y el objetivo de este apartado del trabajo es realizar una integración de Acolyte con este software para comprobar su capacidad de reemplazar la edición existente y proporcionar nuevas funcionalidades a través de la creación de un lenguaje dedicado.

#### Entidades y propiedades

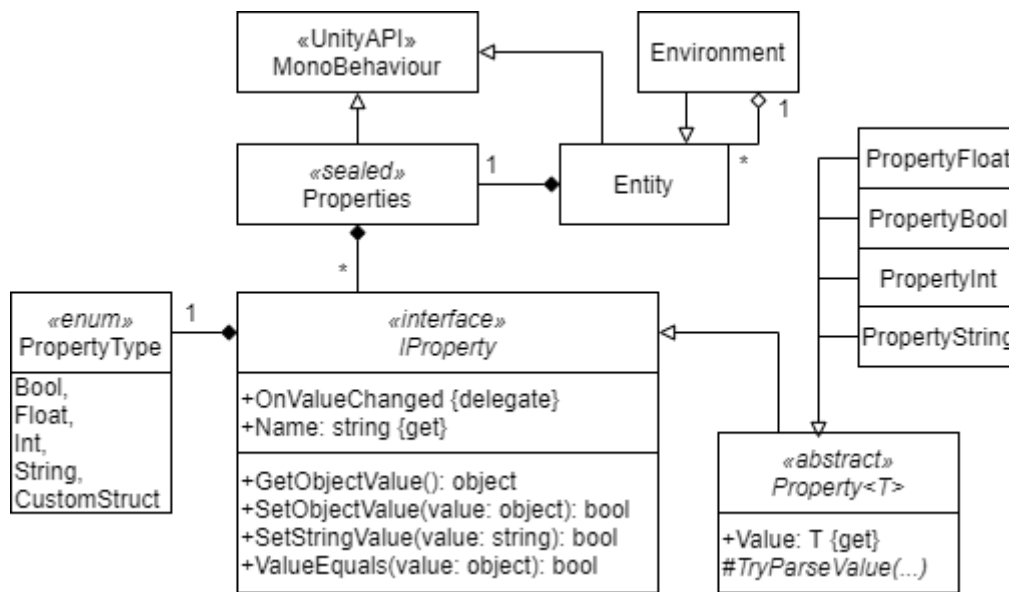
Para permitir el funcionamiento de los procedimientos, el software incluye una estructura de entidades con propiedades asignadas. Las propiedades suponen una encapsulación en una clase, *Property*, de una variable con un nombre y un tipo asociado, teniendo valores dinámicos interpretables en tiempo real. Se incluyen por defecto las principales variables primitivas:



*bool, int, float* y *string*. El software permite su extensión con nuevas propiedades, incluyendo soporte para estructuras.

Una entidad, *Entity*, es un objeto con una lista de propiedades asociadas, sirviendo como encapsulación para identificarse de manera unívoca y acceder a estas variables. En una escena de Unity, las entidades usables se añaden a una entidad que actúa como raíz denominada entorno, el *Environment*.

**Figura 47.** Diagrama de clases de entidades y propiedades.



Fuente: Elaboración propia.

Se facilitan diferentes utilidades para el uso de propiedades en el entorno de Unity, como la posibilidad de ser creadas a partir de las líneas de una variable string, utilizando un prefijo definido en cada clase de la propiedad (`int myProperty = 2`), un *PropertiesTracker* para el seguimiento de un array de propiedades, permitiendo el seguimiento de sus cambios, y varios componentes de Unity (*MonoBehaviours*) para su uso, a partir de la clase abstracta *PropertiesBehaviour*:

- ▶ Un *PropertiesLoop* que permite asignar una lógica de comportamiento a un array serializado de propiedades ejecutada cada *n* segundos.
- ▶ Un *PropertiesObserver* usado para especificar lógica ejecutada ante cualquier cambio en las propiedades indicadas en un array serializado.

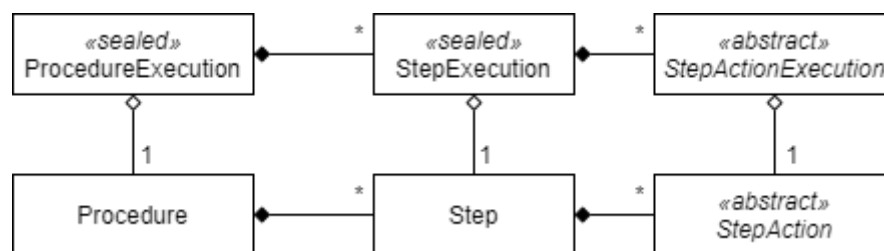
- Un *PropertiesPawn* para ejecutar lógica a un array serializado de propiedades a través de una función pública que deberá ser llamada cuando se desee.

## Pasos y procedimientos

ProcedUnity incluye una estructuración en forma de procedimientos formados por pasos, que contienen la información básica asociada a cada uno. Los pasos, a su vez, contienen una lista de *StepAction*, la clase abstracta base cuyas generalizaciones deben contener el conjunto de datos utilizado para su funcionamiento.

Este funcionamiento se realiza a partir de las ejecuciones, clases dedicadas a cada tipo que se encargan de su ejecución dentro del funcionamiento paso a paso: el *ProcedureExecution*, el *StepExecution*, y el abstracto *StepActionExecution*, del que se debe generalizar para especificar la lógica de funcionamiento de un tipo de acción.

**Figura 48.** Diagrama de clases de procedimientos.



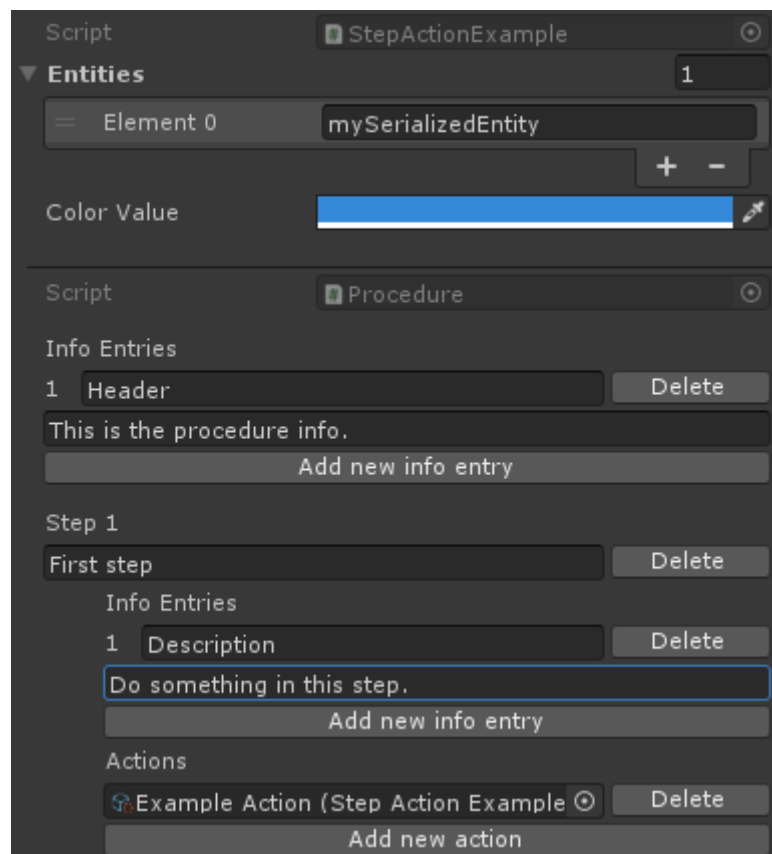
Fuente: Elaboración propia.

- El *ProcedureExecution* permite comenzar la ejecución de un *Procedure*, y se encarga de instanciar e inicializar los *StepExecution* adecuados en función de la lista de *Step*. Según el estado actual, permite pasar al siguiente paso, reiniciar el paso actual, o cancelar el proceso.
- El *StepExecution* se encarga de controlar la ejecución del *Step* asignado al inicializar por parte del *ProcedureExecution*. Instancia e inicializa a su vez los *StepActionExecution* en función de la lista de *StepAction*, y controla su estado interno en función de las acciones, completándose cuando cada acción se haya completado. Adicionalmente permite reiniciarse y autocompletarse.

- Cada *StepActionExecution* utiliza los datos del *StepAction* asignado y procesa la lógica definida por su clase generalizadora. Están basados en cuatro funciones: *Initialize(...)*, *Execute(...)*, *UndoExecution(...)* y *SetExecutionCompletedState(...)*. La lógica interna debe manejar un callback para notificar de cuándo se han completado.

A través de la creación en código de nuevos *StepAction* y su *StepActionExecution*, se permite la creación de todo tipo de implementaciones con infinita complejidad, funcionando sobre el proceso de procedimientos paso a paso. La edición en Unity se realiza a través de los mencionados *ScriptableObjects*, que permiten la creación de assets con datos en el editor del motor. *Procedure* y *StepAction* heredan de *ScriptableObject*, mientras que *Step* es una clase serializable dentro del asset de *Procedure*. La figura 49 muestra un procedimiento y un asset derivado de *StepAction* que ejemplifica serializando una lista de entidades y un color.

**Figura 49.** Assets de edición de *Procedure* y *StepAction*.



Fuente: Elaboración propia.

En la figura 50 se observa el código que inicializa las ejecuciones a partir de los datos de acción contenidos en la lista de *Step*.

**Figura 50.** Fragmento de código con ejecución de acciones de paso.

```
public StepExecution(..., Step step, ...)
{
    ...
    InitializeActionExecutions(step.actions.ToArray());
    ...
}
...
public void Execute(Action OnStepCompleted)
{
    ...
    for(int i = 0; i < actionExecutions.Length; i++)
    {
        pendingActionExecutions.Add(actionExecutions[i]);
        actionExecutions[i].Execute(HandleActionExecutionCompletion);
    }
    ...
}
```

Fuente: Elaboración propia.

El software incluye otras funcionalidades de utilidad, entre las que se encuentra principalmente la gestión de un registro, que incluye un interpretador abstracto para ser generalizado y utilizado como se desee (envío de datos a API externa, notificaciones, guardado de archivos de registro, gestión de eventos, etc.) y un interfaz dedicado con el que depurar y probar los procedimientos y las entidades y propiedades existentes.

#### 4.2.10. Prueba de uso: Demo

Para llevar a cabo la prueba de integración, se realiza primero una demo sobre el paquete software concretando sus aspectos abstractos. La demo consiste en la realización de un proceso paso a paso de interacción simple sobre un modelo 3D simplificado de robot industrial.

Para esta operación se crean una serie de clases que permiten llevar a cabo interacción con objetos de la escena para llevar a cabo una acción de paso. Un componente *Interactor* permite a un objeto de escena ser clicado para cambiar el valor de una propiedad, y contiene código

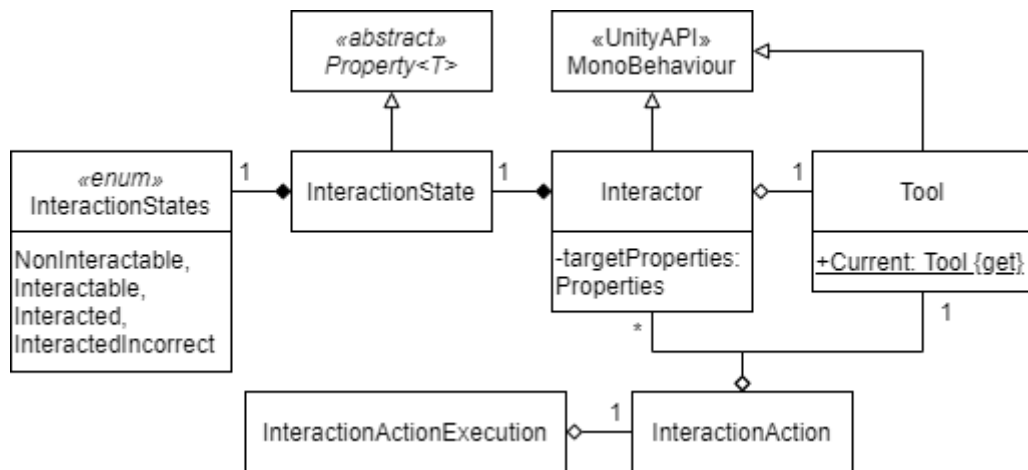
que escucha cambios en dicha propiedad para interpretarlos, haciendo uso de un patrón estrategia. La dependencia en el uso de propiedades implica que es posible realizar cambios que no necesariamente pasan por el componente *Interactor* provocando que aun así éste sea capaz de procesar su estado correctamente, algo que puede ocurrir por ejemplo con el uso de autocompletar en la ejecución de un paso.

