Diagramas esenciales del lenguaje unificado de modelado para los requisitos agiles en el desarrollo de software.

Cristian Vicente Ramírez chaparro.

Universidad Nacional Abierta y a Distancia.

Escuela de Ciencias Básicas, Tecnología e Ingeniería

Tecnología en Desarrollo de Software

Bogotá D.C.

2020.

Diagramas esenciales del lenguaje unificado de modelado para los requisitos agiles en el desarrollo de software.

Cristian Vicente Ramírez chaparro.

Universidad Nacional Abierta y a Distancia.

Escuela de Ciencias Básicas, Tecnología e Ingeniería

Tecnología en Desarrollo de Software

Bogotá D.C.

2020.

Nota de aceptación

_____

_____

_____

_____

_____

_____

Firma del presidente del jurado

_____

Firma del jurado

_____

Firma del jurado

Bogotá, 30 noviembre de 2020

Dedicatoria

Esta monografía la cual es de carácter explicativo está dedicada a todos los lectores que busquen información sobre el UML y las metodologías águales, especialmente a desarrolladores de software, ingenieros del área informática o gestores de proyectos informáticos, y más especialmente a personas que son nuevas en el uso tanto de metodologías agiles como el lenguaje de modelado unificado en la tecnología de desarrollo de software.

También dedico este articulo al jurado encargado de la revisión de esta monografía y su presidente. Asimismo, a la tutora que me ayudo con la elaboración de este documento Carmen Emilia Rubio especiales gracias por su ayuda a la hora de la elaboración de este documento. Espero que la información compartida en este documento sea vista con aceptación.

Agradecimiento

Agradezco a la Universidad Nacional Abierta y a Distancia esta experiencia, a todos los tutores que hicieron parte de mi formación académica y especialmente a Carmen Emilia Rubio y Rafael Pérez Holguín quien con su ayuda pude dar con todo lo necesario para el desarrollo de esta monografía.

A mis padres que Dios me los aguarde y a los cuáles les estoy agradecido por su paciencia y solidaridad. También agradezco a Dios quien me permitió el desarrollo de esta monografía. Por último agradezco a Javier Jiménez quien me ayudo con los requisitos para que el siguiente documento sea real.

Resumen

En este documento se presenta el Lenguaje Unificado de Modelado, así como las características de las metodologías agiles de desarrollo de sistemas informáticos de esta manera se divide principalmente en cuatro secciones las cuales son: el UML, las metodologías agiles (MA), la constancia de uso de diagramas y métodos de desarrollo y por último el uso de gráficos del lenguaje sin tener en consideración cual es o son las agile methodologies (AM) elegidas para el desarrollo.

Este tema esta parametrizado bajo la cantidad de frecuencia en la que se usan los diagramas, para entender cuáles de estos son tanto fundamentales como críticos para el desarrollo de un sistema informático. Para esto se cuenta con los casos de éxito, así como la experiencia de los mismos creadores teniendo esto presente ya que hay modelos anteriores al nombrado de los cuales los autores también fueron participes.

Para cumplir con el contexto: se describe que es el UML, cuál es su historia también cuales son los diagramas relacionados a este lenguaje, además se tratan tanto las ventajas como las desventajas, así como los casos de éxito; también se dio investigación sobre las metodologías agiles y cuáles son sus características.

Palabras clave

UML, metodologías agiles, desarrollo, diagramas críticos.

Abstract

In this document the Unified Modeling Language is presented as well as the characteristics of the agile methodologies of development of computer systems in this way it is divided mainly into four sections which are: the UML, the MA, the constancy of use of diagrams and methods of development and finally the use of graphics of the language without taking into account which is or are the AM chosen for development.

This topic is parameterized under the amount of frequency in which the diagrams are used, to understand which of these are both fundamental and critical for the development of a computer system. For this, we have the success stories, as well as the experience of the creators themselves, bearing this in mind, since there are previous models to the named one in which the authors were also participants.

In order to comply with the context: it is described what the UML is, what its history is also what the diagrams related to this language are, as well as the advantages and disadvantages, as well as the success cases; Research was also given on agile methodologies and what their characteristics are.

Key words

UML, agile methodologies, development, critical diagrams.

Contenido

Contenido de imágenes

Contenido de tablas

Introducción

El lenguaje de modelado unificado (UML) es un lenguaje que busca el poder hacer cualquier tipo de diagrama a cualquier tipo de escala pare representar la interacción del sistema y los objetos que lo conforman, la importancia de este recae en sus comienzos.

"El más destacado entre ellos era el Análisis estructurado y diseño estructurado [Yourdon-79] y sus variantes, entre las que se encontraba el Diseño estructurado de tiempo real [Ward-85]. Esos métodos, originalmente desarrollados por Constantine, DeMarco, Mellor, Ward, Yourdon y otros, alcanzaron cierta renegación en el área de los grandes sistemas, especialmente en sistemas contratados por el gobierno en los campos aeroespacial y de defensa, en los que los contratistas insistían en un proceso de desarrollo organizado y una amplia documentación del diseño y la implementación." (Rumbaugh, Jacobson, & Booch, El Lenguaje Unificado de Modelado Manual de Referencia 2.0, 2007).

