INTRODUCCIÓN A PATRONES DE DISEÑO DE SOFTWARE PARA EL DESARROLLO DE VIDEOJUEGOS

Hans Schaa, Felipe Besoaín & Nicolas A. Barriga, Enero 2020. // Universidad de Talca
Escuela de Ingeniería en Desarrollo de Videojuegos y Realidad Virtual.

Resumen

Los patrones de diseño orientados a objetos son una metodología reconocida para reutilizar soluciones a problemas comunes. Sin embargo, hay muy poca bibliografía, guías y ejemplos para utilizarlos en el contexto de desarrollo de videojuegos. En este artículo, explicaremos algunos de los patrones más útiles para videojuegos, y mostraremos en detalle como pasar desde el patrón al diseño y luego a la implementación. El código fuente completo está disponible. Dada la falta de ejemplos y guías en motores modernos, creemos que éste puede ser un buen punto de partida para nuevos desarrolladores de videojuegos.

Palabras clave

Patrones, diseño, videojuegos, motores.

Abstract

Object oriented design patterns are a well known methodology for reusing common problem's solutions. However, there is very little bibliography, guidance and examples for using them in the context of videogame development. In this article, we will explain a few of the most useful design patterns for videogames, and show in detail how to go from pattern to design to implementation. Full source code is provided. Given the lack of examples and guides in modern game engines, we believe this can be a good starting point for new game developers.

Keywords

Patterns, design, video games, engines.

Introducción

El desarrollo de videojuegos ha tenido un crecimiento exponencial en los últimos años debido a muchos factores como el cambio de modelo de negocios en las empresas desarrolladoras de game engines, penetración de tecnologías y bajo costo de hardware mínimo requerido para desarrollo de videojuegos. Además, se ha facilitado el acceso a software como *Unity, CryEngine, Unreal, Game Maker y Godot* entre otros. Este hecho ha generado interés en la comunidad de desarrolladores atrayendo cada vez más personas al desarrollo de aplicaciones interactivas. *Unity* posee una baja curva de aprendizaje en comparación a *Unreal* y *CryEngine* lo que permite que personas de todas las edades y con pocos conocimientos sobre desarrollo de videojuegos puedan crear su primer proyecto interactivo. Con *Unity* se han creado miles de videojuegos, algunos con la capacidad de generar una fortuna, como es el caso de *Cuphead, Ori and the Blind Forest* y *Hearthstone*.

Uno de los géneros más comunes es el género de videojuegos de plataforma como *Donkey Kong* (1981) o *Rayman Adventures* (2015). Existe una gama muy amplia de videojuegos de este tipo, los que se caracterizan por la presencia de variados obstáculos que explotan las mecánicas de movimiento tales como salto, agacharse o algo tan simple como la espera frente a eventuales enemigos u objetos móviles. A medida que pasan los años, el desarrollo de videojuegos de plataforma ha experimentado múltiples modificaciones. Ha habido fusiones con otros géneros, y cambios en los conceptos detrás del diseño, lo que genera nuevos desafíos, dinámicas de juego, curvas de dificultad y flow (Csikszentmihalyi, 2009). Con el avance de la tecnología se han podido diseñar entidades que tienen comportamientos más complejos y variados, con controladores cada vez más sofisticados y fieles a la intención del diseñador (Sayed, Sinha, & Reith, 2017).

La implementación de *Inteligencia Artificial* (IA) de enemigos fácilmente puede desencadenar en código ilegible, difícil de mantener y con poca flexibilidad (Abbes, Khomh, Gueheneuc, & Antoniol, 2011). En la búsqueda de material bibliográfico para la creación de comportamientos de NPC[1] nos encontramos con poca información que nos ayude a escribir código eficiente y elegante, que asegure una fácil mantención a medida que el proyecto crezca, generando así software robusto y modular. Como si el hecho de la poca información no fuera suficiente, en algunas de estas implementaciones (por ejemplo (Aversa, Kyaw, & Peters, 2018)) no se plantean soluciones eficientes. También accedimos a la *Unity Store* para analizar las implementaciones gratuitas de *Finite-State Machines* y *Command Pattern*, en ella encontramos algunos assets que implementan el patrón estado, poco mantenibles o como se detalla en el capítulo 4 difíciles de expandir.

En la siguiente sección describiremos qué es un patrón de diseño y cómo pueden ser una solución para refactorizar nuestro *código spaguetti*[2] a la hora de crear controladores para entidades de nuestro videojuego. Luego, en el capítulo 3 haremos un estudio de caso en lenguaje C# para el motor *Unity*[3] dando a conocer cómo podemos evitar antipatrones en la codificación de controladores para nuestro videojuego. En la sección IV presentamos las mejoras propuestas a la implementación del libro (Aversa et al., 2018) y los assets[4] de la *Unity Store* que tienen implementaciones de los patrones Comando y Estado, sus ventajas y desventajas. Finalizamos con un resumen de lo presentado anteriormente e indicamos posibles direcciones de investigación futura.

Trasfondo

Programar software a gran escala es una tarea que conlleva muchas horas de trabajo humano. Para hacer más fácil abordar este desafío, a menudo el software se divide lógicamente en subtareas que son diseñadas, programadas y probadas de forma autónoma por distintos grupos de desarrolladores (Ampatzoglou & Chatzigeorgiou, 2007). Uno de los paradigmas que más se utilizan en la actualidad es la programación orientada a objetos, sin embargo, en la práctica los ingenieros de software tienen que lidiar con problemas frecuentes a la hora de diseñar aplicaciones que no puedan resolverse mediante los métodos actuales y herramientas disponibles. Como respuesta a estos problemas se han elaborado distintos patrones de diseño para desarrollar software modular y con bajo acoplamiento (Zimmer et al.,1995).