Para este control de estado se ha optado por especificar una nueva propiedad personalizada *InteractionState* que hace uso de un enumerador con los valores: *NonInteractable*, *Interactable*, *Interacted* e *InteractedIncorrect*. La estrategia actual para interpretar la propiedad hace uso de un grupo de animaciones que permiten cambiar el color del objeto afectado por la interacción, palpitando en azul cuando está *Interactable*, coloreando en verde cuando está *Interacted*, y coloreando en rojo cuando su valor es *InteractedIncorrect*.

A partir de las clases abstractas *StepAction* y *StepActionExecution* se generalizan *InteractionAction* e *InteractionActionExecution*, para gestionar los datos y la lógica de ejecución, teniendo *InteractionAction* un array de *Interactors* serializable, y la ejecución usando estos datos para controlar los cambios de la propiedad, registrando las interacciones existentes. La ejecución del paso se completa cuando todas las propiedades de *Interactors* en el array hayan pasado por el estado *Interacted* o *InteractedIncorrect*.

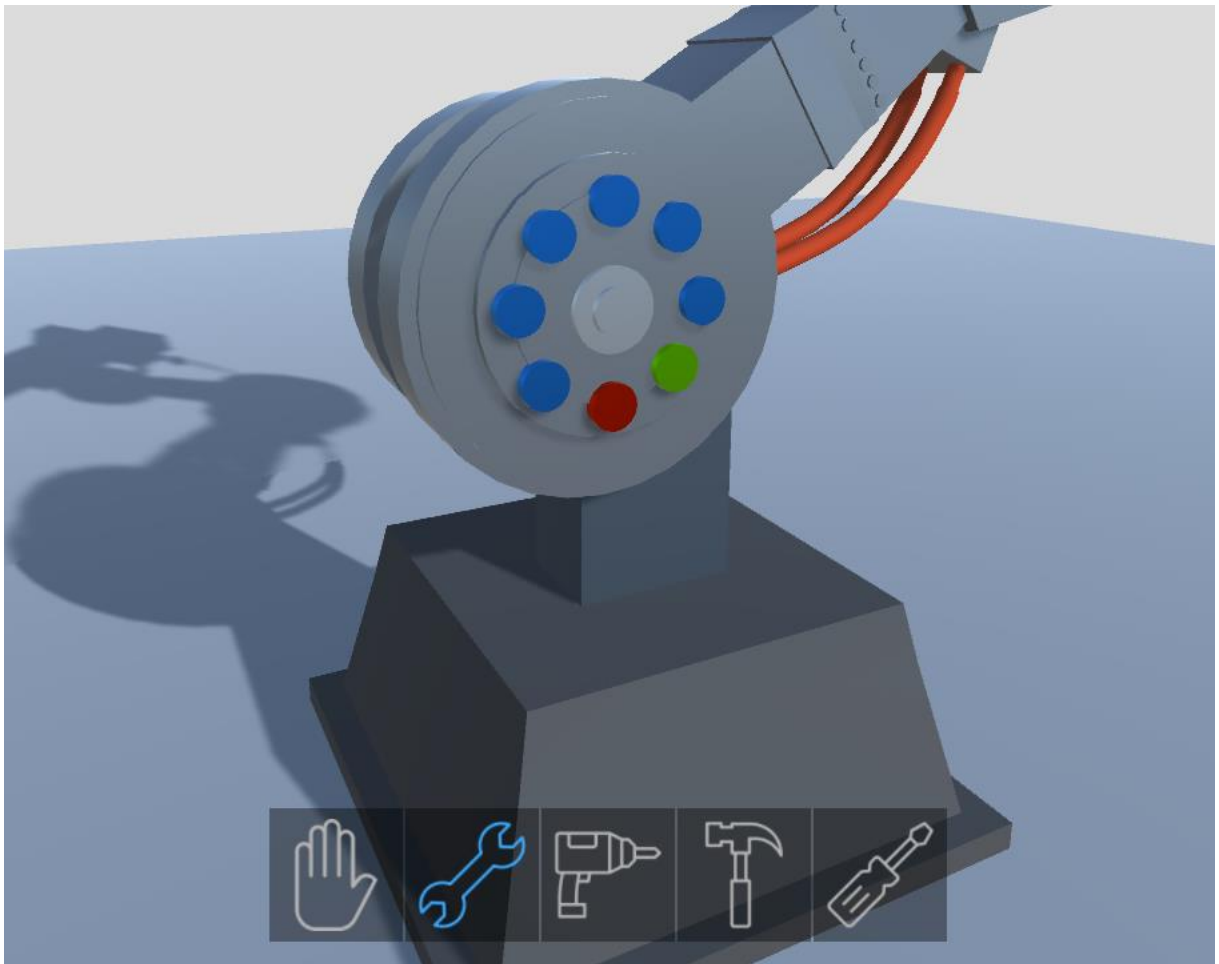
Adicionalmente, se hace crea un componente *Tool* que permite especificar en un *Interactor* una supuesta 'herramienta' requerida para hacer la interacción. *Tool* contiene un singleton para significar la herramienta seleccionada actualmente, y se incluye en escena mediante un menú de interfaz simple que permite seleccionar entre diferentes iconos de herramienta. La herramienta requerida debe por tanto serializarse en *InteractionAction* para ser luego pasada a todos los *Interactors* del array como requisito.

**Figura 51.** Diagrama de clases de interacción en la demo.



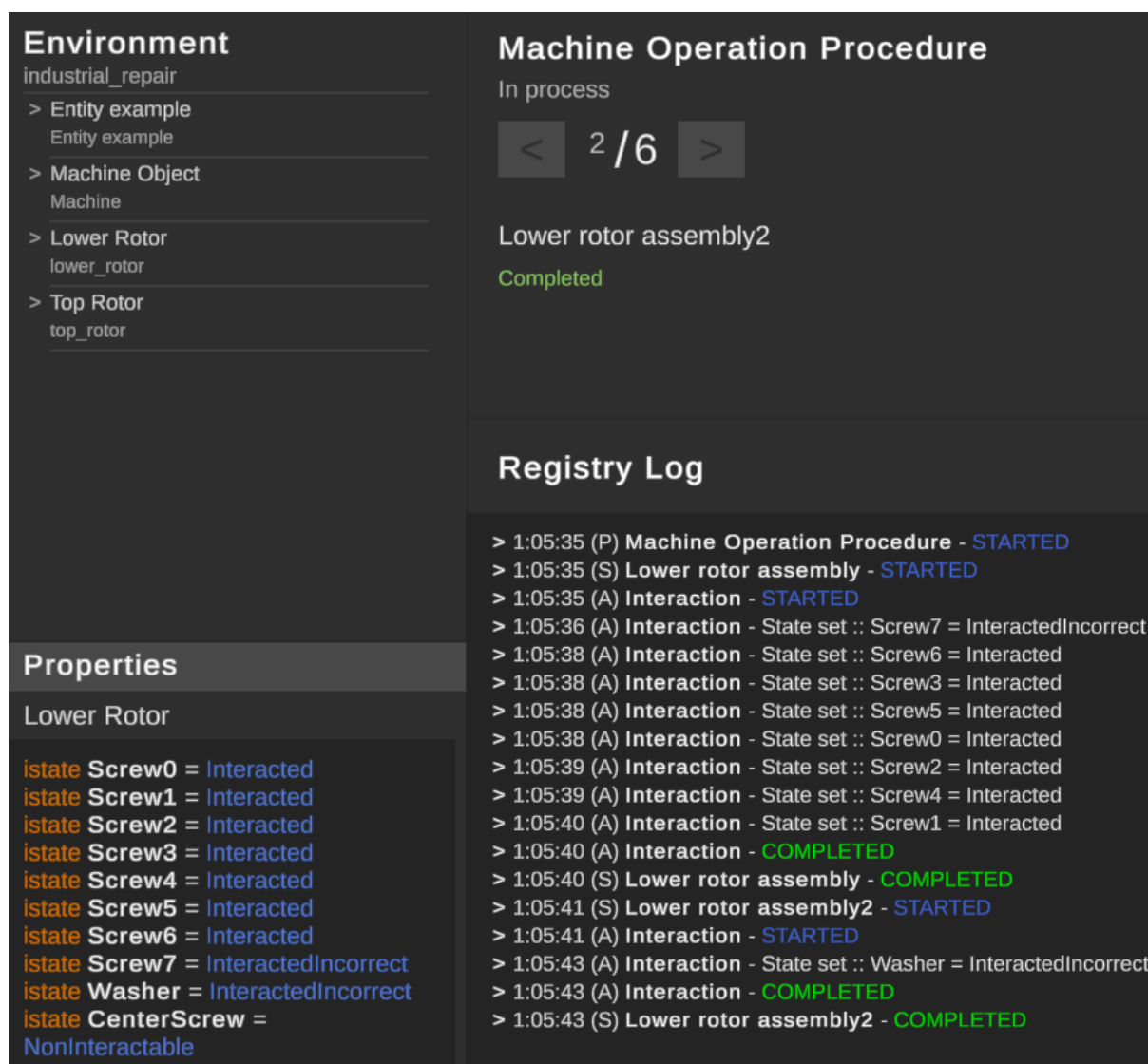
Fuente: Elaboración propia.

**Figura 52.** Captura de ejecución de procedimiento demo.



Fuente: Elaboración propia.

**Figura 53.** Captura de interfaz de depuración de procedimientos.



Fuente: Elaboración propia.

#### 4.2.11. Prueba de uso: Integración

A partir de ProceUnity se lleva a cabo la integración de Acolyte y su prueba mediante la demo desarrollada.

#### Requisitos básicos de inclusión de Acolyte

Para permitir el uso de Acolyte es mandatario realizar tres implementaciones básicas:

- Una generalización de *Language*, donde debemos definir nuestro lenguaje.

- ▶ Una generalización de *ScriptAsset*, donde debemos especificar el lenguaje previamente creado como parámetro genérico para poder tener assets que representen scripts de nuestro nuevo lenguaje.
- ▶ Una generalización de *Declexicon*, cuya creación de instancia deberemos incluir en nuestro nuevo *Language* a través de la función obligada por herencia *CreateDeclexicon()*, siendo el objeto con estado donde debemos definir el árbol de declexemas con las funciones deseadas.

Para su uso en el software, se implementan las clases *StepScript*, *StepScriptAsset* y *StepScriptDeclexicon*.

### StepScript

El objetivo fundamental de *StepScript* es sustituir el uso de un array de acciones serializadas mediante referencias a assets en el editor de Unity para definir la lógica de un paso por una referencia a un script, permitiendo especificar qué debe ocurrir ante la ejecución de un paso mediante código editable en tiempo de ejecución, evitando la dependencia en assets independientes usables sólo desde el editor de Unity y condensando esta edición en líneas de código con declaraciones expresivas. Para cumplir con este objetivo se utiliza la siguiente estrategia:

- ▶ Un *Step* contiene ahora una referencia a un *StepScriptAsset*.
- ▶ *StepScriptDeclexicon* contiene una lista dinámica de *StepActions*.
- ▶ Las declaraciones creadas en el declexicon pueden crear una nueva *StepAction* mediante una de sus palabras clave y añadirla a la lista.
- ▶ Una función interna permite obtener la última acción añadida con un tipo concreto mediante parámetro genérico.
- ▶ Las funciones asignadas a declexemas subsecuentes obtienen la última *StepAction* añadida en el tipo correcto y modifican sus datos internos según la declaración.
- ▶ El callback del declexicon se invoca devolviendo la lista como array, y es utilizado por *StepExecution* al ejecutar el script de su *Step* como alternativa al array que previamente se encontraba serializado en los datos del propio *Step*.



La figura 54 muestra la nueva implementación en *StepExecution* en contraposición con la figura 50.