Esto ha aumentado desde que la potenciación de la tecnología de ha dado dando como resultado que es uno de los más populares y así mismo, uno de los más importantes en la tecnología de desarrollo de software.

En conjunto con estos diagramas se espera tratar en esta monografía las metodologías ágiles como: el Scrum, la Programación eXtrema, El TTD, la Integración continua, Refactoring también el Kanban, los autores más importantes en este ámbito son

Kent Beck desarrollando extrema (Bench & Cinthia, 2015) así como el test driven development (TDD) (Beck, 2002), Martin Fowler que propuesto en el campo de metodologías la integración continua, así como el refactoring. Taiichi Ohno desarrollo en la empresa de automóviles Toyota el "Kanban" por otra parte, Ikujiro Nonaka como Takeuchi desarrollaron e identificaron el "Scrum" (Bahit, 2011).

Lo más primordial a la hora de aprender UML es la forma correcta de leer, así como desarrollar sus esquemas ya que estos son los fundamentales para el desarrollo de un sistema de cómputo, debido a que se van interpretando en capas de abstracción hasta lograr un código fuente el cual debe de estudiado y mantenido según los esquemas de requisitos.

Teniendo más a consideración para lo que esta monografía desarrolla se tiene en consideración el aprendizaje, así como la maestría en los casos de uso, actividades, clases, pero en menor medida los de estado. En cuanto a metodologías agiles se considera de gran importancia saber cómo se componen los grupos de desarrollo las tarjetas de actividades, tanto el cuándo como el modo de contactar con el cliente final.

Aunque ciertas metodologías no son del todo implementadas en las características mencionadas, ha de ser de suma importancia mantenerlos en cuenta cuando se trata de un proyecto nuevo o evitar incertidumbre en un proyecto que ya se esté desarrollando. Siendo así se tratarán los diagramas del UML y su relación con las metodologías agiles, así como presentar cuales son los diagramas críticos para que se pueda dar la una ejecución por las metodologías agiles (MAs).

Planteamiento del problema

El desarrollo del lenguaje de modelado unificado (UML) ha traído grandes ventanas al campo de la informática en cuanto a un desarrollo más viable, y genera menos obsolescencia en los sistemas de información; empleo que también se da cuando se aplica una metodología ágil hará que los tiempos sean menores debido a que se puede medir la cantidad de esfuerzo que puede realizar una dicha función dentro de un programa.

Sin embargo, la gran cantidad de diagramas generan la desventaja “ochenta veinte (80/20)” cual determina que el ochenta por ciento (80%) de los problemas en el diseño de un sistema informático se puede resolver con el veinte por ciento (20%) de estos, ante tantos métodos de desarrollo ágil solo se debe de escoger uno o en algunos casos varios de forma que se complementen.

Todos estos problemas dados hacen que el presente documento cuyo carácter es explicativo, se desarrolle bajo el título de diagramas esenciales del UML para requisitos agiles en el desarrollo de software, ya que muchos tipos de desarrollo requieren de diversos tipos de soluciones dando un como resultado un montón de gráficos, así como MA, que si no se saben usar podrán no tener el efecto óptimo.

En caso del lenguaje incluso puede llegar a retrasar el proyecto si no se sabe adecuar al sistema que se está desarrollando o evaluando, si esto se da entonces no se puede evitar tener costos de software altos, otro gran inconveniente es no poder medir con exactitud cuánto se va demorar una función en su implementación, o saber cuándo, así como dónde se están generando los cuellos de botella.

Estando entre lo más delicado, el hecho de tener un proyecto altamente diagramado pero desactualizado, cuya estructura no cumple con la visión de cómo realmente está el código, siendo este uno de los fines últimos de los esquemas.

Justificación

La razón por la cual se propone el desarrollo de la monografía es presentar un aporte a la disciplina que permita determinar cuál o cuáles de los diferentes diagramas considerados en el lenguaje de modelado unificado que son indispensables para un proceso de requisición ágil, con el objetivo de propender por la generación de código en tiempo más cortos, haciéndolo de carácter explicativo.

El impacto social académico se genera porque la monografía se convierte en una fuente de consulta académica para los estudiantes que están es su proceso de formación o quieran profundizar en la Lenguaje de Modelado Unificado. De igual forma como segundo escenario es la comunidad en general, que quieran considerar la propuesta en entornos productivos, para apoyar la toma de decisiones e incorporar su contenido o parte de él.

En sus metodologías de trabajo o por el contrario evaluar otras posibilidades que en todos los casos dependerán de los objetivos y perspectivas de los proyectistas, gerentes de proyectos o en general los responsables de la dirección del proyecto, consiguiendo el poder trabajar con una menor cantidad de documentación que es innecesaria en las primeras versiones de un proyecto, y tapar los agujeros de botella con personal disponible con el que no se contara con metodologías menos agiles.

Objetivos

Objetivo general

Describir el Lenguaje de modelado unificado mediante su historia diagramas, casos de usos, actividades, estados y clases, explicando cuáles son sus aplicaciones en metodología ágiles.

Objetivos específicos

- Identificar las fuentes de información necesarias para el desarrollo del presente documento frente a los diagramas de UML y las metodologías agiles.