[1] *del inglés: Non-Player Character*
[2] *El código spaguetti es un término peyorativo para los programas de computación que tienen una estructura de control de flujo compleja e incomprensible.*
[3] *https://unity.com*
[4] *Elementos externos que pueden ser utilizados en un proyecto, como sonidos, imágenes o clases.*

Patrones de Diseño

Los patrones de diseño son microarquitecturas o módulos de construcción de alto nivel. Un programa construido bajo esta arquitectura es probable que exhiba buenas propiedades como modularidad, separación de tareas y mantenibilidad (Antoniol, Fiutem, & Cristoforetti, 1998). La idea principal detrás de los patrones de diseño es apoyar la reutilización de la información de diseño, lo que permite a los desarrolladores plantear bases sólidas en la arquitectura del *software* a construir, minimizando la probabilidad de fracaso del desarrollo del proyecto. Un patrón de diseño nombra, extrae e identifica los aspectos clave de una estructura de diseño común, lo que lo hace útil para crear un diseño reutilizable orientado a objetos (Gamma, Helm, Johnson, & Vlissides, 1994). Usualmente se agrupan en:

- Creacionales: Se refieren al proceso de creación de objetos.
- Estructurales: Se ocupan de la composición de clases u objetos.
- Comportamiento: Caracterizan las formas en que las clases u objetos interactúan y distribuyen la responsabilidad.

En desarrollo de videojuegos usamos algunos patrones con más frecuencia que otros (Freeman-Hargis, 2006; Nystrom, 2014). A continuación, mostraremos una reseña de dos patrones, en los cuales nos enfocaremos en el resto de este artículo.

Patrón Estado: El comportamiento dinámico es vital en los sistemas de *Inteligencia Artificial* para NPCs. Este patrón nos brinda una forma de modificar el comportamiento de una entidad cuando su estado interno ha cambiado. Si bien hay distintas implementaciones de este patrón, todas tienen elementos en común, como los estados, y las transiciones para cambiar entre ellos dependiendo de las condiciones del contexto.

Patrón Comando: Enfocado en encapsular tareas, los comandos son a menudo operaciones simples que pueden ponerse en cola y ser llamados para ejecutarse en una secuencia rápida. A menudo se pueden mejorar con métodos que vuelvan a un estado deseado el contexto del videojuego para ejecutar eventualmente el comando.

Arquitectura de un motor de videojuego

Un *Game Engine* está compuesto por un conjunto de módulos, que se controlan por medio de un proceso de administración principal. Asimismo, provee de un editor, el cual se utiliza para crear la experiencia de juego deseada (Gregory, 2017). Estos módulos a menudo se les llama subsistemas, algunos de ellos son:

- *Graphics*.
- *Dynamics*.
- *Audio*.
- *Input System*.

La figura 1 muestra una de las arquitecturas más comunes en el desarrollo de motores de videojuegos, donde los óvalos de contorno sólido son subsistemas esenciales y los punteados representan módulos que se pueden encontrar en juegos más complejos (Qu, Song, & Wei, 2013).

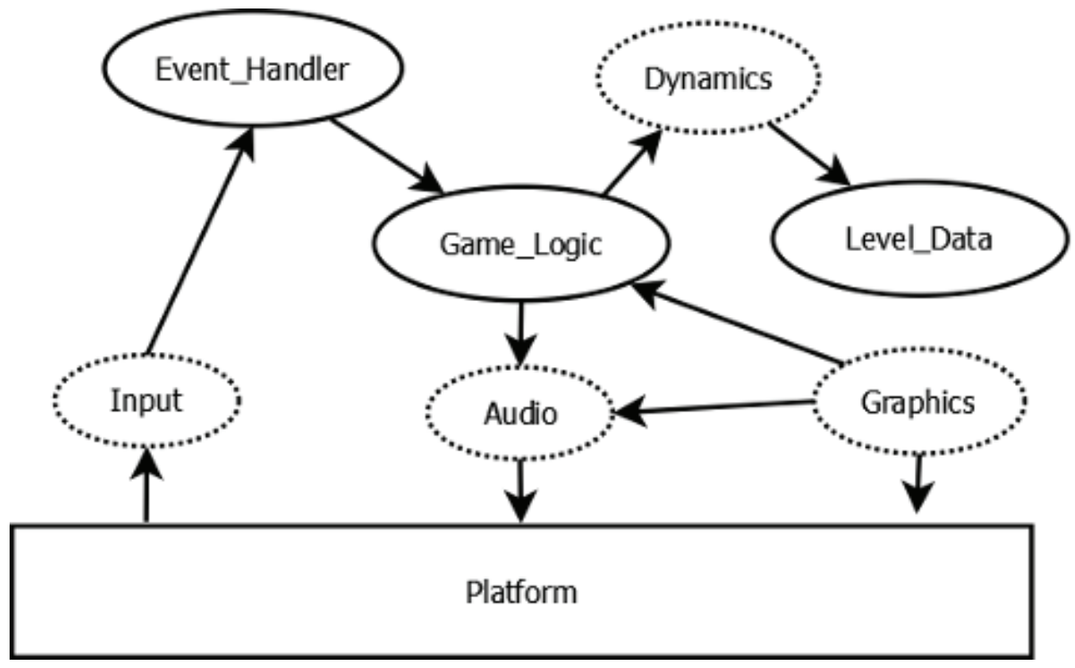

Fig. 01. *Common Game Architecture*

Unity es uno de los motores de videojuegos multiplataforma más utilizado (Toftedahl, 2019). Actualmente el motor soporta más de 25 plataformas, se puede usar para realizar aplicaciones de realidad aumentada, virtual y mixta. Este motor ha sido adoptado por industrias ajenas al videojuego, como cine, arquitectura, ingeniería y construcción, demostrando que es un software versátil. *Unity* es de código cerrado, por lo que no tenemos acceso directo a la arquitectura usada en su construcción. No obstante, por medio del flujo de trabajo propuesto por el mismo motor de videojuegos (Baron, 2019), podemos distinguir dos elementos claves:

- Componentes.
- API 4 de secuencias de comandos.