**Figura 54.** Fragmento de código con ejecución de acciones de paso usando script.

```
public void Execute(Action OnStepCompleted)
{
    ...
    step.Script.Execute((StepAction[] stepActions) =>
    {
        InitializeActionExecutions(stepActions);

        for(int i = 0; i < actionExecutions.Length; i++)
        {
            pendingActionExecutions.Add(actionExecutions[i]);
            actionExecutions[i].Execute(HandleActionExecutionCompletion);
        }
    });
    ...
}
```

Fuente: Elaboración propia.

### Prueba de declaraciones

Para permitir crear un lenguaje adaptado al paquete *ProcedUnity*, se hace uso de las capacidades de modularización de *Decllexicon* para hacer a *StepScriptDecllexicon* una clase parcial y establecer la clase principal con la integración base de *Acolyte* y un apartado de esta clase en la carpeta con el contenido demo, con las implementaciones exclusivas para este. En la carpeta de la demo, se incluye también un contenedor con patrón singleton para objetos de tipo *Tool*, permitiendo identificarlos.

Para la demo se añade un árbol de decllexemas que permita crear las interacciones como prueba de integración. Se incluye el siguiente proceso:

- ▶ Palabra clave “interact” asociada a una función que instancia una *InteractionAction* y la añade a la lista de *StepAction* del decllexema. Se le añade tolerancia a “with”.
- ▶ Identificador subsecuente de tipo *GameObject[]*, que permite elegir entre los objetos identificados en escena, asociado a una función que añade el componente *Interactor* obtenido de cada uno de los objetos identificados al *InteractionAction* previo.
- ▶ Palabra clave subsecuente “using” utilizada por motivos de expresividad.

- Identificador subsecuente de tipo *Tool*, que permite elegir entre los *Tools* identificados en escena, asociado a una función que añade la referencia obtenida al *InteractionAction* previo.

Esto permite declaraciones como: “interact with {object}” para acciones sin herramienta requerida o “interact with {object} using {tool}” para especificar una herramienta.

### **Statements y Expressions**

Para probar el uso de estas dos características, se crea una clase sobre la demo que permite especificar un modo de herramienta, pudiendo elegir en un enumerador entre “manual” y “electric”. Se facilita un interfaz con dos botones en escena para realizar este cambio de estado. Desde la demo, se añade al lenguaje *StepScript* dos expresiones: “manual mode” y “electric mode” que resuelven a bool en función del modo actual (Figura 55).

**Figura 55.** Fragmento de código con creación modularizada de *Expressions* en demo.

```
public class ToolModeExpressions
{
    [RuntimeInitializeOnLoadMethod()]
    private static void AddExpressions()
    {
        StepScript.OnInstanceCreated += (StepScript stepScript) =>
        {
            stepScript.AddExpression("manual mode",
                () => { return ToolMode.Mode == ToolModes.manualMode; });
            stepScript.AddExpression("electric mode",
                () => { return ToolMode.Mode == ToolModes.electricMode; });
            ...
        }
    }
}
```

Fuente: Elaboración propia.

Se añade también a *StepScript* el *Statement* de tipo *IfElse*. Mediante estos dos añadidos, se pueden realizar derivaciones condicionales en ejecución de script en función del modo escogido por el usuario, y definir más de una declaración de interacción con herramientas diferentes.

## Ejecución de funciones

El uso de scripts ejecutables permite ahora a ProcedUnity añadir cómodamente otro tipo de lógica sin depender de assets con datos ni ningún tipo de implementación adicional.

Como prueba, se añaden las siguientes declaraciones básicas:

- ▶ Palabra clave “set” como inicio de declaración.
- ▶ Palabras clave subsecuentes “property” y “camera” como dos posibles caminos.
- ▶ Para “property”, literal subsecuente para seleccionar un nombre de propiedad que es guardado en una variable parte del declexicon.
- ▶ Se continúa el literal con palabra clave “in”, seguido de un identificador de entidades, que permite indicar a quién pertenece la propiedad buscada, cuya referencia también se guarda mediante el estado del declexicon.
- ▶ Se continúa con “as”, seguido de un literal para especificar el valor a asignar a la propiedad. La función asignada a este último declexema utiliza las variables locales asignadas previamente para acceder a la propiedad de la entidad y cambiar su valor por el nuevo.
- ▶ Para “camera”, palabra clave subsecuente “target”, seguido de un identificador de objetos con un contenedor asignado específicamente dedicado a identificar objetos a usar como punto de referencia de la cámara 3D. Función asignada que proporciona el objeto identificado al componente de comportamiento de la cámara.

Se permiten, ante la ejecución de un paso, operaciones como “set property {property} in {entity} as {value}”, o “set camera target {object}”.

## Eventos

Otro posible uso se demuestra mediante eventos. Se crea una clase *Event* que encapsula un delegado para permitir invocar eventos identificados por un nombre. Se implementa adicionalmente un contenedor para eventos. En cualquier lugar del código se puede crear una instancia de nuevo evento e invocar cuando se crea conveniente.

Se implementa la siguiente declaración como prueba demostrativa:

- ▶ Palabra clave “when”, seguido de identificador de eventos, cuyo resultado se guarda como variable en el declexicon.

- Palabra clave subsecuente “alert”, seguido de literal para introducir un mensaje de texto.  
La función asignada al literal encapsula una función para activar una alerta en pantalla que es añadida al delegado del evento para su ejecución cuando este se invoque.

Utilizando un nuevo componente *Alert* asignado a un interfaz de usuario, se permite utilizar declaraciones como “when {event} alert {your\_message}”, que activarán el interfaz en pantalla mostrando la alerta cuando se invoque al evento. Debido a la limitación actual de Acolyte para procesar literales con el mismo carácter separador utilizado para separar las palabras en el análisis sintáctico, se reemplazan los caracteres “\_” por “ ” antes de mostrar una alerta para permitir escribir espacios.

**Figura 56.** Fragmento de código con declaración de eventos en demo.

```
private Event selectedEvent;
...
var when = new Keyword("when");
var selectEvent = new Identifier<Event>(SelectEvent, GetEventsContainer);
var eventAlert = new Keyword("alert");
var eventAlertText = new Literal(EventAlert);
when.Then(selectEvent);
selectEvent.Then(eventAlert);
eventAlert.Then(eventAlertText);
...
private void SelectEvent(Event ev) => selectedEvent = ev;

private void EventAlert(string alert)
{
    if(selectedEvent == null) return;

    void action() { Alert.ShowAlert(alert); }
    ev.action += action;
}
```

Fuente: Elaboración propia.

La figura 56 muestra todo el código necesario para implementar la declaración de eventos en el declexicon.

La figura 57 muestra un ejemplo de resultado final en forma de script plenamente funcional.

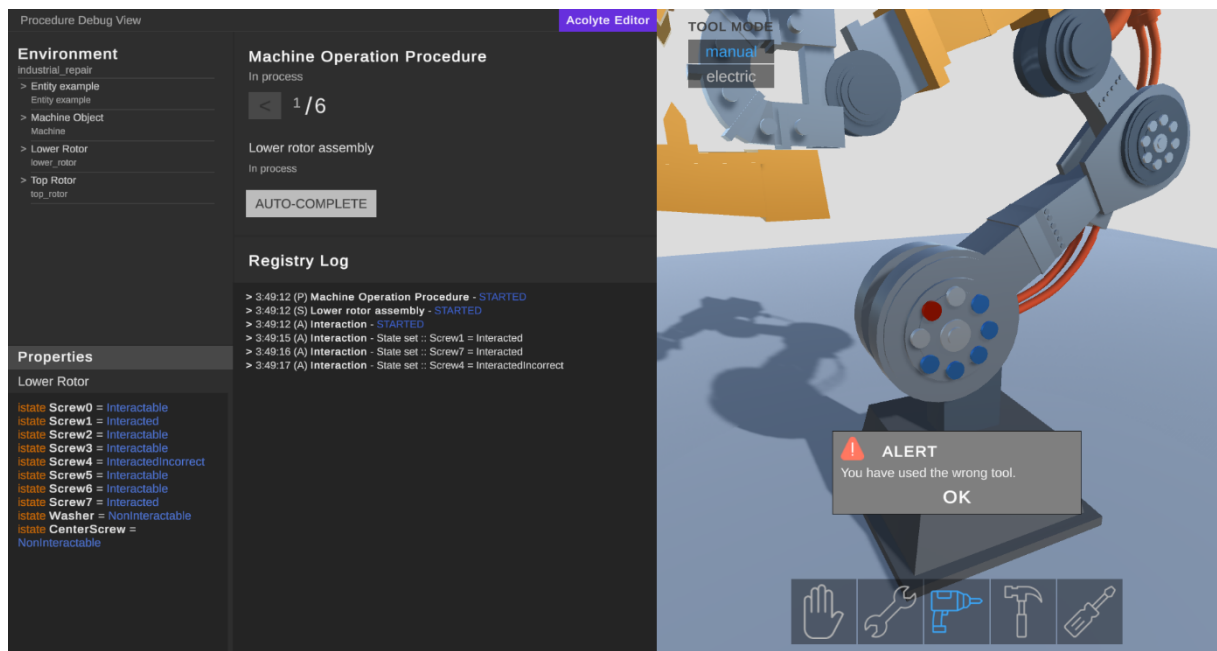
La figura 58 muestra una captura de la aplicación resultante ejecutándose.

**Figura 57.** Código ejemplificando StepScript en demo.

```
set camera target target1
if manual mode
interact with LowerRotorOuterScrews using wrench
else
interact with LowerRotorOuterScrews using drilling-machine
endif
when Interaction_Incorrect alert You_have_used_the_wrong_tool.
```

Fuente: Elaboración propia.

**Figura 58.** Imagen de aplicación demo final.



Fuente: Elaboración propia.

## 4.3. EVALUACIÓN

### 4.3.1. Integración en motor, interoperabilidad y dependencias

El software desarrollado se encuentra integrado en el motor de videojuegos Unity sin ningún tipo de dependencia adicional. La base fundamental del código está desacoplada del motor, exponiendo un diseño independiente y reutilizable en otros entornos con dependencias únicamente en el lenguaje utilizado y las librerías básicas del entorno .NET para C#.

El editor muestra dependencias inherentes en múltiples componentes del motor, pero también contiene parte de lógica desacoplada para los análisis sintácticos y la información de renderizado, pudiendo adaptarse con facilidad a otros entornos de desarrollo o a otros sistemas de interfaz dentro de Unity.

La inclusión del software en cualquier proyecto realizado con Unity es extremadamente sencilla, requiriendo únicamente de añadir las carpetas con el contenido de Acolyte y su editor en algún sitio del proyecto para que este sea compilado automáticamente y esté disponible para el resto del código con total interoperabilidad.

#### 4.3.2. Simplicidad, requisitos de conocimiento de interfaz y facilidad de uso

El diseño de Acolyte está centrado en simplificar el proceso de desarrollo de un nuevo lenguaje. Acolyte permite realizar este desarrollo requiriendo únicamente conocimientos en los siguientes dominios<sup>13</sup>:

- ▶ Conocimientos del lenguaje C#.
- ▶ Conocimientos básicos acerca del funcionamiento del motor Unity.
- ▶ Conocimientos de la API de Acolyte.

Se estima que cualquier desarrollador que conozca un lenguaje como C# y tenga conocimientos de desarrollo en tecnologías de videojuegos podrá comprender con facilidad las características ofrecidas por su API y conceptos como *Script*, *Statements* o *Expressions*.

Para el proceso de desarrollo de un lenguaje en Unity utilizando Acolyte se establecen los siguientes pasos mínimos:

- ▶ Creación de script C# con generalización de *Declexicon* donde incluir la lógica declarativa.
- ▶ Creación de script C# con generalización de *Language*.
- ▶ Creación de script C# con generalización de *ScriptAsset* para el asset contenedor de los scripts del lenguaje.

---

<sup>13</sup> Se obvia especificar la necesidad de conocimientos sobre el código incluido en Unity por los desarrolladores sobre el que se pretenda interoperar con un nuevo lenguaje.

El uso mínimo funcional de Acolyte, consistente en la creación de declaraciones mediante el *Decllexicon* y sus *Decllexemas* implica conocer un total de 7 clases: *Script*, *Language*, *Decllexicon*, *Keyword*, *Literal*, *Identifier* y el conocimiento del interfaz *IIdentifierContainer*.