- Describir que es el UML, su historia, sus diagramas, casos de éxito de implementación, ventajas y desventajas además de cuáles son las metodologías agiles.

- Construir la monografía teniendo en cuenta que es el UML, cuál es su historia, sus diagramas de caso de uso, actividades, estado y clases del UML. Además de las diferentes metodologías agiles.

Diseño metodológico

Esta monografía se formó a través de un método investigativo debido a las diversivas fuentes académicas, libros y diapositivas especializadas en el UML tal como las metodologías ágiles; correlacionada entre ellas para la conclusión del trabajo; dando estos resultandos una metodología cualitativa representada mediante la presente, cuyo carácter resulta explicativo.

Se presenta una viabilidad del proyecto alta ya que hay mucho material de primera de fuente dada por la Universidad Nacional Abierta y a Distancia, además de otras fuentes de gran prestigio académico buscadas por el estudiante; dando a entender que es posible realizar la presentación completa del marco teórico sin grandes obstáculos, con un cumplimiento total en la pregunta de investigación, así como en las temáticas de esta monografía.

UML

Que es UML

En el libro Lenguaje unificado de modelado manual de referencia de los aclamados James Rumbaugh, Ivar Jacobson & Graddy booch definen el UML como un acrónimo de Leguaje Unificado de modelado, por supuesto sus siglas en inglés como Unificated Modeling Lenguaje, un lenguaje de modelado visual que se usa para documentar, construir, visualizar y especificar artefactos de un software como sistema. Comenzando de la siguiente forma:

"El Lenguaje Unificado de Modelado (UML) es un lenguaje de modelado visual que se usa para especificar. visualizar, construir y documentar artefactos de un sistema de software. Captura decisiones y conocimiento sobre los sistemas que se deben construir. Se usa para entender, diseñar, hojear, configurar, mantener, y controlar la información sobre tales sistemas. Está pensado para usarse con todos los métodos de desarrollo. etapas del ciclo de vida, dominios de aplicación y medias. El lenguaje de modelado pretende unificar la experiencia pasada sobre técnicas de modelado e incorporar las mejores prácticas actuales en un acercamiento estándar. UML incluye conceptos semánticos, notaciones, y principios generales. Tiene partes estáticas, dinámicas, de entorno y organizativas. Está pensado para ser utilizado en herramientas interactivas de modelado visual que tengan generadores de código así como generadores de informes. La especificación de UML no define un proceso estándar pero está pensado para ser útil en un proceso de desarrollo iterativo. Pretende dar apoyo a la mayoría de los procesos de desarrollo orientados a objetos" (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

Cual captura conocimientos y decisiones sobre los sistemas que se desean construir, usado para controlar, mantener, configurar, hojear, diseñar y entender la información de los sistemas, pensado para usarse con los medios, dominios de aplicación y etapas del ciclo de vida.

Pretendiendo usar antiguas prácticas de modelado e incorporar las mejores prácticas actuales, pero mediante un acercamiento de estándar. Incluyendo principios generales, notación y conceptos semánticos; teniendo partes organizativas, dinámicas, de entorno y estáticas.

Su especificación no define un estándar, pero está hecho para ser útil en proceso dinámicos, pretendiendo dar mayor apoyo a los procesos de desarrollo orientado a objetos. Este capta la estructura estática y el comportamiento dinámico de un sistema, este sistema se modela con objetos que cumplen una función que finalmente son de beneficio para un usuario externo.

Luego la estructura estática define los tipos de objetos para un sistema y la implementación del mismo, de igual modo su relación entre objetos; el comportamiento dinámico define por su parte la historia de los objetos en el tiempo y la comunicación entre ellos, esto con el fin de cumplir con los objetivos del sistema propuesto.

“El modelar un sistema desde varios puntos de vista separados pero relacionados, permite entenderlo para diferentes propósitos” (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000). El UML no es un

lenguaje de programación: sin embargo, hay herramientas que permiten que el UML se convierta en código.

Utilidad que también permite realizar ingeniería inversa a programas existentes; tampoco es un lenguaje altamente formal pensado para probar teoremas, hay otros métodos como el UML, pero no son tan útiles o prácticos.

“UML es un lenguaje de propósito general, para domino especializado, tales como la composición de IGU, diseño de circuitos VLSI o inteligencia artificial basada en reglas, podría ser más apropiada una herramienta especializada con lenguaje especial. UML es un lenguaje de modelado discreto. No se creó para modelar sistemas continuos como los basados en ingeniería y física.

UML quiere ser un lenguaje de modelado universal, de propósito general, para sistemas discreto, tales como los compuestos por software, firmware o lógica digital” (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000) . Con este párrafo se cierra el breve resumen de UML en su versión uno punto tres.

En esa se nos especifica que el UML es un modelo estándar para tareas de sistemas estándar, y aunque muy completa, no está especializa para realizar diagramas complejos de alguna disciplina relacionada con la sistematización informática.

Historia del UML

Cuando comenzó la programación orientada a objetos entre mil novecientos setenta (1970) y ochenta (1980) se generó una incertidumbre conforme a que diseño se tomaría orientado a objetos, durante los finales de los ochentas e inicios de los noventas se generaron varios libros en los cuales se indicaba por distintos autores el proceso de diseño para un aplicativo orientado a objetos. En el libro UML 1.3 se detalla un comienzo del proceso de la siguiente manera.