Unity está basado en componentes y por medio de éstos se construye un videojuego. En la figura 2 podemos encontrar una jerarquía de alto nivel, donde se aprecia una arquitectura en la cual los componentes pueden estar compuestos por otros componentes. Dentro del flujo de trabajo de *Unity* podemos encontrar las siguientes etapas:

- Importado de *Assets* (archivos como texturas, sonidos, música, modelos 3D, etc.).
- *Drag and drop* de *Assets* sobre la escena de Juego.
- Programar *behaviors*, sistemas o *managers*.
- Agregar componentes sobre entidades (*scripts, physics components, renderers*, etc.).

Para el desarrollo de este documento nos centraremos en la etapa de programación de behaviors, mediante el uso de patrones del tipo comportamiento.

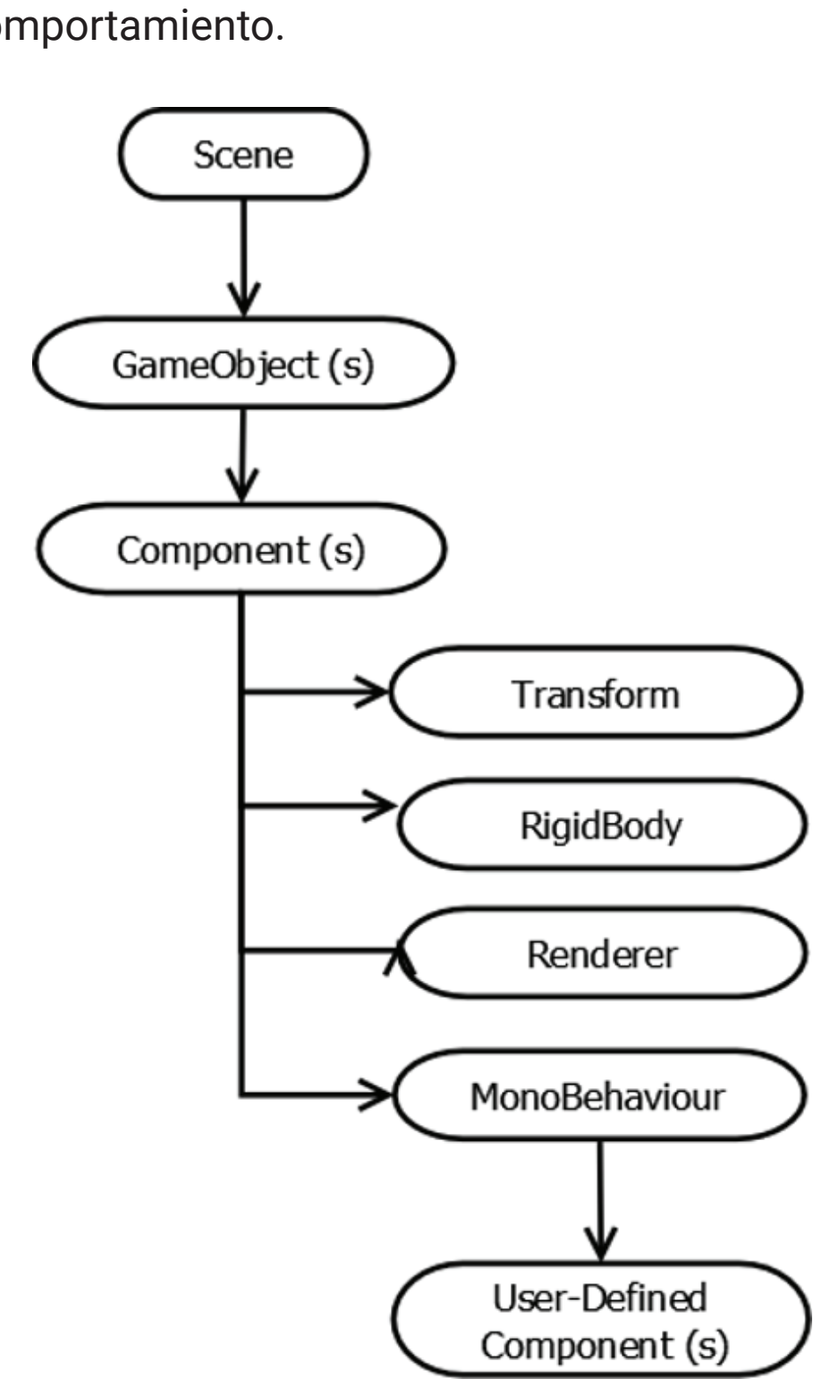

Fig. 02. *Unity Components Diagram*

Patrones de Diseño de Videojuegos

Algunos autores (Nystrom, 2014; Baron, 2019) han presentado nuevos patrones de diseño aplicables a problemas específicos del desarrollo de videojuegos.

Game Loop: En general, el código de un videojuego está estructurado en un ciclo de juego, donde por cada iteración procesamos entradas de usuario, por otra parte, el contexto de juego cambia según las lógicas internas hasta que esta iteración termina (Qu et al., 2013). La representación y la forma en que se procesa la lógica de juego se pueden codificar con subprocesos de eventos, dando como resultado un código más simple. En proyectos a pequeña escala o basados en turnos, con poca o ninguna animación, este enfoque funciona perfectamente. Como se puede apreciar en la figura 3, el patrón *Game Loop* describe el ciclo que corre continuamente durante el tiempo de juego, ejecutando las siguientes funciones:

o *Process Input:* Maneja cualquier entrada del usuario que haya ocurrido desde la última llamada.

o *Update Game:* Avanza la simulación del juego un paso.

o *Render:* Dibuja el juego para que el jugador pueda ver lo que sucedió.

Una buena implementación de este patrón permitirá la actualización de los objetos del mundo a una velocidad similar sin importar el *hardware* donde se esté ejecutando el videojuego (Nystrom, 2014).