El proceso de ejecución para un nuevo lenguaje requiere únicamente de referenciar una instancia de *Script* a través de un *ScriptAsset* y hacer uso de su función *Execute(...)*. El diseño del decllexicon implica que un programador es libre de utilizar el contexto y el tipo de callback como crea conveniente.

La implementación de árboles de decllexemas utilizada en la prueba de integración contiene el siguiente número total de líneas de código, contando espacios e indentación estándar:

- ▶ Declaraciones base de Acolyte sobre *ProcedUnity*: 144 líneas.
- ▶ Declaraciones para creación de acciones de interacción en demo: 58 líneas.

La generalización de *Decllexicon* implementada contiene en total adicionalmente:

- ▶ 6 declaraciones de variables (para gestión interna de estado).
- ▶ 11 funciones de asignación a decllexemas.
- ▶ 4 funciones de inicialización (2 + 2 debido a modularización de la clase).
- ▶ 5 funciones de tipo *get* para facilitar la obtención de contenedores.

Estos números demuestran que realizar una implementación de *Decllexicon* es una tarea fácilmente abordable, donde la mayoría de la implementación consiste en organizar las variables y funciones que permiten construir los árboles de decllexemas, teniendo en gran medida dependencia en las preferencias y conocimientos de los desarrolladores y no en una imposición de Acolyte que requiera conocimientos adicionales.

## Visibilidad

El código de Acolyte incluye consideraciones para mejorar su API reduciendo la visibilidad innecesaria. Hace uso de nombres de espacios para dividir modularmente la API y de clases parciales y anidación de clases privadas para mejorar la organización y el proceso de desarrollo sobre Acolyte otorgando características que no invaden el espacio público de uso. La implementación de compilación es un ejemplo de adecuación a los requisitos mínimos de interfaz. Las clases *Compiler* y *Executable* únicamente deben ser referenciadas y usadas por la

clase *Script*, por lo que ambas son clases privadas anidadas bajo *Script*, evitando aumentar la cantidad de información disponible en la API de Acolyte.

Para reducir la cantidad de clases se utilizan también patrones de paradigma procedural como el proceso de análisis sintáctico para la creación de *RenderUnits* para el editor, siendo una estructura que contiene una función pública estática que realiza todo el proceso organizándose a través de múltiples funciones locales a la función.

## Editor

La edición de scripts es intuitiva a través de las guías de color para las palabras y la contextualización. Es posible utilizar un script con un nuevo lenguaje sin apenas conocerlo gracias a los contextos que guían la escritura a medida que se añaden palabras, sobre todo de cara a validar palabras más peligrosas como *strings* de identificación de objetos.

Las guías de usuario no obstante no son suficiente como para competir con la facilidad de uso de una herramienta con un diseño de UX dedicado, por lo que los esfuerzos en esta dirección deben estar enfocados en facilitar el mayor nivel de abstracción posible en el lenguaje.

La falta de contextualización según el progreso actual de escritura, similar a un contexto de autocompletar, puede resultar un problema si la cantidad de palabras posibles es muy elevada.

Existe una falta de información con respecto a un lenguaje que permita especificar datos asociados a cada palabra en forma de comentarios, aclaraciones, etc., permitiendo mostrar *tooltips* al pasar el cursor por encima de una palabra, dificultando la comprensión del lenguaje editado y necesitando por tanto de documentación externa sobre el mismo para su uso.

## Complejidad Aditiva

El propio desarrollo de Acolyte y su uso proveen de un diseño adaptado a añadir complejidad de manera aditiva a través de características como las instrucciones en forma de objetos generalizables, o la existencia de *Statements* y *Expressions*. Usar estos apartados del software y añadir nuevos es una tarea relativamente sencilla, aunque las implementaciones actuales son demasiado simples como para demostrar usos más complejos, como podrían ser



expresiones con un análisis sintáctico de su contenido y manejo de operadores lógico-matemáticos.

#### 4.3.3. Flexibilidad, modularización y capacidad de adaptación

Acolyte propone un diseño que facilita libertad y flexibilidad en su uso por parte de un equipo desarrollador que pretenda desarrollar un lenguaje:

- ▶ Acolyte no fuerza ningún patrón concreto para la implementación de la lógica declarativa, otorgando libertad flexible con el uso del objeto con estado *Declexicon*.
- ▶ Garantiza interoperabilidad total con el resto de código gracias al uso de C# compilado en Unity.
- ▶ Herramientas como los identificadores y el interfaz de contenedor de identificación permiten realizar implementaciones muy flexibles para identificar todo tipo de objetos serializados según las preferencias de cada implementación.
- ▶ La creación de un lenguaje se permite de manera totalmente modular a lo largo de varios paquetes software, incluyendo declexemas modulares a través de clases parciales, así como el añadido modular de *Statements* y *Expressions* a un lenguaje.
- ▶ La implementación actual de *Expressions* es extremadamente limitada, careciendo de ningún tipo de análisis sintáctico de una línea de código.
- ▶ Los análisis sintácticos se realizan exclusivamente línea a línea y la separación de palabras se hace mediante un carácter concreto indicado en los datos de lenguaje. La falta de definición de alcances como alcances para un *Statement*, alcances para procesar el contenido dentro de corchetes o paréntesis, o alcances para procesar literales en función de caracteres de principio y fin, es un problema para un uso más efectivo de Acolyte.
- ▶ En este trabajo se ejemplifican diferentes posibles usos dentro del declexicon, como variables locales al objeto, clases anidadas, uso de *singletons* o variables públicas estáticas. La flexibilidad ofrecida por el diseño del declexicon es de gran utilidad para adecuar la implementación a las preferencias y necesidades de cada desarrollador o proyecto, pero puede resultar confuso cómo realizar este proceso, careciendo de ningún tipo de patrón explícito o ayuda que lo guíe.

#### 4.3.4. Prueba de integración de uso

La prueba de uso de Acolyte sobre ProceUnity y la demo se considera un éxito para cumplir con los requisitos propuestos, habiendo sido capaz de cumplir con la funcionalidad básica de servir como sustitución de los assets de datos para configurar las acciones. El proceso de creación de la declaración que permite crear interacciones y la sustitución en código del uso de los assets de configuración por la ejecución del script obteniendo los mismos resultados ha sido fácil y de corta duración.

Adicionalmente, se ha podido demostrar la utilidad de otras posibles características incluyendo funcionalidades avanzadas sobre la ejecución de pasos a través de declaraciones que ejecuten funciones abstraídas que pueden ser de gran utilidad, incluyendo la capacidad de editar cómo responder a eventos, así como el añadido del *Statement IfElse* y del uso de una expresión booleana para comprobarse.

Se considera, no obstante, que son necesarias un mayor número de implementaciones en entornos más complejos para poner a prueba la efectividad de Acolyte de manera realista, especialmente en relación a demostrar que la flexibilidad otorgada por Acolyte, que permita cumplir con los requisitos en estos entornos, no quiebra los requisitos de simplicidad y facilidad de uso de Acolyte.

## 5. CONCLUSIONES Y TRABAJO FUTURO

### 5.1. CONCLUSIONES

Acolyte demuestra la posibilidad de simplificar el desarrollo de Lenguajes de Dominio Específico dentro de un motor de videojuegos sin la necesidad de utilizar herramientas externas ni conocimientos adicionales, facilitando su uso por desarrolladores expertos en tecnología de videojuegos. Se cumplen requisitos de simplicidad, facilidad de uso, flexibilidad, modularización y se demuestra su potencial con la creación del lenguaje simple *StepScript* en una demo sobre un paquete software independiente.

Acolyte provee de capacidades de creación de declaraciones y del añadido de complejidad aditiva, demostrando que la implementación actual tiene gran potencial de extensión, incluyendo características imperativas.

A través de su editor integrado, la creación y modificación de scripts para los lenguajes creados es una tarea intuitiva y guiada, haciendo uso de análisis sintáctico para colorear palabras y proporcionar un contexto indicando las posibilidades de escritura al usuario, mejorando la experiencia de uso de perfiles de reducida tecnicidad que tengan conocimientos de un dominio específico ajeno a la programación.

Mediante estas características, se estima que Acolyte y el desarrollo y uso de DSLs son viables como una parte integrada del desarrollo de software utilizando motores de videojuegos con el propósito de sustituir algunas de las necesidades de desarrollo orientadas a herramientas dedicadas, así como a la necesidad de utilizar lenguajes menos específicos y menos expresivos para programar lógica de comportamiento que no deba ser recompilada ni requiera una reconstrucción del software. Se considera especialmente óptimo para reducir los recursos necesarios en facilitar capacidades de configuración o edición fuera del editor de motor sobre las aplicaciones ya construidas.

Sin embargo la implementación actual de Acolyte muestra carencias en diferentes áreas importantes para facilitar su efectividad y mejorar la expresividad de los lenguajes creados, como posibilitar la definición de alcances y la claridad de uso de su proceso de implementación

de declaraciones. También existe una falta de características en su editor que faciliten su uso como alternativa a herramientas con un diseño de UX dedicado, como mejoras en la contextualización y la posibilidad de añadir información sobre el lenguaje para guiar en mayor medida la escritura de scripts.

La prueba de integración de uso es prometedora, habiendo cumplido los requisitos establecidos, pero el resultado no se considera suficiente para demostrar la efectividad de Acolyte en proyectos de mayor envergadura y complejidad requiriendo una implementación de mayor tamaño. Más pruebas con el uso de Acolyte son necesarias, así como pruebas de uso de usuarios finales sobre lenguajes creados en proyectos destinados a dominios específicos.

## 5.2. TRABAJO FUTURO

### Capacidades imperativas y variables

Acolyte se puede beneficiar de la inclusión de mayores capacidades imperativas, como pueden ser características de control de flujo como bucles.

Con menor prioridad, se podría incluir la definición y uso de variables dentro de un script, incluyendo el manejo de operadores y el uso de variables predefinidas en código, así como palabras clave aplicables al uso de variables (*const*, *static*, *readonly*, etc).

### Alcances

Una de las principales prioridades debería ser la inclusión de capacidades de procesar alcances en Acolyte. El principal reto, no obstante, es facilitar que la definición de cualquier alcance sea altamente configurable, permitiendo el uso de corchetes, paréntesis, comillas u otros caracteres para definir alcances en diferentes contextos, como pueden ser comienzo y fin de literales, comienzo y fin de expresiones analizables sintácticamente, o conjunto de líneas que deben procesarse en común a partir de un *Statement* o alguna declaración y deben incluir, por tanto, algún tipo de gestión de localidad.

### **Análisis complejo de *Statements***

La implementación actual de *Statements* es demasiado simple y limitante. Debería incluirse un proceso de análisis más complejo, incluyendo el procesado de operadores lógicos y matemáticos para interpretar un tipo y obtener un valor resultante.

### **Datos de información sobre lenguaje**

De manera modular y siguiendo la complejidad aditiva, se plantea incluir la posibilidad de asociar un conjunto de datos exclusivamente para el editor que determinen información asociada a palabras, y permitan aclaraciones a los usuarios del editor mediante *tooltips*.

### **Guías de edición**

La edición de scripts en el editor de Acolyte se puede beneficiar de mejoras de usabilidad en la contextualización, como utilizar el texto escrito para priorizar la información de contexto mediante lógica de autocompletar.

### **Implementación de decllexema**

Se plantea buscar diseños aplicables a la implementación de un decllexema que orienten y faciliten el proceso, simplificando en mayor medida la definición de declaraciones y el manejo de su estado y las variables necesarias para las funciones de decllexema.

### **Pruebas en proyectos reales**

Para demostrar en mayor medida las capacidades y la efectividad de Acolyte, se deberían realizar pruebas en proyectos de mayor envergadura aplicables a dominios específicos, tanto con Acolyte desde un inicio del proyecto como incluyendo Acolyte para proporcionar configuración y edición sobre un desarrollo preexistente.

### **Entornos de edición**

Se podrían buscar alternativas a la implementación actual del editor en Unity, incluyendo pruebas con otros paquetes de interfaz para el motor, o el desarrollo de un editor web que se pueda comunicar en tiempo real con el código de Unity a través de conexión por WebSocket o HTTP para consultar información del lenguaje y de contexto.