"Los métodos de desarrollo para los lenguajes de programación tradicionales, tales como Cobol y Fortran, emergieron en los años 70 y llegaron a ser ampliamente difundidos en los 80. Principalmente entre ellos estaba el análisis estructurado y el diseño estructurado [Yourdon-79] y sus variantes tales como Desafío estructurado de tiempo real [Ward-85] y otros. Esos métodos originalmente desarrollados por Constantine, DeMarco, Mellor, Ward, Yourdon, y otros, alcanzaron cierta penetración en el área de los grandes sistemas, especialmente para los proyectos contratados por el gobierno en los Campos aeroespacial y de defensa, en los cuales los contratistas insistieron en un proceso de desarrollo mecanizado y en una amplia documentación del diseño e implementación del sistema. Los resultados no fueron siempre tan buenos como se esperaba". (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

Para el OPPSLA 94 se presentaban contrariedades menores entre los distintos autores y sus formas de diseños, todas aprobadas por profesionales en materia para esta

fecha la OMG ya consideraba la estandarización y dejo carta abierta sin embargo no se evidenciaba de un ambiente llamativo para que esta estandarización fuera realizable.

Graddy Boch intento organizar una reunión informal que le fue de fracaso. Sin embargo, Jim Rumbaugh renuncio a su trabajo para unirse a Boch en la estandarización de modelos. Para mil novecientos noventa y cinco el UM había sido estandarizado en su versión 0.8 ese mismo año Ivar Jacobson se uniría al equipo unificado. En el libro UML 1.3 se escribe que hubo un esfuerzo de unificación cada vez mayor.

"Hubo algunos intentos tempranos de unificar los conceptos entre métodos. Un ejemplo destacable fue Fusión por Coleman y sus colegas [Coleman-94]. el cual incluye conceptos de OMT [Rumbaugh-91], Booch [Booch.91], y CRC [Wirfs-Brock-90]. Como no involucró a los autores originales, permaneció como otro nuevo método en lugar de un reemplazo de varios de los métodos existentes. El primer intento exitoso de combinar y reemplazar los métodos existentes llegó cuando Rmnbaugh se unió a Booch en Rational Software Corporation en 1994. Ellos empezaron combinando conceptos de los métodos OMT y Booch obteniendo como resultado una primera propuesta en 1995. En ese momento Jacobson también se unió a Rational y comienza a trabajar con Booch y Rumbaugh. Su trabajo conjunto fue llamado Lenguaje Unificado de Modelado (UML). El impulso ganado al tener a los autores de tres de los métodos más importantes trabajando juntos para unificar sus enfoques desplazó la balanza en el campo de las metodologías orientadas a objetos, donde había habido muy poco incentivo (o al menos poca voluntad) de los metodólogos de abandonar algunos de sus propios conceptos para alcanzar la armonía." (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

Para mil novecientos noventa y seis (1996) el equipo había desarrollado su método poniéndole otro nombre: Lenguaje unificado de modelado (UML). Se creó una fuerza de trabajo en el OMG con el fin de estandarizar el lenguaje, Mary Loomis fue nombrada directora y Jim Odel se unió como co-director, planeaba una negociación con rational studio para la estandarización del lenguaje.

Para mil novecientos noventa y siete (1997) varias organizaciones enviaron sus estándares de modelado las cuales se enfocaban en un metamodelo y una notación opcional, En respuesta a la OMG rational libero su versión 1.0 de UML, en todo caso OMG acepto la estandarización del modelo en septiembre de este año, donde estaban varios socios de gran reconocimiento en el medio que ayudaron a esta estandarización como lo son IBM, Microsoft y Oracle.

En un contexto más amplio se demuestra que el UML como estandarización de modelado comenzó a partir de los setenta (70) con la estructura y análisis de diseño, posterior mente este método evolucionaría a las técnicas de orientación a objetos en mil novecientos ochenta (1980) donde el diseño se partiría en tres líneas de desarrollo la de Booch: mil novecientos noventa y uno (1991), mil novecientos noventa y tres (1993).

OMT trabajado por James Rumbaugh en sus estándares uno y dos, e Ivar Jacobson quien tendría en método OOSE. Al presentarse el modelo UM se unieron los métodos Boch y OMT cuales un año después serian complementados por los métodos OOSE generando el UML 1.0.

En el libro UML 1.3 se nos revelan algunas empresas y personas que hiceron parte de la propuesta: “Las siguientes personas formaron parte del equipo principal de desarrollo de la propuesta de UML o trabajaron en el equipo de revisién:

Data Access Corporation: Tom Digre.

DHR Technologies: Ed Seidewitz.

HP: Martin Griss.

[BM: Steve Brodsky, Steve Cook, Jos Warmer.

I-Logix: Eran Gery, David llarel.

ICON Computing: Desmond D' Souza.

IntelliCorp and James Martin &Co.: Conrad Bock, James Odell.

MCI Systemhouse: Cris Kobryn, Joaquin Miller.

ObjectTime: John Hogg, Bran Selic.

Oracle: Guus Ramackers.