Update Method: La función de este patrón es simular una colección de objetos independientes diciéndoles a cada uno que procese una serie de instrucciones a la vez. De esta manera cada entidad en el juego debe encapsular su propio comportamiento. Esto mantendrá el ciclo de juego ordenado y facilitará la adición y eliminación de entidades. Una vez por *frame*, el *game loop* recorre la colección y llama al método Update de cada objeto. Esto le da a cada uno la oportunidad de realizar el comportamiento de un frame. Al llamarlo en todos los objetos por cada frame, se comportarán de forma simultánea.

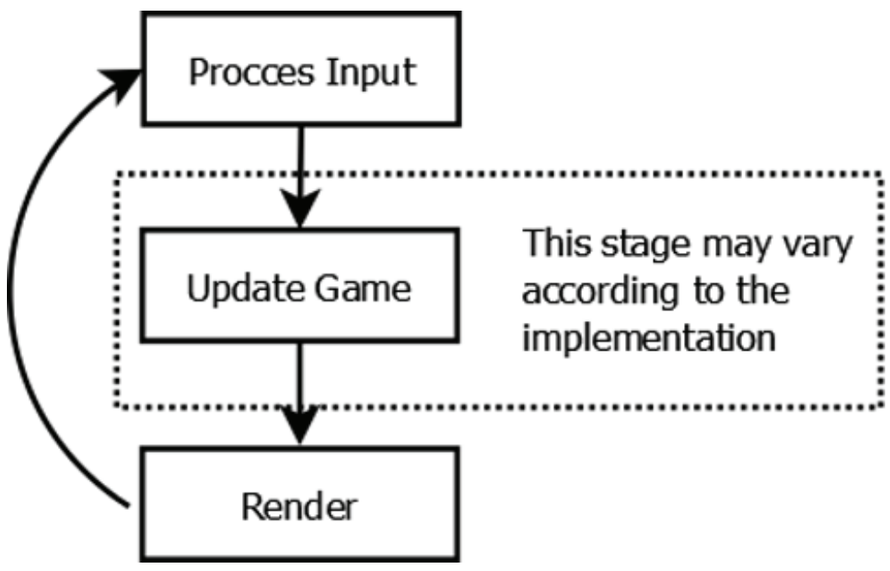

Fig. 03. Basic *Game Loop*

Implementación

Para ejemplificar los patrones de diseño presentados en la sección anterior, utilizaremos un juego de plataformas que hemos dejado disponible a la comunidad[5]. En el resto de esta sección utilizaremos C# para extractos de código y UML para los diagramas de clase.

Patrón Estado

Para implementar el patrón estado hemos utilizado una *Máquina de Estados Finitos* (FSM[6]). Una FSM se puede definir como una máquina abstracta que consta de un grupo de estados, eventos y una función de transición de estado. Las máquinas de estados finitos representan sistemas efectivos de elección de estados ampliamente utilizados en dominios como los videojuegos y la robótica móvil (Bonet, Palacios, & Geffner, 2010), facilitan la comunicación a nivel de diseño de juego con el equipo de desarrollo ya que proporcionan estructuras simples para representar comportamientos. Las FSM son enfoques tradicionales para otorgar alguna conducta particular a un NPC. Estas se han usado en juegos FPS[7] , como Quake de ID Software hasta juegos de estrategia como *Warcraft III* de Blizzard Entertainment, *Age of Empires y Enemy Nations*. También juegos como *Halo 2* de Bungie han recurrido a las FSM en su variante HFSM (Máquina de estados jerárquico) (Sweetser & Wiles, 2002).

La estructura implementada de la máquina de estados finitos comienza por una clase FSM que tiene la responsabilidad de guiar el cambio de estado haciendo uso de los métodos contenidos en la clase *FSMState* (Fu & Houlette, 2003). Estos métodos son:problemas específicos del desarrollo de videojuegos.

La estructura implementada de la máquina de estados finitos comienza por una clase FSM que tiene la responsabilidad de guiar el cambio de estado haciendo uso de los métodos contenidos en la clase FSMState (Fu & Houlette, 2003). Estos métodos son:

• *Act*: Contiene lo que hará el estado.

• *CheckTrans*: Método que verifica si alguna de las transiciones es válida para cambiar de estado.

• *Enter/Exit*: Métodos opcionales.

- *DoBeforeEntering.*
- *DoBeforeLeaving.*

Como se puede apreciar en la figura 4 la clase FSMState posee una lista de FSMTransition. Estas transiciones tienen una referencia al estado de destino y usan su método *IsValid* para comunicar a la FSM que es momento de hacer transición. De esta forma cada vez que se cumpla la condición definida en el método *IsValid* podremos retornar true y dar la señal para cambiar de estado. Como se aprecia en el diagrama de estado de la figura 5, tomaremos un enemigo que posee los estados *IdleState* y *Walk State*, estos dos comportamientos son representados por instancias de clases que heredan de *FSMState*, también se aprecian las transiciones *Idle time is 0* y *Walk time is 0* que permiten el cambio de estado, las dos instancias heredan de *FSMTransition*.