### **Procesado de errores y depuración**

Diseñar e implementar código sobre Acolyte que permita identificar errores y depurar los procesos de compilación y ejecución, para facilitar comprender posibles problemas relacionados con la ejecución de los lenguajes creados. Podría ser posible realizar una integración de estas capacidades en el editor, para tener funcionalidades como *breakpoints* o excepciones que se muestren en el propio interfaz y se notifiquen al compilar un script.

### **Elaboración de pruebas automatizadas**

Solidificar la implementación de Acolyte a través de la inclusión de pruebas automatizadas unitarias y de integración.

## Referencias bibliográficas

- Research and Markets (2020, Agosto). Global Game Engines Market By Type, By Application, and By End Use, Forecasts to 2027. Recuperado el 12 de Junio de 2021 de <https://www.researchandmarkets.com/reports/5206637/global-game-engines-market-by-type-by#src-pos-1>
- Politowski, C., Petrillo, F., Montandon, J. E., Valente, M. T. y Guéhéneuc, Y. (2020). Are Game Engines Software Frameworks? A Three-perspective Study. <http://doi.org/10.1016/j.jss.2020.110846>
- Blizzard Entertainment (2018, Junio 15). Dev Watercooler: World of Warcraft Classic. *Official World of Warcraft News*. <https://worldofwarcraft.com/en-us/news/21881587/dev-watercooler-world-of-warcraft-classic>
- Dan Staines (2017, Febrero 27). Unsung Heroes of the Games Industry: Tools Programmers. *IGN Entertainment*. <https://www.ign.com/articles/2017/02/06/unsung-heroes-of-the-games-industry-tools-programmers>
- Mernik, M., Heering, J. y Sloane, A.M. (2005, Diciembre 1). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4). <http://doi.org/10.1145/1118890.1118892>
- Barišić, A., Goulão, M. y Amaral, V. (2015, Septiembre). Domain-Specific Language domain analysis and evaluation: a systematic literature review. Faculdade de Ciencias e Tecnologia, Universidade Nova da Lisboa. <http://dx.doi.org/10.5281/zenodo.265487>
- lung, A., Carbonell, J., Marchezan, L., Rodrigues, E., Bernandino, M., Basso, F.P. y Medeiros, B. (2020, Agosto 28). Systematic mapping Study on domain-specific language development tools. *Empirical Software Engineering*, 25. <https://doi.org/10.1007/s10664-020-09872-1>
- Van Deursen, A., Klint, P.T. y Visser, J. (2000, Enero). Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(06), 26-36. <https://doi.org/10.1145/352029.352035>
- Mauw, S., Wiersma, W.T. y Willemse, T.A.C. (2004, Junio 15). Language-Driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14(06), 625-663. <https://doi.org/10.1142/S0218194004001828>

- Moreno-Ger, P., Martínez-Ortiz, I., Sierra, J.L. y Fernández-Manjón, B. (2006, Enero). Language-Driven Development of Videogames: The e-Game Experience. *ACM Computing Surveys*, 37(4). En Harper, R., Rauterberg, M. y Combetto, M. (Eds.), *Entertainment Computing – ICEC 2006*. Springer. <https://doi.org/10.1007/11872320>
- Ward, M.P. (1994, Octubre) Language Oriented Programming. <http://www.gkc.org.uk/martin/papers/middle-out-t.pdf>
- Dmitriev, S. (2004, Noviembre) Language Oriented Programming: The Next Programming Paradigm. <https://resources.jetbrains.com/storage/products/mps/docs/Language Oriented Programming.pdf>
- Geisler, B.J., Mitropoulos, F.J. y Kavage, S. (2019, Abril). GAMESPECT: Aspect Oriented Programming for a Video Game Engine using Meta-languages. *2019 SoutheastCon*, pp. 1-8. <https://doi.org/10.1109/SoutheastCon42311.2019.9020369>
- Zahari, A.S., Ab Rahim, L., Nurhadi, N. y Aslam, M. (2020, Junio 13). A Domain Specific Modeling Language for Adventure Educational Games and Flow Theory. *International Journal on Advanced Science Engineering and Information Technology*, 10(3). <https://doi.org/10.18517/ijaseit.10.3.10173>
- Eclipse Foundation (2021). Xtext Documentation. Xtext. Recuperado el 28 de Agosto 2021 de <https://www.eclipse.org/Xtext/documentation>
- Fandom (2021). Unreal Script. *Unreal Wiki*. Recuperado el 22 de Agosto 2021 de <https://unreal.fandom.com/wiki/UnrealScript>
- Epic Games (2021a). Introduction to Blueprints. *Unreal Engine Documentation*. Recuperado el 28 de Agosto 2021 de <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/>
- Núñez-Valdez, E.R., Sanjuan Martínez, O., Pelayo García-Bustelo, B.C., Cueva-Lovelle, J.M. y Infante Fernández, G. (2013). Gade4all: Developing Multi-platform Videogames based on Domain Specific Languages and Model Driven Engineering. *Revista IJIMAI*, 2(2). <http://doi.org/10.9781/ijimai.2013.224>



- González García, C., Núñez-Valdez, E.R., Moreno-Ger, P., González Crespo, R., Pelayo García-Bustelo, B.C. y Cueva-Lovelle, J.M. (2019). Agile development of multiplatform educational video games using a Domain-Specific Language. *Universal Access in the Information Society*. <http://doi.org/10.1007/s10209-019-00681-y>
- Star Wars Jedi Knight II: Jedi Outcast (versión PC) [Videojuego]. (2002). Raven Software, LucastArts.
- Valve Software (2021, Agosto 24). Console Command List. *Valve Developer Community*. [https://developer.valvesoftware.com/wiki/Console\\_Command\\_List](https://developer.valvesoftware.com/wiki/Console_Command_List)
- Morais, D. K., Roesch, L.F.W., Redmile-Gordon, M., Santos, F.G., Baldrian, P., Andreote, F.D., Pylo, V.S. (2018, Julio 30). BTW – Bioinformatics Through Windows: an easy-to-install package to analyze marker gene data. *PeerJ*. <https://doi.org/10.7717/peerj.5299>
- Khronos Group (2021, Julio 26). OpenGL Shading Language. *OpenGL Wiki*. Recuperado el 26 de Julio 2021 de [https://www.khronos.org/opengl/wiki/OpenGL\\_Shading\\_Language](https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language)
- Hachisuka, T. (2015, Mayo 22). Implementing a Photorealistic Rendering System using GLSL. Cornell University. <https://arxiv.org/abs/1505.06022>
- Epic Games (2021b). Documentación Unreal Engine 4. Recuperado el 26 de Julio 2021 de <https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Materials/Editor/>
- Emil Drkusic (2020, Marzo 31). Learn SQL: SQL Query examples. *SQLShack.com*. Recuperado el 10 de Agosto 2021 de <https://www.sqlshack.com/learn-sql-sql-query-examples/>
- Microsoft Corporation (2017, Marzo 16). Control-of-Flow. *SQL Reference, Language Elements*. Recuperado el 10 de Agosto 2021 de <https://docs.microsoft.com/en-us/sql/t-sql/language-elements/control-of-flow?view=sql-server-ver15>
- Oracle Corporation (2016, Enero). SQL Language Reference. Recuperado el 10 de Agosto 2021 de [https://docs.oracle.com/cd/E11882\\_01/server.112/e41084.pdf](https://docs.oracle.com/cd/E11882_01/server.112/e41084.pdf)
- Unity Technologies (2021a). The UXML format. *Unity Manual*. Recuperado el 3 de Agosto 2021 de <https://docs.unity3d.com/Manual/UIE-UXML.html>
- Unity Technologies (2021b). Unity Style Sheets. *Unity Manual*. Recuperado el 3 de Agosto 2021 de <https://docs.unity3d.com/Manual/UIE-USS.html>

Troy Wolverton (2018, Septiembre 14). One of the leading companies in the video-game business is gunning to take over the enterprise software industry. *Business Insider*. <https://www.businessinsider.com/unity-technologies-sees-big-opportunities-outside-of-video-games-2018-9>

Microsoft Corporation (2021). C# Coding Conventions. *Microsoft Docs*. Recuperado el 10 de Septiembre de <https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>

Eclipse Foundation (2021). The Grammar Language. *Xtext Documentation*. Recuperado el 24 de Agosto 2021 de [https://www.eclipse.org/Xtext/documentation/301\\_grammarlanguage](https://www.eclipse.org/Xtext/documentation/301_grammarlanguage)

Unity Technologies (2021c). Scriptable Objects. *Unity Manual*. Recuperado el 2 de Septiembre 2021 de <https://docs.unity3d.com/Manual/class-ScriptableObject.html>

Unity Technologies (2021d). GameObjects. *Unity Manual*. Recuperado el 2 de Septiembre 2021 de <https://docs.unity3d.com/Manual/GameObjects.html>

Unity Technologies (2021d). TextMeshPro. *Unity Manual*. Recuperado el 4 de Septiembre 2021 de <https://docs.unity3d.com/Manual/com.unity.textmeshpro.html>

## Anexo A. Artículo

# Simplificando el desarrollo integrado de Lenguajes de Dominio Específico en un motor de videojuegos

Pablo González Martínez

Universidad Internacional de la Rioja, Logroño (España)

Fecha 22/09/2021



## PALABRAS CLAVE

Motores de videojuegos,  
Lenguajes de Dominio  
Específico, Scripting.

## RESUMEN

Los motores de videojuegos son un importante entorno de desarrollo software gracias a sus capacidades tecnológicas y creciente facilidad de uso, con cada vez más relevancia en sectores ajenos al entretenimiento. El uso de Lenguajes de Dominio Específico (DSL) es común en estas tecnologías, pero no se considera el proceso de desarrollo de nuevos DSL como una posible parte integrada en el desarrollo de aplicaciones como alternativa al *scripting* con lenguajes de propósito general o al desarrollo de herramientas para modificar los datos que dirigen la lógica. En este trabajo se desarrolla un paquete software integrado en un motor de videojuegos que permite simplificar el desarrollo de nuevos DSL de alto nivel, incluyendo un editor de código integrado y una prueba de uso con una aplicación desarrollada sobre el motor. El software desarrollado proporciona simplicidad y facilidad de uso para el desarrollo de DSLs, y permite contemplar este proceso como una alternativa al costoso desarrollo de herramientas dedicadas o al uso de lenguajes con menos especificidad, expresividad y adecuación al uso.

## I. INTRODUCCIÓN

Un motor de videojuegos comprende un elaborado conjunto tecnológico que permite la construcción de complejas simulaciones dinámicas ejecutadas en tiempo real. Aunque normalmente se crean y extienden con código fuente implementado en lenguajes de la familia C, de bajo nivel y compilados, aunque el uso de scripting para simplificar el desarrollo está extendido.

El proceso de desarrollo de esta lógica depende de los perfiles que se considere óptimo involucrar. La posibilidad de que usuarios con conocimientos reducidos de programación puedan modificar procesos lógicos o incluso crear estos procesos desde cero es posible gracias al uso de programación con scripts mediante lenguajes de más alto nivel de abstracción, contando además algunos motores con el uso de lenguajes de programación

visual. No obstante, el uso del motor por parte de estos perfiles de menor tecnicidad en los ámbitos de codificación está dominado por las implementaciones software denominadas herramientas; software integrado en el motor que está dedicado a capacitar la creación y modificación del contenido de las aplicaciones desarrolladas requiriendo de una previa implementación de lógica dirigida por datos, que serán el objeto de modificación por parte de dichas herramientas.

Los nuevos sectores de aplicabilidad de la tecnología de videojuegos, como industria o defensa, tienen algunos requisitos que implican necesidades extensivas de edición y configuración que a menudo deben ponerse a disposición del cliente final, para ser usados por usuarios de perfiles de diferente tecnicidad con conocimientos en dominios específicos asociados al objetivo funcional de las aplicaciones desarrolladas. Proporcionar acceso al motor usado para el desarrollo de la aplicación es una solución

deficiente, ya que tendrá inherentemente una sobrecarga de complejidad, requerirá del uso de un software genérico adicional de terceros si utilizamos un motor comercial generalista, no provee de un conjunto tecnológico integrado, adaptable y flexible con las necesidades concretas del proyecto, y además necesita de realizar de nuevo el proceso de construcción de la aplicación y su actualización en cada dispositivo para ver reflejado cualquier cambio.