Platinum Technologies: Dilhar DeSilva.

Rational Software: Grady Booch, Ed Eykholt. Ivar Jacobson,

Gunnar Overgaard, Karin Palmkvist, James Rumbaugh

SAP: Oliver Wiegert.

SOFTEAM: Philippe Desfray.

Sterling Software: John Cheesman, Keith Short.

Taskon: Trygve Reenskaug.

Unisys: Sridhar lyengar, GK Khalsa.” (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

Los modelos evolucionaban de uno a otro de forma separada por tanto era fácil confundir a los usuarios, si se daba una orientación a objetos esto hace que el mercado sea

más estable y se puedan usar modelos maduros y de salidas rápidas. La combinación de los modelos podría hacer que los modelos encontraran solución a problemas que no podrían haber sido bien tratados con los modelos por separado.

Una función aparte del desarrollo de software es el modelado de sistemas industriales dado el caso como lo puede ser una planta embotelladora donde se muestra como un actor puede realizar toda clase de funciones con un sistema o que la secuencia en la que se ejecuta el ciclo siempre será de estación a estación hasta dar por finalizada la producción de embotellado.

Diagramas en el UML

Diagrama de casos de uso

Definición

Es la secuencia de un proceso mediante la interacción de un usuario u otro sistema con los pasos detallados de estos en el mismo que se estudia.

Funcionamiento

En esencia es la interacción de un usuario con el sistema entendiéndose en tres fundamentos: el primero es que capta alguna función del usuario, como segundo se encuentra que varía de tamaño además de lograr un objetivo para el mismo. El tercero es adquirir analizando que es lo que desea hacer, abordando funciones concretas dando nombre y texto descriptivo.

En el libro UML 1.3 se describe: “La vista de los casos de uso, modela la funcionalidad del sistema según lo perciben los usuarios externos, llamados actores. Un caso de uso una unidad coherente de funcionalidad. expresada como transacción entre los actores y el sistema. El propósito de la vista de casos de uso es enumerar a actores y casos de uso, y demostrar qué actores participan en cada caso de uso.” (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

En casos de que sea muy grandes es mejor que muestre sus actividades en un diagrama de este tipo siendo más practico un cliente por diagrama que puede no solo ser una persona, sino que además puede ser otro sistema siendo una de las confusiones más comunes entre actores.

Todas las interacciones con un sistema remoto deben aparecen en el gráfico cuales solo se debe mostrar cuando la interacción es con otros métodos se deben de apreciar las acciones cuando estás solo requieran de caso. Algunos desarrolladores caen erróneamente en la idea que un sistema es un actor.

En un caso de uso siempre importa de forma ultima el caso de uso, pero se puede dar que un usuario configure a varios iguales, se pueden moldear para que las configuraciones afecten a otro también tiene una lista de estos diagramas, rastrear a los actores es saber que requieren cada uno siendo esto necesario dadas las necesidades por las que compiten cada uno de ellos, el saber que recursos requiere cada uno determina si se comparten gráficos de la misma índole, eso también puede darse en experiencias de seguridad por poner algún ejemplo del fenómeno.

En una ontología para la representación de conceptos de diseño de software se señala que “Un caso de uso se puede definir Como una secuencia de acciones, incluyendo acciones alternas, que el sistema puede realizar y que producen un resultado concreto para un actor que interactúa con el sistema.” (Gloria, Acevedo, & Moreno, 2011).

Elementos

Actores: es el término empleado para **referirse a los usuarios** mediante el desempeño de un papel, ha de a calarse que un cliente puede tener varios roles. Su representación se puede visualizar en la figura No. X. Un ejemplo de un actor en un caso uso es un Gerente.

Ilustración 1: Elemento Actor. Fuente: Propia.

Acción del usuario: El diagrama de caso de uso esta formadas en su totalidad por interacción llamadas **caso de uso**, cual es la **acción de un usuario** con el sistema o de otro sistema con el sistema, siempre deberá comenzar con un verbo sin tiempo verbal.

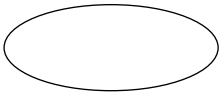

Ilustración 2: Elemento Acción. Fuente: Propia.

Asociación: Si una acción está ligada a otra acción entonces se usa la **asociación** la

Ilustración 3: Elemento Asociación. Fuente: Propia.

cual describe que una lleva a otra dentro del sistema.

La asociación directa: La directa asociación se da cuando una **acción siempre llevara a otra acción.**

Ilustración 4: Elemento Asociación Directa. Fuente: Propia.

Extensión: siempre se trabajará como **extend** sin traducir es la extensión de acción, lo cual significa que una cual está asociada con otra, pero esta solo se **ejecutara dada determinada situación,** también se puede dar cuando tal **puede tener dos posibles resultados** que **no se ejecutaran a la vez.**

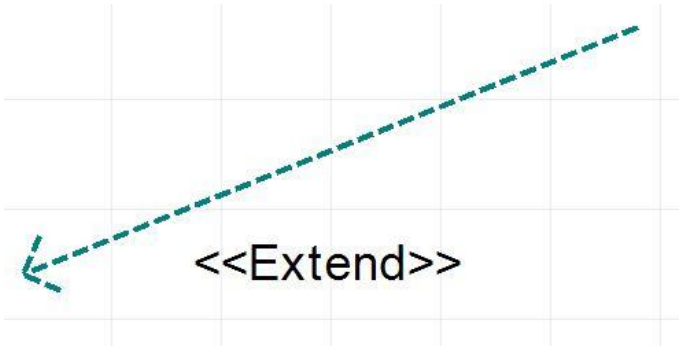

Ilustración 5: Elemento Extend. Fuente: Propia.