- *IdleState.*
- *WalkState.*
- *TimeTransition.*

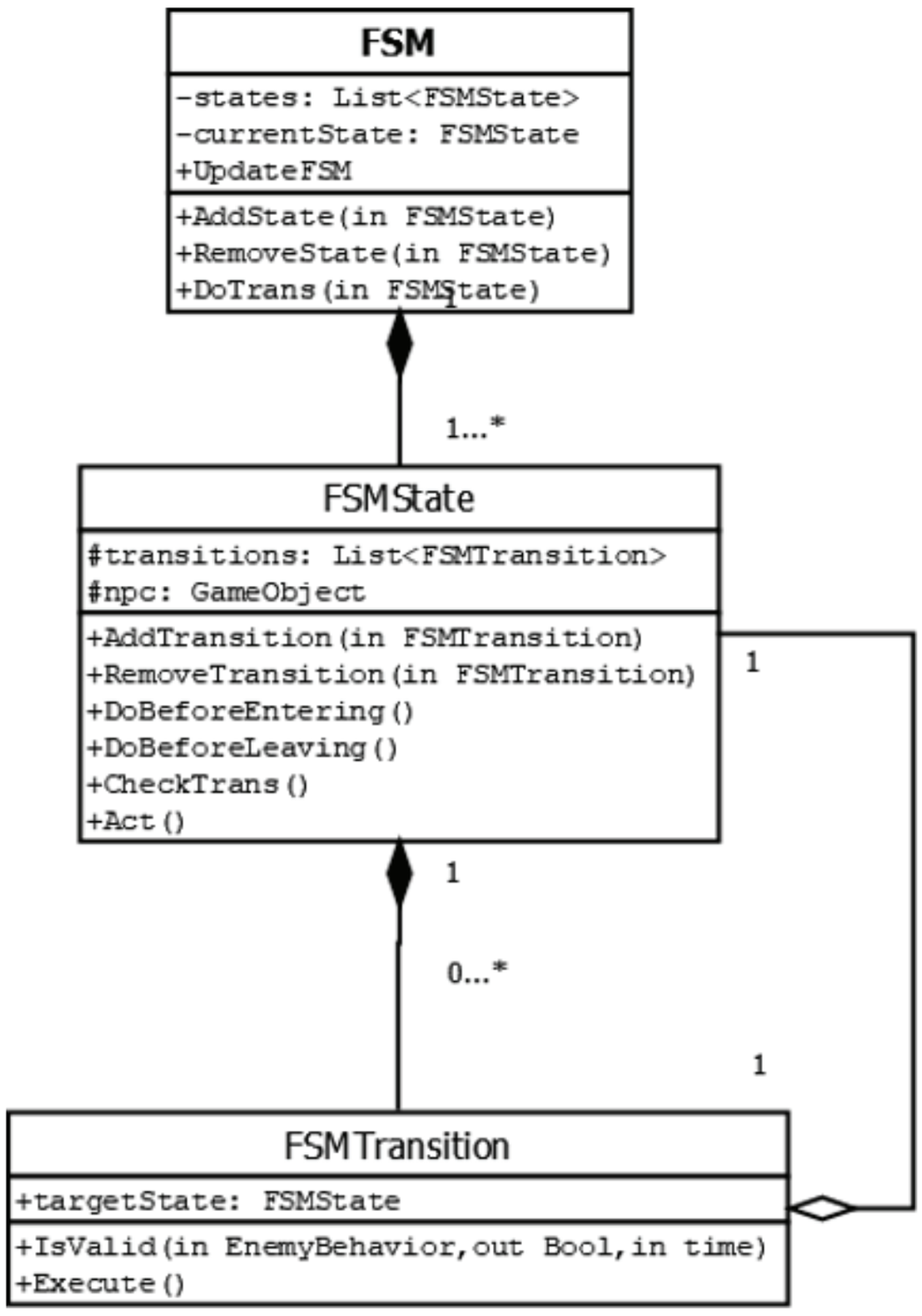

Fig. 04. *FSM Architecture*

[5] *https://github.com/hansschaa/FSM Command Patterns*
[6] *del inglés: Finite-State Machine.*
[7] *del inglés: First Person Shooter.*