El desarrollo de herramientas para permitir configurar y editar el software supone un coste de recursos elevado, especialmente cuando no se tienen las utilidades proporcionadas por el editor del motor de videojuegos debido a que estas capacidades deben estar integradas en las aplicaciones construidas.

Los Lenguajes de Dominio Específico (DSL) son lenguajes de programación adaptados a resolver problemas en un dominio de aplicación concreto en vez de proporcionar funcionalidades de propósito general, permitiendo mayor expresividad y facilidad de uso [1]. A lo largo de la historia de los videojuegos, se ha hecho uso de lenguajes de dominio específico para su desarrollo, algunos de ellos originados en el propio ámbito de los videojuegos y otros de uso generalizado para dominios comunes fuera de este.

Sin embargo, no se considera la creación y uso de nuevos DSL integrados como una parte regular del proceso de desarrollo de cada aplicación software, con el objetivo de aliviar las necesidades de desarrollo de herramientas para configuración, edición o lógica de alto nivel, que puede ser especialmente relevante cuando se deban adecuar a dominios que no tengan relación con el proceso de creación del software, sino con su objetivo funcional una vez construido. La complejidad del desarrollo de DSL, su necesidad de herramientas externas y soluciones de integración, y la existencia de requisitos que pueden imponer un alto nivel de especificidad y poca reusabilidad en las soluciones hacen que sea complicado para un equipo desarrollador optar por esta vía.

Este trabajo práctico propone desarrollar un software que simplifique e integre la capacidad de desarrollar Lenguajes de Dominio Específico textuales con un alto nivel de abstracción como parte del código de aplicación sobre un motor de videojuegos, sirviendo para especificar la configuración y edición del comportamiento lógico, con el objetivo de transformar el proceso de diseño y desarrollo de DSL en un proceso más sencillo, directamente integrado con el resto de código de aplicación, y accesible para cualquier programador de tecnología de videojuegos reduciendo al mínimo los requisitos de conocimiento para su uso. Se expone como una alternativa ocasional durante el ciclo de vida de desarrollo del software al desarrollo de herramientas dedicadas, al uso de lenguajes de scripting de propósito general, o al uso de DSL creados con herramientas externas, permitiendo abstraer la complejidad de los procesos y las implementaciones, facilitar el scripting de alto nivel con un alto grado de expresividad y especificidad, mejorar la accesibilidad al desarrollo integrado de lenguajes simples reduciendo los requisitos de conocimiento, y estimando que puede ser de especial utilidad para reducir los recursos de desarrollo necesarios para cumplir con los requisitos de los proyectos software para los sectores de aplicabilidad de la tecnología ajenos al entretenimiento, pudiendo mejorar sus tiempos, costes y tareas de prototipado o validación.

## II. ESTADO DEL ARTE

El uso de DSL está presente en la tecnología de videojuegos, siendo algunos de ellos ubicuos en prácticamente cualquier

desarrollo moderno, como los lenguajes de shading necesarios para los gráficos acelerados por hardware, otros de uso generalizado fuera del ámbito de los videojuegos e integrados en algunos proyectos para tareas concretas, y otros diseñados para facilitar el proceso del propio desarrollo en motores o entornos concretos con el objetivo principal de reducir la necesidad de programadores experimentados para la implementación de la lógica de un videojuego, incrementando la accesibilidad al desarrollo con lenguajes más sencillos y abstractos.

### *Estado del arte de Lenguajes de Dominio Específico*

En la literatura existente acerca de DSL se puede observar que la creación de un nuevo DSL no es una tarea trivial, y requiere del uso de múltiples herramientas dedicadas al proceso de desarrollo y mantenimiento de un lenguaje, incluyendo generadores de código, validadores, comprobadores de modelo y analizadores léxicos, semánticos y sintácticos, y la diversidad de estas herramientas hace que sea difícil establecer qué características son necesarias para construir un lenguaje en un contexto específico [2]. En [1] se indica la complejidad del desarrollo de DSL debido a que requiere conocimientos de dominio y desarrollo de lenguajes, incluye técnicas más variadas que las existentes para GPL, y cuyas tareas de soporte, estandarización y mantenimiento pueden volverse un problema al que asignar recursos significativos. La elección de desarrollar un DSL puede hacerse evidente únicamente después de haber realizado una inversión considerable en el desarrollo de un software para un dominio. Entre las desventajas de uso de DSL, en [3] se listan, entre otras, su coste de diseño, implementación y mantenimiento.

El enfoque conocido como dirigido por lenguajes (Language-Driven) está centrado en el diseño de DSL adaptado al sistema en construcción como parte del desarrollo, basándose en la obtención de una definición formal del problema a partir de un diseño de dominio, que se utiliza para diseñar su sintaxis, semántica y pragmática [4]. En [5] se muestra un ejemplo de este enfoque aplicado a videojuegos, pero se considera el dominio específico al propio desarrollo de juegos, especificando que la semántica operacional del lenguaje es la base del desarrollo del motor de videojuegos: «La actividad de implementación del lenguaje trata a su vez con la construcción de un motor de juego. Este motor estará basado en la descripción del lenguaje producida durante el diseño de lenguaje» (p. 5). Consideran el lenguaje construido como una herramienta de autoría de contenido para el desarrollo, y notan la necesidad de un enfoque incremental para permitir adaptación a diferentes géneros de videojuegos.

El término Programación Orientada a Lenguajes (LOP) es utilizado por múltiples personas para describir diferentes enfoques de programación, aunque todos ellos con puntos en común. [6], lo describe como basar el desarrollo de un programa en el desarrollo de un lenguaje especificado formalmente, orientado a dominio, de muy alto nivel, con dos fases independientes en el desarrollo, la implementación del sistema con el lenguaje y la implementación de un compilador, transpilador o interpretador para el lenguaje utilizando tecnología existente. [7] describe el LOP como la elección de uno o varios DSL existentes, o el desarrollo de uno o varios DSL si es necesario, para llevar a cabo el desarrollo de un programa, a partir del modelado conceptual realizado por el programador, indicando que es una manera mucho más efectiva de programar, ya que las soluciones a los problemas suelen modelarse de una manera no imperativa mediante instrucciones, y los lenguajes de propósito general requieren traducir conceptos de alto nivel en el dominio en características de programación de bajo nivel. En [8] desarrollan un DSL de alto nivel con características de programación orientada a aspectos (AOP) destinado a su uso

durante el desarrollo de videojuegos con el motor de videojuegos Unreal Engine 4, con el objetivo de reducir la duplicación de código además de proporcionar una plataforma AOP agnóstica del código de juego, pudiendo generar código C++, LUA, Blueprints o Skookum Script, para poder funcionar en el motor. El DSL es creado mediante Xtext, una herramienta que permite el desarrollo de lenguajes de programación y DSL. Xtext provee de un potente entorno con analizador sintáctico, enlazador, comprobador de tipos y compilador, con la creación del lenguaje estando definida en su propio DSL Grammar Language que permite describir sintaxis concreta y cómo está mapeada al modelo semántico.

### *Estado del arte de lenguajes integrados para videojuegos*

El scripting de aquella lógica de más alto nivel que carece de gran complejidad técnica y no tiene grandes necesidades de rendimiento o uso de memoria -estando estas características en el código fuente que la soporta a un nivel más bajo de abstracción-, es uno de los principales objetivos en la búsqueda de abstracción proporcionada por lenguajes de alto nivel, teniendo principalmente tres objetivos en el desarrollo con tecnología de videojuegos: capacitar la programación del comportamiento del software sin requerir una recompilación del código fuente, acelerar el proceso de desarrollo a través de eliminar complejidades de bajo nivel y mejorar la legibilidad de las implementaciones, y permitir a perfiles con menos conocimientos de programación participar en el desarrollo de esta lógica. Existen diversos lenguajes que permiten el scripting en motores de videojuegos, como LUA, y otros estando dedicados exclusivamente a un motor concreto, como UnrealScript. Otros ejemplos muestran el uso de DSL centrados en facilitar el proceso de desarrollo y hacerlo más accesible, aunque su uso está orientado a proporcionar modelos independientes a partir de los que generar el juego, mediante un DSL que, a través de un motor de transformación utilizando plantillas para cada plataforma objetivo, permite obtener el videojuego en código nativo [9][10]. La efectividad de los lenguajes textuales para facilitar el desarrollo por perfiles menos técnicos es a veces puesta en duda, con motores como Unreal Engine habiendo descartado UnrealScript por un sistema de programación visual denominado Blueprints [11].

Algunos videojuegos realizan implementaciones que permiten interpretar líneas de comandos textuales, comúnmente a través de una consola dedicada que forma parte de la aplicación construida, en algunos casos accesible para los usuarios finales. Las consolas de línea de comandos actúan esencialmente como un tipo adicional de input para el software, pero sin estar ligado a ninguna limitación física derivada del uso de periféricos, permitiendo una interpretación sin límites de una cadena de caracteres escrita. De cara a los usuarios finales, han estado tradicionalmente relacionados con dos usos principales: activar “trampas” que afectan al comportamiento del juego de una manera no intencionada por el diseño de juego, y permitir modificaciones adicionales de características del juego que no están presentes en los menús de interfaz de configuración, ya que los desarrolladores estiman que no tienen suficiente utilidad para el jugador medio como para justificar su inclusión en los menús. En caso de implementar comandos, no obstante, su objetivo principal será el de actuar como una pieza clave del proceso de desarrollo en sí, ya que se utilizarán para los procesos de pruebas manuales, siendo estos el objetivo de las “trampas” que podrán opcionalmente ponerse a disposición de los jugadores. El uso de líneas de comando como método para utilizar software es común en dominios específicos de alta complejidad, y en la literatura se pueden encontrar ejemplos de la limitación que suponen debido a la falta de familiaridad de sus usuarios finales con el uso

requerido. En [12] se destaca el entrenamiento intensivo requerido para utilizar de manera efectiva los interfaces de líneas de comandos en el ámbito de la bioinformática, debido en gran parte a ser desarrollados para UnixShell, y especifican al respecto que «la bioinformática ha sido una causa de ‘dolor de cabeza’ para muchos científicos, incluso aquellos que han crecido en la era de las computadoras».

Otros ejemplos de DSL usados en videojuegos son los lenguajes de shading, como GLSL, HLSL, MSL, PSSL así como variantes y derivados, usados para la programación de *shaders*, pequeños programas utilizados para el renderizado en GPU. Contienen una sintaxis similar a C/C++ incluyen abstracciones para facilitar la programación del dominio de los gráficos por computador. Son lenguajes con gran complejidad usados por perfiles expertos por las abstracciones proporcionadas, y cuando perfiles sin conocimientos de programación deben crear *shaders*, se suele recurrir a herramientas de programación visual que compilan a alguno de estos lenguajes. Lenguajes menos comunes son SQL para la gestión de bases de datos relacionales, cuyas características declarativas permiten una alta expresividad con código más fácilmente comprensible por cualquier usuario en comparación con otros lenguajes, y lenguajes de marcado y estilo, inspirados en los lenguajes de uso común para el desarrollo web como HTML y CSS, teniendo el motor Unity dos sendas implementaciones inspiradas en cada uno para el desarrollo de interfaces de usuario [13][14].

## III. OBJETIVOS Y METODOLOGÍA

### *Objetivo general*

Diseñar e implementar un conjunto software integrado en un motor de videojuegos que permita crear, mantener, editar y ejecutar Lenguajes de Dominio Específico textuales de alto nivel que afecten al comportamiento de la lógica de ejecución de las aplicaciones construidas sin necesitar herramientas externas, sirviendo como fuente de configuración y edición, priorizando la simplicidad y facilidad de uso de todas las funcionalidades proporcionadas.

### *Objetivos específicos*

Desarrollar un paquete software integrado en un motor de videojuegos que permita crear, modificar y ejecutar DSL como parte del código compilado de aplicación a través de scripts de cadenas de caracteres utilizando su API. Iterar para optimizar el proceso de creación y mantenimiento de los lenguajes desde un punto de vista de legibilidad, facilidad de uso e interoperabilidad con el resto de código de aplicación y adecuación a posibilitar la extensión del software fuera del alcance del trabajo con un enfoque de complejidad aditiva para asegurar su ajuste a los requisitos básicos de alto nivel. Desarrollar un paquete software sobre el anterior que proporcione un entorno de interfaz integrado en el motor, que permita gestionar, consultar y editar los scripts para los lenguajes creados. Desarrollar un paquete software para el motor, independiente de los anteriores, que muestre necesidades de requisitos asociados a los nuevos sectores de aplicabilidad de la tecnología de videojuegos y provea de un entorno donde comprobar las capacidades de integración del software desarrollado sobre implementaciones software de aplicación preexistentes.