Generalización: se da cuando un caso de uso **lleva a otro diagrama igual** simplificado en una simple acción o caso de uso dentro del diagrama actual.

Ilustración 6: Elemento Generalización. Fuente: Propia.

Inclusión: **include** que se trabaja siempre sin traducir es el hecho de que una acción **automáticamente llevara a otra acción** se diferencia de las asociaciones directas porque solo se da entre las mismas burbujas.

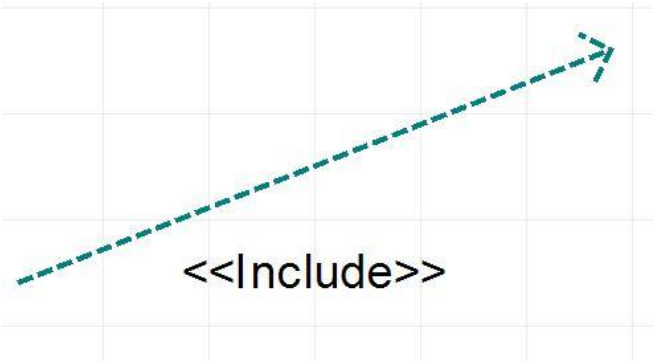

Ilustración 7: Elemento Include. Fuente: Propia.

Sistema de caso de uso: se utiliza como medida para evitar que aparezca en el diagrama un **actor con el nombre sistema**, todos los que se realice se colocan dentro de la caja del sistema dando a entender que todas las actividades se están ejecutando dentro del sistema y que este **interactúa de manera dinámica con el usuario**.

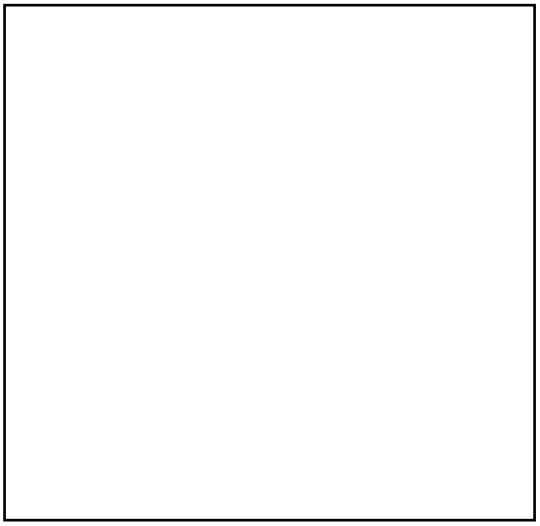

Ilustración 8: Elemento Sistema. Fuente: Propia.

Dependencia: se usa cuando **una actividad** dentro y **depende de otra** para ejecutarse, por lo general se utiliza cuando se da una secuencia que se puede explicar brevemente sin un diagrama de secuencias, se puede ver en interacciones complejas donde se usa un mismo recurso o mutua la acción.

Ilustración 9: Elemento Dependencia. Fuente: Propia.

Ejemplo

Se espera desarrollar un prototipo para un videojuego en donde se contrala a un personaje el cual se enfrentará a su enemigo final, para esto hay que saber cuál será el comportamiento del usuario al sistema y viceversa dadas unas mecánicas de prerrequisitos las cuales serán:

- El enemigo generado y el jefe final atacara al personaje jugable.
- El personaje jugable podrá atacar al enemigo.
- El jefe final creara enemigos para el personaje jugable.
- Los ataques del jefe final serán a distancia.
- Los ataques del jugador serán a distancia.
- Los ataques de los enemigos generados serán de tacto.
- El sistema generara música cuando el jefe final este solo.
- Generará otra música cuando haya enemigos generados por el jefe final.
- El jefe final podrá defenderse.
- El personaje jugable podrá defenderse.
- El personaje jugable tendrá una barra de energía que solo se gaste cuando se cubre.
- El personaje jugable podrá volar.

- El personaje jugable podrá moverse hacia los lados
- El personaje jugable podrá moverse haca atrás y delante
- El jefe final se podrá mover hacia los lados, adelante y atrás.
- El jefe final estará volando.
- Los enemigos generados podrán volarán.
- Los enemigos se moverán hacia los lados, hacia delante y hacia atrás.
- Si el jefe final muere se va a una cinemática.
- Si el personaje jugable muere se mostrará una cinemática.

Para una mejor explicación se desarrollarán dos diagramas el primero muestra la interacción del jugador, enemigo y mecánicas en sí, el segundo explica lo relacionado entre el jefe final con el personaje jugable pero también sus diferencias.

Para el primer diagrama se ha de identificar lo siguiente:

1. Actores
2. Acciones y como se relacionan

Para el segundo bosquejo se muestra en la figura No Y se nos presenta.