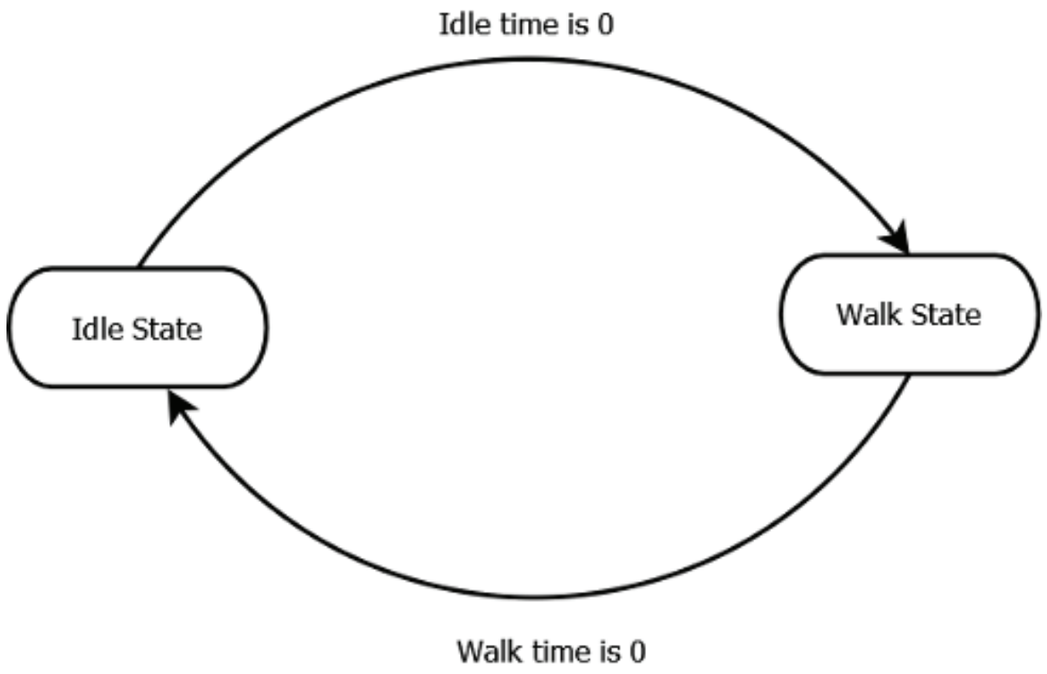

Fig. 05. Estados del enemigo ejemplo

Asumiremos que la transición fue agregada a cada lista de transiciones de los estados con sus parámetros correspondientes junto al tiempo de espera para desencadenar el cambio de estado. En el algoritmo de la figura 6 se puede apreciar lo anteriormente descrito. Recorreremos todos los elementos en nuestra variable transitions entregando el tiempo actual desde que hicimos el cambio de estado. Como ilustra el algoritmo de la figura 7 al recibir true por parte del método *IsValid*, estamos afirmando el cambio de estado y devolveremos a la instancia de nuestra FSM la transición que ha devuelto true. Como ilustra el algoritmo de la figura 8 el método *FSMUpdat*e será quién lleve a cabo el ciclo principal de nuestra FSM (Método *UpdateFSM* es llamado en cada frame por nuestro NPC para así mantener actualizado su estado frente a los cambios del entorno), preguntando en cada frame si alguna de las transiciones del estado actual del NPC es válida. Si éste es válido, el método *DoTrans* es invocado para hacer el cambio de estado y llamar a las funciones que configuran los parámetros correspondientes del nuevo y antiguo estado. En la figura 9 por medio de un diagrama de flujo se muestra el proceso para configurar e iniciar la *Finite State Machine* de una entidad cualquiera. Se pueden apreciar las etapas básicas a considerar para implementar el patrón Estado.

```
1.  public virtual FSMTransition CheckTrans(){
2.      foreach (FSMTransition t in transitions){
3.          if(t.IsValid(npc)){
4.                  return t;
5.          }
6.      }
7.      return null;
8.  }
```

Fig. 06. Clase *FSMState, método CheckTrans*

```
1.  public boolean IsValid(EnemyBehaviour enemyBehaviour, float time){
2.      if (time > _totalTime)
3.              return true;
4.
5.      return false;
6.  }
```

Fig. 07. Clase TimeTransition, método IsValid

```
1.  public void DoTrans(FSMTransition t){
2.      t.OnTransition();
3.      currentState.DoBeforeLeaving();
4.      currentState = t.targetState;
5.      currentState.DoBeforeEntering();
6.  }
7.
8.  public virtual void FSMUpdate(){
9.      FSMTransition t = currentState.CheckTransitions();
10.     if (t != null)
11.         DoTrans(t);
12.     else
13.         currentState.Act();
14. }
```