### *Metodología*

El trabajo se realiza de manera iterativa e incremental, con una división en iteraciones a partir de la división del software a



desarrollar en conjuntos funcionales cohesivos. Cada iteración incluye fases de especificación de requisitos, diseño, implementación y pruebas. Debido a las necesidades del desarrollo propuesto, la integración se considera como una parte inherente de algunas iteraciones y no como una posible fase. Los requisitos de cada iteración son requisitos de más bajo nivel estipulados a partir de identificar un posible resultado funcional de las necesidades de desarrollo identificadas por los requisitos de alto nivel. Según las necesidades funcionales, cada iteración puede tener dependencia en una o más implementaciones del software en iteraciones anteriores, incluyendo dentro de sí las tareas de revisión, refactorización y extensión del código previo para que sea posible cumplir con sus objetivos funcionales y realizar mejoras observables que permitan su evolución de cara a una mejor adecuación a los requisitos no funcionales.

Debido a la necesidad de experimentar y prototipar para cumplir con los objetivos del trabajo, se considera que las fases de diseño, implementación y pruebas se adecúan a la posibilidad de tener sus propias iteraciones, pudiendo repetirse para mejorar la adecuación al objetivo funcional.

A partir de considerar los requisitos más fundamentales de alto nivel del trabajo, se denomina complejidad funcional aditiva al principio guiado por tres reglas que es aplicable a todos los ámbitos del desarrollo orientados al usuario, desde la división de iteraciones al diseño y detalles de implementación del software: la complejidad funcional es inversamente proporcional a la prioridad, capacitar el añadido de características más complejas de manera limpia, sencilla, cohesiva y desacoplada es más importante que el propio añadido de características, y el añadido de mayor complejidad nunca debe romper la adecuación a requisitos de simplicidad, legibilidad y facilidad de uso preexistentes. Debido a que el software desarrollado tiene en gran medida el objetivo de ser utilizado por otros programadores a través de su API, se utiliza como principio para los detalles de implementación la evaluación de los requisitos de conocimiento de interfaz. Se denomina requisitos de conocimiento de interfaz de un conjunto software a la necesidad de conocer información concreta sobre su interfaz público para llevar a cabo una tarea de programación utilizando dicho software.

#### IV. CONTRIBUCIÓN

Se desarrolla un paquete software integrado en el motor de videojuegos Unity, enteramente en lenguaje C# que permite el desarrollo integrado de Lenguajes de Dominio Específico. Su funcionamiento se basa en proporcionar simplicidad, facilidad de uso y total interoperabilidad con el resto de código de aplicación. Este paquete software se le ha denominado Acolyte, debido a la relevancia del uso de funciones de primera clase a través de delegados en su funcionamiento, los cuales están contruidos como parte del lenguaje C# y contienen una función Invoke() para su ejecución.

##### *Declaraciones, invocaciones y estado*

El proceso declarativo de Acolyte se fundamenta en especificar palabras como cadenas de caracteres enlazados a una función encapsulada en un delegado. Se crea una estructura en forma de árbol denominada Lexemas Declarativos Enlazables (LDE) que enlazan los denominados *decllexemas*, las unidades básicas de declaración para un lenguaje. En Acolyte existen tres tipos de decllexema: los literales, que encapsulan una invocación con parámetro string y permiten la abstracción de funciones a las que se envía un texto literal; las palabras clave (*keywords*) que son reconocidas mediante una cadena de caracteres token y tienen una invocación asociada, y los identificadores, a los que se asocia

una función que procesa un objeto de tipo concreto a partir del string de la palabra.

Para construir declaraciones se deben crear las instancias necesarias de cada decllexema y asociar las funciones deseadas. Las instancias creadas pueden después ser enlazadas, formando la estructura de árbol requerida (Figura 1). Es obligatorio que una palabra inicial de árbol sea de tipo palabra clave.

```
Printer printer = null;

void SetError() => printer = new ErrorPrinter();
void SetWarning() => printer = new WarningPrinter();
void Print(string message)
{
    printer.message = message;
    printer.Print();
}

var printError = new Keyword("error", SetError);
var printWarning = new Keyword("warning", SetWarning);
var message = new Literal(Print);

printError.Then(message);
printWarning.Then(message);

// Ejemplo: "error Hello-World!"
```

Fig. 1. Código ejemplo de construcción de declaración.

Este uso implica que las funciones deben utilizarse con consideraciones de estado, permitiendo utilizar variables afectadas por funciones invocadas con anterioridad, o hacer uso de instancias con clases utilizadas para gestionar la localidad de las funciones invocadas mediante un objeto con estado.

Los decllexemas permiten también el uso de una función *Tolerate()* para indicar un string subsecuente que será tolerado, permitiendo mejorar la expresividad de los lenguajes. Se utiliza como ejemplo la declaración “interact {objectId}”, que se puede transformar en “interact with {objectId}”.

##### *Lenguaje, Scripts y Decllexicon*

El código incluye encapsulaciones para crear un lenguaje y para manejar scripts como objetos referenciables en código. A través del denominado *Decllexicon*, se provee de un entorno con estado en el que crear los árboles declarativos, especificar las funciones y manejar de manera local los procesos creados por las invocaciones a dichas funciones. Un decllexicon se instanciará para cada lenguaje creado en cada script de dicho lenguaje, permitiendo aislar la localidad de invocaciones de función y variables por cada uno. Por tanto, para un lenguaje creado con Acolyte, existirán  $n$  número de scripts y un equivalente  $n$  número de decllexicons y por tanto árboles de declaración instanciados en función del decllexicon para dicho lenguaje.

##### *Compilación y ejecución de scripts*

Aunque no se priorizan necesidades de optimización de rendimiento y uso de memoria, Acolyte implementa un compilador que procesa un código línea a línea y produce un array de instrucciones. Las instrucciones están implementadas como una clase heredable, que permiten indicar qué debe ocurrir al ejecutar un script de manera ordenada en un array. La instrucción básica simplemente contiene un delegado que encapsula una función que haya sido procesada por el compilador al encontrar un decllexema válido en su análisis sintáctico. Un objeto script en Acolyte es ejecutado tras compilarse, permitiendo a cualquier parte del código hacer uso de una función *Execute()* simplemente a través de tener una referencia al script, y se permite el envío de un objeto de contexto y un delegado de callback para mayor flexibilidad de uso.

## Sentencias y expresiones

Siguiendo el principio de complejidad aditiva, se incluye el procesamiento de sentencias imperativas mediante la clase *Statement*, ejemplificadas con una sentencia IF/ELSE. Para ello, Acolyte provee de una instrucción de salto, que puede modificar el índice de ejecución del array de instrucciones, y otra instrucción derivada de salto con condición, que realizará el salto indicado en función del resultado de una invocación a un delegado que resuelve a un booleano. El compilador procesará los *Statements* existentes permitiendo añadir las instrucciones pertinentes según su análisis sintáctico, teniendo los tokens de *Statement* preferencia sobre las palabras clave declarativas. El añadido de *Statements* requiere de una sola línea de código donde se indica la instancia y se especifican los tokens (Figura 2).

```
AddStatements(new IfElse("if", "else", "endif"));
```

Fig. 2. Código de creación de Statement IfElse.

Adicionalmente, se realiza una implementación simple de expresiones, las cuales relacionan un string concreto a un delegado que resuelve a un tipo, permitiendo a un lenguaje especificar strings que pueden ser utilizados como expresión para sentencias imperativas, como es el caso de la comparación IF/ELSE. Como se observa en la figura 3, se pueden añadir expresiones a un lenguaje, e indicar en un script “if profile is expert”, y la comprobación condicional llamará a la función indicada para establecer si debe procesar el salto de la instrucción añadida por el “if”.

```
AddExpression("profile is expert", IsProfileExpert);
```

Fig. 3. Código de creación de Expression que resuelve a bool.

## Implementación en Unity

Aunque el núcleo del software desarrollado está totalmente desacoplado del motor Unity, se proveen algunas implementaciones orientadas a su uso en el motor. Para el uso de scripts en Unity, se encapsula una referencia dentro de una clase *ScriptAsset* que permite crear assets en el editor, facilitando el manejo y referenciado de scripts. Adicionalmente, se crea un contenedor de objetos con tipo genérico identificables mediante string que permite utilizar un componente en escena para serializar objetos que puedan posteriormente actuar como contenedor de consulta en un declexema identificador. En la figura 4 se observa un ejemplo en código que permite identificar objetos del tipo de Unity *GameObject* en forma de array, permitiendo que se invoque la función *SelectObjects()* con el array de objetos que hayan sido serializados en escena usando el identificador que se haya especificado en el script. Se podría por tanto escribir “destroy {objectsId}” para destruir los objetos.

```
var setCloning = new Keyword("clone", SetCloning);  
var setDestruction = new Keyword("destroy", SetDestruction);  
var selectObj = new Identifier<GameObject[]>(SelectObjects,  
() => { return gameObjectsContainer; });  
  
setCloning.Then(selectObj);  
setDestruction.Then(selectObj);
```

Fig. 4. Código de declaración usando un identificador a GameObject.

## Editor de texto integrado en Unity

Acolyte incluye un editor de texto integrado en Unity, que es accesible tanto desde el modo Play de su editor como desde cualquier aplicación construida. A través de un análisis sintáctico, se identifican los tipos de palabras existentes y se colorean de acuerdo a un asset de configuración, permitiendo a cualquier

usuario identificar las palabras válidas, su tipo, y los errores en la escritura (Figura 5).

```
1 if profile is expert  
2 interact with AssemblyParts2 using herramienta  
3 abcdefg  
4 endif
```

Fig. 5. Captura de edición de texto en editor de Acolyte.

Adicionalmente, se calcula la palabra seleccionada según el símbolo de intercalación que marca la edición activa sobre el texto, y otro proceso de análisis para la línea en la que se está editando procesa los datos del lenguaje para obtener información contextual, creando una lista con todas las palabras que pueden ser incluidas para la selección:

Si es índice 0:

- Indicación línea ignorada por comentario (<Comment>).
- Lista de posibles token de Statements.
- Lista de posibles palabras clave iniciales del declexicon.

En índices posteriores, en función del análisis de las palabras anteriores:

- Lista de posibles Expressions, sólo si se ha indicado tipo en su definición.
- Lista de posibles palabras clave subsecuentes.
- Lista de posibles identificadores de objeto según el tipo del genérico del identificador.
- Indicación de posibilidad de introducir un literal (<Literal>).
- Indicación de palabra tolerada, a partir del declexema anterior (<Tolerated>).

Cualquier otro resultado aparecerá con una entrada <Unrecognized>.

La figura 6 muestra tres ejemplos de contextualización para una selección concreta. La selección mostrada en la figura 5 correspondería con el tercer ejemplo.

Statements (3)	Expression: boolean	Object identifiers (3)
if	profile is expert	Machine
else	booleanExample	AssemblyParts1
endif	true	AssemblyParts2
Keywords (4)	false	
set		
place		
interact		
debug		

Fig. 6. Captura de contextualización en editor de Acolyte.

## Prueba de uso

Para demostrar las capacidades de Acolyte se realiza un desarrollo de un paquete software denominado *ProcedUnity* que permite crear procedimientos paso a paso proporcionando utilidades abstractas. Su funcionamiento se basa en un procedimiento que contiene un array de pasos, que a su vez contiene un array de acciones. Las acciones deben ser implementadas según las necesidades de cada proyecto, y dependen de un asset de datos para determinar su comportamiento. Cuando un procedimiento es ejecutado, se consultan por cada paso sus acciones serializadas en el editor y se ejecuta la lógica a partir de sus datos. Sobre este software se crea una sencilla demo que consiste en realizar interacciones sobre subobjetos de un modelo simplificado de robot industrial



utilizando opcionalmente una herramienta seleccionada en interfaz.