En el diagrama que se muestra en la ilustración Diez (10) se puede evidenciar como el jugador gasta energía, se defiende, ataca, vuela, se mueve en x e y, mientras que el/los enemigos puedes realizar estas mismas acciones, uno de los enemigos puede generar más enemigos, el sistema mientras tanto puede contar los enemigos que existen, así como generar la música de combate.

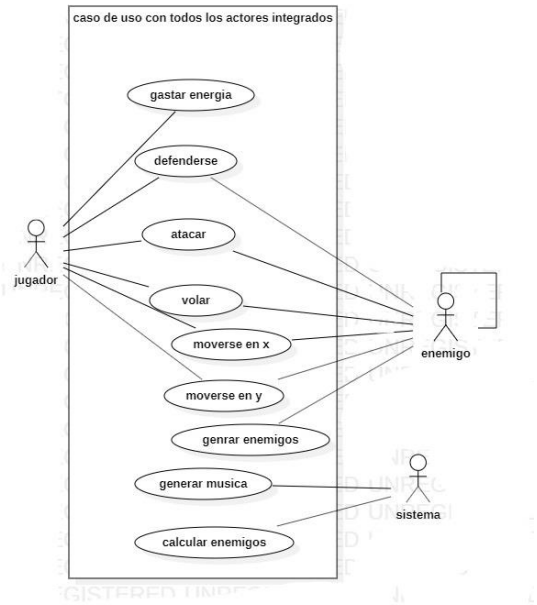

Ilustración 10: Ejemplo Caso de Uso. Fuente: Propia.

Como se ha escrito para este diagrama el hecho de que el sistema esté como un actor es un error a la hora de realizar un gráfico, por consiguiente, el siguiente diagrama nos muestra como el sistema pasa de ser actor a parte del mismo diagrama manteniendo las mecánicas con las cuales se dan los requisitos.

En el mostrado en la ilustración once (11) se nos muestra como el jugador y el jefe final pueden defenderse, moverse, volar, y disparar; se evidencia como al defenderse con el personaje jugable gasta energía así mismo si es disparado mientras realiza esta acción se desaparecerá el impacto, sin importar cuál de los dos se esté cubriendo, solo hay dos actores ya que el sistema se ha integrado.

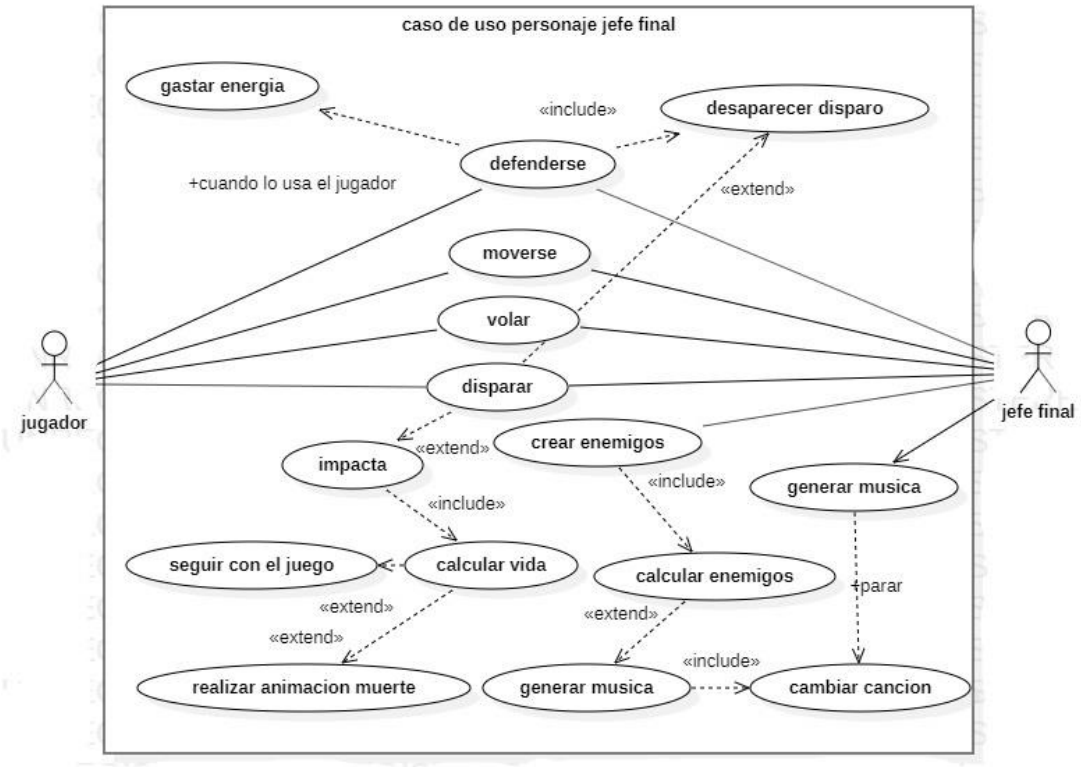

Ilustración 11: Ejemplo 2 caso de uso. Fuente: Propia.