Fig. 08. Clase FSM, métodos *DoTrans* y *FSMUpdate*

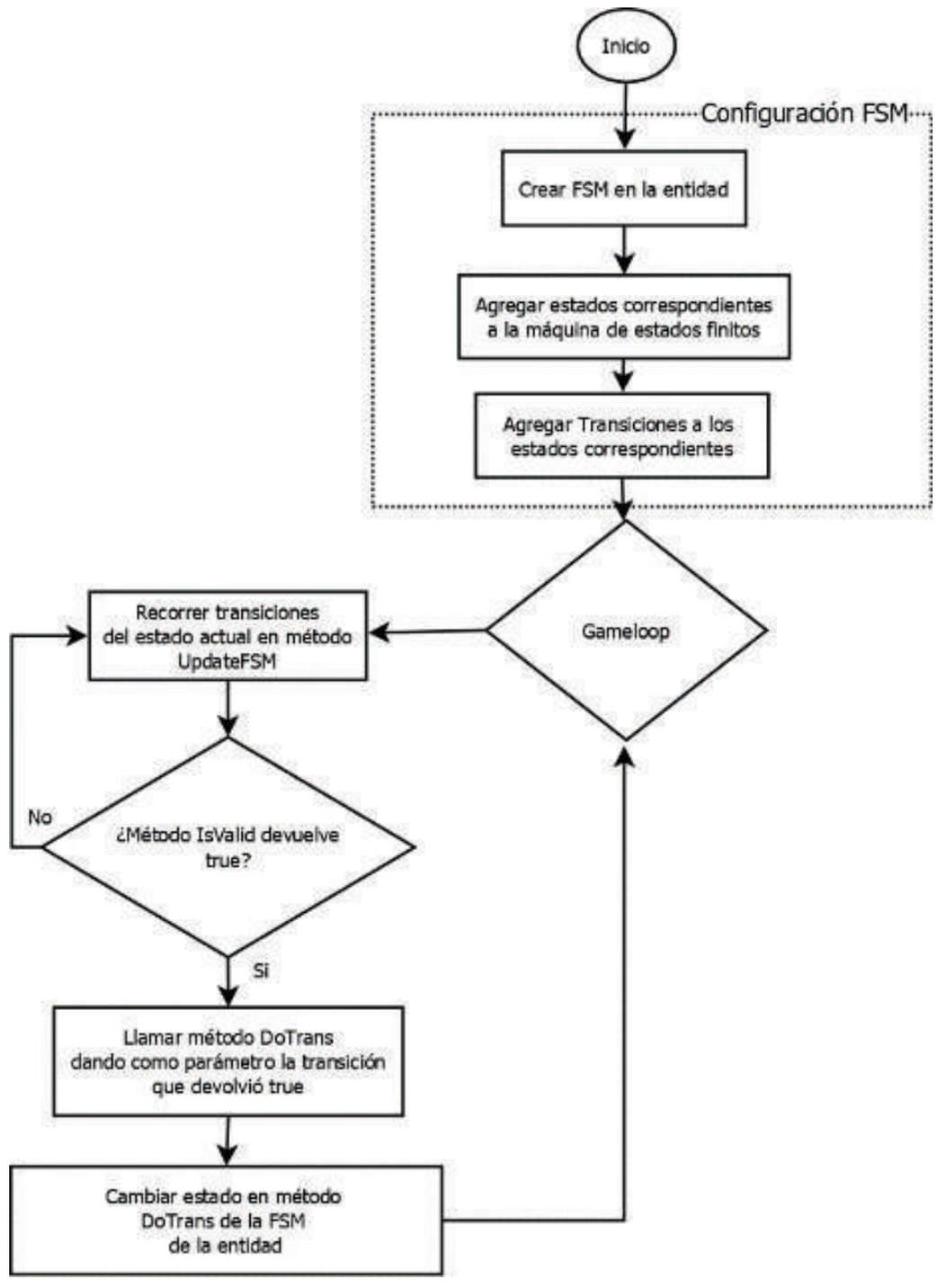

Fig. 09. Diagrama de Flujo FSM

Patrón Comando

En la implementación del Patrón Comando (ver figura 10) podemos identificar una clase que nos permitirá recibir el input del usuario. A esta clase la llamaremos *PlayerInputHandler*. Acá preguntaremos por el tipo de entrada recibida y despacharemos la ejecución de los comandos. La siguiente clase importante por mencionar es la clase *Command*. Heredaremos de ésta para crear cada acción específica ligada a un *input* del usuario. En adición a estas clases tenemos también una llamada *CommandProcessor*, la cual ejecutará el cambio de comando, tal como lo hace la clase FSM en el patrón anterior. La clase PlayerInputHandler contendrá tres variables que heredan de *Command*, estas corresponden a las acciones de saltar, caminar y golpear. Por medio del método *HandleInput* haremos la captura de la entrada del usuario al momento de jugar. *Command* será la superclase para cada comando a gatillar, por ejemplo, tendremos variables del tipo antes mencionado como saltar, caminar, correr, agacharse, etc. En la figura 11, encontramos el evento *HandleInput* que es donde escuchamos por las entradas del usuario y hacemos la ejecución del comando correspondiente. A continuación, en el algoritmo de la figura 12 podemos visualizar la implementación del método Execute, encontramos la llamada a *IsValid* donde se encuentra la condición para que éste comando sea accionado, si esta devuelve true podemos seguir con la ejecución de ese comando.

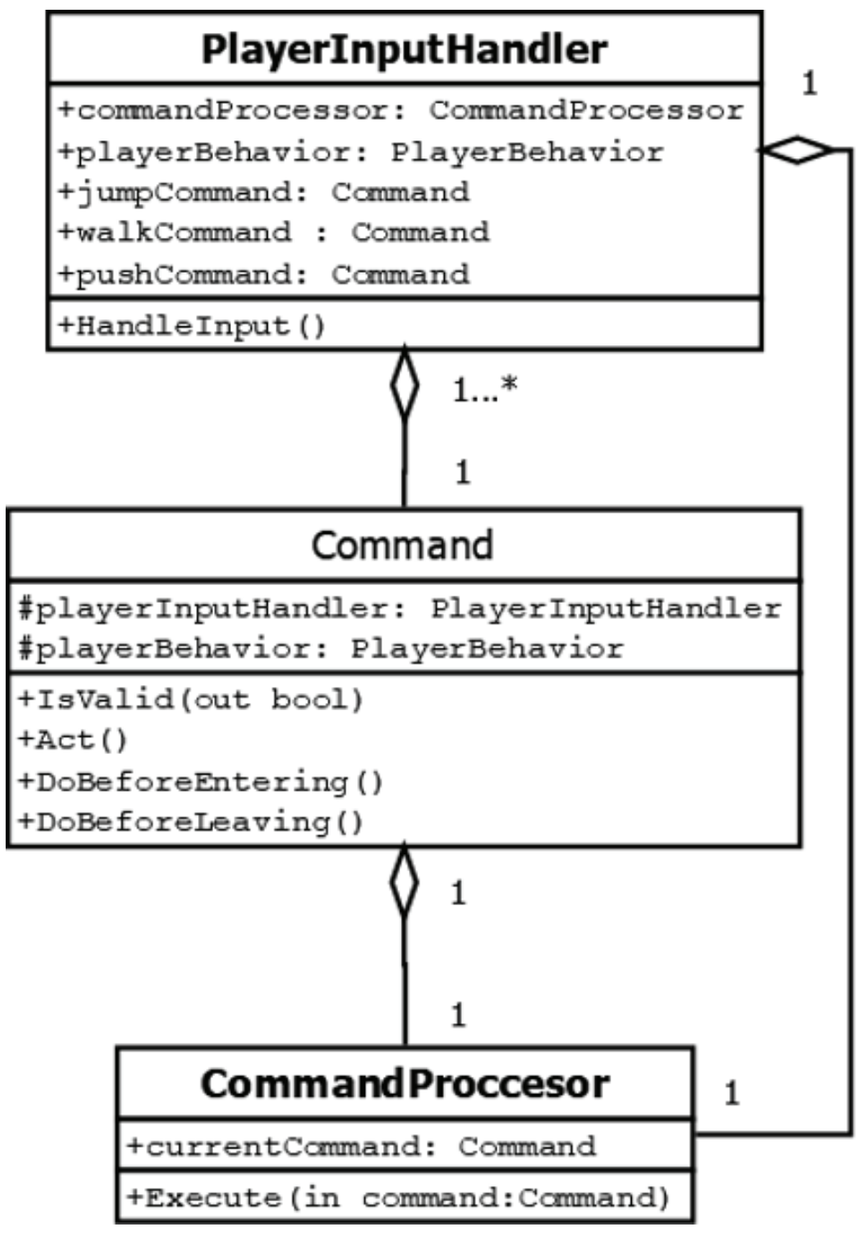

Fig. 10. *Command Pattern Architecture*

Discusión

Respecto de la FSM sugerida en la sección anterior, podemos apreciar que un enfoque basado en la modularidad hace de nuestro código un sistema con mayor mantenibilidad y expansibilidad. En la implementación descrita en (Aversa et al., 2018) se evidencian dos problemas claves:

- Transiciones estáticas: Las transiciones eran embebidas de forma estática en la clase de cada estado, lo que impedía usar un mismo estado para dos entidades con diferentes comportamientos, ya que eventualmente podían tener distintos estados a los cuales hacer transición.