Se realiza una implementación de Acolyte sobre la demo que permite sustituir el uso de acciones serializadas en el editor por scripts ejecutados cuyo procesamiento devuelve un array de acciones procesables por la ejecución de un paso, obteniendo el mismo resultado a través de una declaración. Se implementa un declexicon que incluye un sencillo árbol de declexemas que permite escribir “interact with {objectId} using {toolId}”, permitiendo especificar los datos equivalentes a una acción de interacción. Adicionalmente, se implementan declaraciones que permiten ejecutar funciones al comenzar un paso de procedimiento, demostrando la capacidad de extender de manera significativa de manera muy sencilla las funcionalidades ofrecidas por el software previo con capacidad inherente de edición sobre las mismas, incluyendo declaraciones como “set camera target {targetId}” para establecer el punto de órbita de la cámara usada en la demo, “set property {propertyId} in {entity} as {value}” para cambiar propiedades en las entidades usadas en los procedimientos, o “when {eventId} alert {value}”, incluyendo una implementación que muestra una alerta ante la invocación de un evento, mostrando la capacidad de procesar lógica fuera del momento de ejecución del propio script.

## V. RESULTADOS

El software desarrollado se encuentra integrado en el motor de videojuegos Unity sin ningún tipo de dependencia adicional. La base fundamental del código está desacoplada del motor, exponiendo un diseño independiente y reutilizable en otros entornos con dependencias únicamente en el lenguaje utilizado y las librerías básicas del entorno .NET para C#. El editor muestra dependencias inherentes en múltiples componentes del motor, pero también contiene parte de lógica desacoplada para los análisis sintácticos y la información de renderizado, pudiendo adaptarse con facilidad a otros entornos de desarrollo o a otros sistemas de interfaz dentro de Unity. La inclusión del software en cualquier proyecto realizado con Unity es extremadamente sencilla, requiriendo únicamente de añadir las carpetas con el contenido de Acolyte y su editor en algún sitio del proyecto para que este sea compilado automáticamente y esté disponible para el resto del código con total interoperabilidad.

El diseño de Acolyte está centrado en simplificar el proceso de desarrollo de un nuevo lenguaje. Acolyte permite realizar este desarrollo requiriendo únicamente conocimientos en los siguientes dominios: conocimientos del lenguaje C#, conocimientos básicos acerca del funcionamiento del motor Unity, y conocimientos de la API de Acolyte.

Se estima que cualquier desarrollador que conozca un lenguaje como C# y tenga conocimientos de desarrollo en tecnologías de videojuegos podrá comprender con facilidad las características ofrecidas por su API y conceptos como Script, Statements o Expressions.

El uso mínimo funcional de Acolyte, consistente en la creación de declaraciones mediante el declexicon y sus declexemas implica conocer un total de 7 clases: Script, Language, Declexicon, Keyword, Literal, Identifier y el conocimiento del interfaz IdentifierContainer.

El proceso de ejecución para un nuevo lenguaje requiere únicamente de referenciar una instancia de Script a través de un ScriptAsset y hacer uso de su función Execute(...). El diseño del declexicon implica que un programador es libre de utilizar el contexto y el tipo de callback como crea conveniente.

La implementación de árboles de declexemas utilizada en la prueba de integración contiene el siguiente número total de líneas de código, contando espacios e indentación estándar:

- Declaraciones base de Acolyte sobre ProcedUnity: 144 líneas.
- Declaraciones para creación de acciones de interacción en demo: 58 líneas.
- La generalización de Declexicon implementada contiene en total adicionalmente:
- 6 declaraciones de variables (para gestión interna de estado).
- 11 funciones de asignación a declexemas.
- 4 funciones de inicialización (2 + 2 debido a modularización de la clase).
- 5 funciones de tipo get para facilitar la obtención de contenedores.

Estos números demuestran que realizar una implementación de Declexicon es una tarea fácilmente abordable, donde la mayoría de la implementación consiste en organizar las variables y funciones que permiten construir los árboles de declexemas, teniendo en gran medida dependencia en las preferencias y conocimientos de los desarrolladores y no en una imposición de Acolyte que requiera conocimientos adicionales.

El propio desarrollo de Acolyte y su uso proveen de un diseño adaptado a añadir complejidad de manera aditiva a través de características como las instrucciones en forma de objetos generalizables, o la existencia de Statements y Expressions. Usar estos apartados del software y añadir nuevos es una tarea relativamente sencilla, aunque las implementaciones actuales son demasiado simples como para demostrar usos más complejos, como podrían ser expresiones con un análisis sintáctico de su contenido y manejo de operadores lógico-matemáticos.

## VI. DISCUSIÓN

Acolyte propone un diseño que facilita libertad y flexibilidad en su uso por parte de un equipo desarrollador que pretenda desarrollar un lenguaje: Acolyte no fuerza ningún patrón concreto para la implementación de la lógica declarativa, otorgando libertad flexible con el uso del objeto con estado Declexicon. Garantiza interoperabilidad total con el resto de código gracias al uso de C# compilado en Unity. Herramientas como los identificadores y el interfaz de contenedor de identificación permiten realizar implementaciones muy flexibles para identificar todo tipo de objetos serializados según las preferencias de cada implementación. La creación de un lenguaje se permite de manera totalmente modular a lo largo de varios paquetes software, incluyendo declexemas modulares a través de clases parciales, así como el añadido modular de Statements y Expressions a un lenguaje. La implementación actual de Expressions es extremadamente limitada, careciendo de ningún tipo de análisis sintáctico de una línea de código. Los análisis sintácticos se realizan exclusivamente línea a línea y la separación de palabras se hace mediante un carácter concreto indicado en los datos de lenguaje. La falta de definición de alcances como alcances para un Statement, alcances para procesar el contenido dentro de corchetes o paréntesis, o alcances para procesar literales en función de caracteres de principio y fin, es un problema para un uso más efectivo de Acolyte. En este trabajo se ejemplifican diferentes posibles usos dentro del declexicon, como variables locales al objeto, clases anidadas, uso de singletons o variables públicas estáticas. La flexibilidad ofrecida por el diseño del declexicon es de gran utilidad para adecuar la implementación a las preferencias y necesidades de cada desarrollador o proyecto,

pero puede resultar confuso cómo realizar este proceso, careciendo de ningún tipo de patrón explícito o ayuda que lo guíe.

## VII. CONCLUSIONES

Acolyte demuestra la posibilidad de simplificar el desarrollo de Lenguajes de Dominio Específico dentro de un motor de videojuegos sin la necesidad de utilizar herramientas externas ni conocimientos adicionales, facilitando su uso por desarrolladores expertos en tecnología de videojuegos. Se cumplen requisitos de simplicidad, facilidad de uso, flexibilidad, modularización y se demuestra su potencial con la creación del lenguaje simple StepScript en una demo sobre un paquete software independiente.

Acolyte provee de capacidades de creación de declaraciones y del añadido de complejidad aditiva, demostrando que la implementación actual tiene gran potencial de extensión, incluyendo características imperativas.

A través de su editor integrado, la creación y modificación de scripts para los lenguajes creados es una tarea intuitiva y guiada, haciendo uso de análisis sintáctico para colorear palabras y proporcionar un contexto indicando las posibilidades de escritura al usuario, mejorando la experiencia de uso de perfiles de reducida tecnicidad que tengan conocimientos de un dominio específico ajeno a la programación.

Mediante estas características, se estima que Acolyte y el desarrollo y uso de DSLs son viables como una parte integrada del desarrollo de software utilizando motores de videojuegos con el propósito de sustituir algunas de las necesidades de desarrollo orientadas a herramientas dedicadas, así como a la necesidad de utilizar lenguajes menos específicos y menos expresivos para programar lógica de comportamiento que no deba ser recompilada ni requiera una reconstrucción del software. Se considera especialmente óptimo para reducir los recursos necesarios en facilitar capacidades de configuración o edición fuera del editor de motor sobre las aplicaciones ya construidas.

Sin embargo la implementación actual de Acolyte muestra carencias en diferentes áreas importantes para facilitar su efectividad y mejorar la expresividad de los lenguajes creados, como posibilitar la definición de alcances y la claridad de uso de su proceso de implementación de declaraciones. También existe una falta de características en su editor que faciliten su uso como alternativa a herramientas con un diseño de UX dedicado, como mejoras en la contextualización y la posibilidad de añadir información sobre el lenguaje para guiar en mayor medida la escritura de scripts.

La prueba de integración de uso es prometedora, habiendo cumplido los requisitos establecidos, pero el resultado no se considera suficiente para demostrar la efectividad de Acolyte en proyectos de mayor envergadura y complejidad requiriendo una implementación de mayor tamaño. Más pruebas con el uso de Acolyte son necesarias, así como pruebas de uso de usuarios finales sobre lenguajes creados en proyectos destinados a dominios específicos.

## REFERENCIAS

- [1] Mernik, M., Heering, J. y Sloane, A.M. (2005, Diciembre 1). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4). <http://doi.org/10.1145/1118890.1118892>
- [2] Jung, A., Carbonell, J., Marchezan, L., Rodrigues, E., Bernardino, M., Basso, F.P. y Medeiros, B. (2020, Agosto 28). Systematic mapping Study on domain-specific language development tools. *Empirical Software Engineering*, 25.

- <https://doi.org/10.1007/s10664-020-09872-1>
- [3] Van Deursen, A., Klint, P.T. y Visser, J. (2000, Enero). Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(06), 26-36. <https://doi.org/10.1145/352029.352035>
- [4] Mauw, S., Wiersma, W.T. y Willemse, T.A.C. (2004, Junio 15). Language-Driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14(06), 625-663. <https://doi.org/10.1142/S0218194004001828>
- [5] Moreno-Ger, P., Martínez-Ortiz, I., Sierra, J.L. y Fernández-Manjón, B. (2006, Enero). Language-Driven Development of Videogames: The e-Game Experience. *ACM Computing Surveys*, 37(4). En Harper, R., Rauterberg, M. y Combetto, M. (Eds.), *Entertainment Computing – ICEC 2006*. Springer. <https://doi.org/10.1007/11872320>
- [6] Ward, M.P. (1994, Octubre) *Language Oriented Programming*. <http://www.gkc.org.uk/martin/papers/middle-out-t.pdf>
- [7] Dmitriev, S. (2004, Noviembre) *Language Oriented Programming: The Next Programming Paradigm*. [https://resources.jetbrains.com/storage/products/mps/docs/Language\\_Oriented\\_Programming.pdf](https://resources.jetbrains.com/storage/products/mps/docs/Language_Oriented_Programming.pdf)
- [8] Geisler, B.J., Mitropoulos, F.J. y Kavage, S. (2019, Abril). GAMESPECT: Aspect Oriented Programming for a Video Game Engine using Meta-languages. *2019 SoutheastCon*, pp. 1-8. <https://doi.org/10.1109/SoutheastCon42311.2019.9020369>
- [9] Núñez-Valdez, E.R., Sanjuan Martínez, O., Pelayo García-Bustelo, B.C., Cueva-Lovelle, J.M. y Infante Fernández, G. (2013). Gade4all: Developing Multi-platform Videogames based on Domain Specific Languages and Model Driven Engineering. *Revista IJIMAI*, 2(2). <http://doi.org/10.9781/ijimai.2013.224>
- [10] González García, C., Núñez-Valdez, E.R., Moreno-Ger, P., González Crespo, R., Pelayo García-Bustelo, B.C. y Cueva-Lovelle, J.M. (2019). Agile development of multiplatform educational video games using a Domain-Specific Language. *Universal Access in the Information Society*. <http://doi.org/10.1007/s10209-019-00681-y>
- [11] Epic Games (2021a). Introduction to Blueprints. *Unreal Engine Documentation*. <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/>
- [12] Morais, D. K., Roesch, L.F.W., Redmile-Gordon, M., Santos, F.G., Baldrian, P., Andreote, F.D., Pylro, V.S. (2018, Julio 30). BTW – Bioinformatics Through Windows: an easy-to-install package to analyze marker gene data. *PeerJ*. <https://doi.org/10.7717/peerj.5299>
- [13] Unity Technologies (2021a). The UXML format. *Unity Manual*. <https://docs.unity3d.com/Manual/UIE-UXML.html>
- [14] Unity Technologies (2021b). Unity Style Sheets. *Unity Manual*. <https://docs.unity3d.com/Manual/UIE-USS.html>

## Anexo B. Enlace a repositorio

**Enlace a repositorio con proyecto completo y código fuente**

<https://github.com/pablogozmaz/unity-acolyte>