- Sistema de IA acoplada al controlador: La forma en la cual actualizamos el comportamiento de nuestra entidad no estaba separada de su propio controlador.

Estos problemas no siguen presentes con la estrategia para definir una FSM propuesta en este artículo.

Unity Asset Store

El *Asset Store* de *Unity* es un portal que reúne assets provistos por la comunidad, tanto de pago como gratuitos. Cabe destacar que, a pesar del carácter oficial de esta tienda, *Unity* no hace una revisión exhaustiva de la calidad de los recursos enviados. Realizamos una búsqueda de assets que provean implementaciones de los patrones comando y estado en esta tienda. Encontramos cinco y cero respectivamente. A continuación, describimos los assets encontrados de FSM, junto a nuestra apreciación sobre ellos:

- *Finite-State Machine Scriptable Objec*t: Posee clases de tipo Enum para los estados y transiciones, lo que puede desencadenar difícil mantención y extensión cuando éstos aumenten, también incluye las transiciones de forma estática en cada uno de los estados.

- *Stateorio*: Búsquedas innecesarias debido a la arquitectura codificada, por ejemplo, en el cambio de estado de la FSM siempre busca el estado

destino cuando éste se podría haber obtenido de la transición que retornó true en su método *IsValid*.

- C# *State Machine*: Mantiene transiciones de forma estática en cada estado de la entidad y no hay desacoplamiento entre la entidad y su comportamiento.

- *BitFSM*: Confuso de usar y documentación vaga para un *asset* de nivel intermedio de uso.

- *RobustFSM*: No existen la transición como tal, incluye código de forma estática en el método que autoriza el cambio dentro de los estados, sistema poco intuitivo.

Conclusiones

Es conocido que llevar a la práctica antipatrones como el generar spaguetti code puede desencadenar en un fracaso temprano del proyecto (Tufano et al., 2015). El uso de patrones de diseño nos puede ayudar a generar estructuras y convenciones que nos liberan de cierto riesgo a la hora de desarrollar *software*. En el desarrollo de videojuegos esto es aún más relevante, debido a que las mecánicas y ciertas características de juego son volátiles por lo ligero y cambiante que puede llegar a ser un *Game Design Document* (GDD). Una solución, sería incluir en el GDD estas buenas prácticas que contribuyen al desarrollo de software modular, robusto y resistente a las eventuales modificaciones del diseño en términos funcionales y no funcionales. Respecto de los *assets* encontrados podemos concluir, que sólo *BitFSM* muestra un sistema modular, el cual con algunos cambios y esfuerzo de parte del equipo de desarrollo puede llegar a ser usado en un proyecto serio pese a su poca documentación. Debido a que no encontramos assets gratis que aborden el patrón comando no se pudo hacer un análisis, dejando como propuesta gratuita la implementación mostrada en este documento. Sin lugar a duda las ventajas de usar patrones de diseño en el desarrollo de nuestros videojuegos se pueden seguir estudiando con mayor profundidad. Como trabajo futuro pro- ponemos hacer investigaciones logrando como producto datos numéricos que den cuenta por medio de métricas acertadas, que el construir software con ayuda de patrones de diseño reflejan una clara disminución de horas-hombre invertidas en mantención, expansibilidad y depuración del código, para reducir costos de tiempo y dinero.

Referencias

Abbes, M., Khomh, F., Gueheneuc, Y.-G., & Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th european conference on software maintenance and reengineering* (pp. 181–190).

Ampatzoglou, A., & Chatzigeorgiou, A. (2007). Evaluation of object-oriented design patterns in game development. *Information and Software Technology, 49(5)*, 445–454.

Antoniol, G., Fiutem, R., & Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. In *Proceedings. 6th international workshop on program comprehension. iwpc'98* (cat. no. 98tb100242) (pp. 153–160).

Aversa, D., Kyaw, A. S., & Peters, C. (2018). *Unity artificial intelligence programming: Add powerful, believable, and fun ai entities in your game with the power of unity2018!* Packt Publishing Ltd.

Baron, D. (2019). *Hands-on game development patterns with unity 2019: Create engaging games by using industry-standard design patterns with c#.* Packt Publishing.

Bonet, B., Palacios, H., & Geffner, H. (2010). Automatic derivation of finite-state machines for behavior control. In *Twenty-fourth aaai conference on artificial intelligence*.

Csikszentmihalyi, M. (2009). *Flow: The psychology of optimal experience.* New York : Harper and Row.

Freeman-Hargis, J. (2006). Ai game programming wisdom 3. In *S. Rabin (Ed.), (chap. Using STL and Patterns for Game AI).* Charles River Media.

Fu, D., & Houlette, R. (2003). Ai game programming wisdom 2. In S. *Rabin (Ed.), (chap. The Ultimate Guide to FSMs in Games)*. Charles River Media.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

Gregory, J. (2017). *Game engine architecture*. AK Peters/CRC Press.

Nystrom, R. (2014). *Game programming patterns*. Genever Benning.

Qu, J., Song, Y., & Wei, Y. (2013). Applying design patterns in game programming. In *Proceedings of the international conference on software engineering research and practice (serp)* (p. 1).

Sayed, R., Sinha, R., & Reith, G. (2017, June 2). *16 games with incredible artificial intelligence.* https://www.youtube.com/watch?v=p4KqryLMseQ.

Sweetser, P., & Wiles, J. (2002). Current ai in games: A review. Australian *Journal of Intelligent Information Processing Systems, 8*(1), 24–42.

Toftedahl, M. (2019). *Which are the most commonly used game engines?* Retrieved 2019-09-30, from https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/Which_are_the_most_commonly_used_Game_Engines.php

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *Proceedings of the 37th international conference on software engineering volume 1* (pp. 403–414).

Zimmer, W., et al. (1995). Relationships between design patterns. *Pattern languages of program design, 57*, 345–364.