Se puede ver cómo mientras esté el jefe final en el escenario el sistema generara música, además el jefe podrá crear enemigos, al crear estos enemigos de forma automática el sistema generara otra música lo que hará que se cambie la canción del jefe final y la de este pare. Si al disparar alguno de los dos el otro resulta impactado automáticamente se calculará la vida de los personajes y con base en esto se sabrá si se sigue con el juego o alguno debe realizar la animación de la muerte.

Diagramas de objetos

Definición

Es una instantánea que define la relación de los objetos en la ejecución del sistema UML 2.0.

Según el libro UML 1.3 se define “Un diagrama de una instantánea es una imagen de un sistema, en un instante en el tiempo. Debido a que contiene imágenes de objetos, se llama diagrama de objetos. Puede ser útil como ejemplo del sistema, por ejemplo, ilustrar las estructuras de datos complicadas o mostrar el comportamiento con una secuencia de instantáneas en un cierto plazo (…). Recuerde que todas las instantáneas son ejemplos de sistemas, no definiciones de sistemas. La definición de la estructura y del comportamiento del sistema se encuentra en las vistas de definición, y construir las vistas de definición es el objetivo del modelado y el desafío.” (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

Funcionamiento

De facto se espera que se utilice para modelar la estructura, esto se comienza a hacer en el momento en que se toma la relación en una determinada interacción y el análisis de la utilización dando como resultado la documentación, construcción, especificación además de visualización de la existencia de estos o instancias dentro del sistema, demostrando las relaciones entre estas.

En una ontología para la representación de conceptos de diseño de software se señala que: “Son utilizados durante el proceso de Análisis y diseño de los sistemas informáticos en la metodología UML. Los diagramas de objetos utilizan un subconjunto de los elementos de un diagrama de clase.” (Gloria, Acevedo, & Moreno, 2011).

Elementos

Objeto: no es solo la representación de **algo tangible**, ya que es especialmente útil para la representación de datos de un sistema, pero no tanto para el modelado o estructuración de una base de datos, pueden ser además datos de todo tipo, como el área, o un mundo que represente esta área.

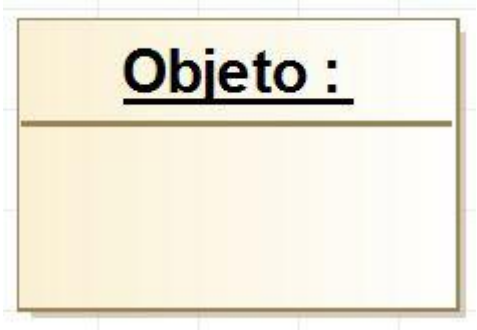

Ilustración 12: elemento Objeto. Fuente: Propia.

Clase: Todo objeto hace referencia a una **clase**, esta es la **agrupación** de uno o varios en un contexto simbólico, lo cual es beneficioso para el desarrollo de otros diagramas especialmente el de clases.

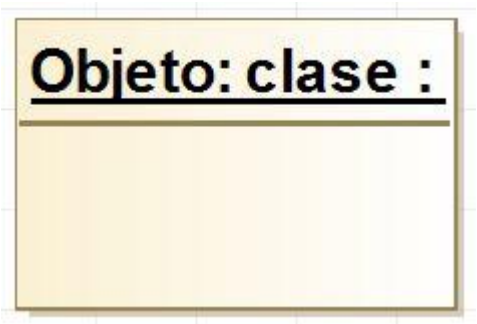

Ilustración 13: Elemento Clase. Fuente: Propia.

Atributos: Los objetos poseen entre ellos **atributos** lo cual significa las **propiedades** de este.

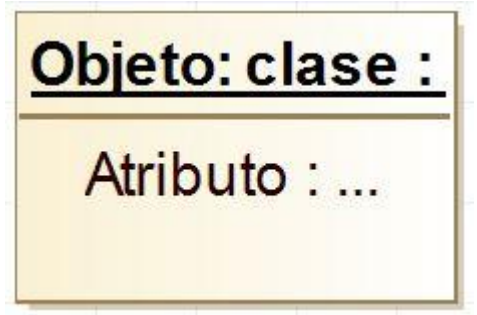

Ilustración 14: Elemento Atributo. Fuente: Propia.

Enlace: es **la relación** entre dos objetos esta puede **llevar un verbo** con tiempo presente el cual describa la relación que se da entre estos dos.

Ilustración 15: Elemento Enlace. Fuente: Propia.

Enlace directo: Un **enlace directo** expresa una **relación** entre objetos **que siempre se da** sin importar otros contextos.

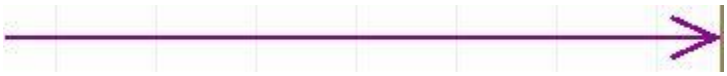

Ilustración 16: Elemento Enlace Directo. Fuente: Propia.

Cantidad: La relación entre objetos puede tener **diferentes cantidades**, como de uno a uno el cual puede no ser expresado con números, de uno a muchos expresado como el "1…*" y de varios a muchos expresado con "*…*", esta relación depende del contexto de la relación.

Ilustración 17: Elemento Cantidad. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el ejemplo que se evidencio en los diagramas de caso de uso en el cual se da el desarrollo de un prototipo para un videojuego, en el cual se tienen que desarrollar los prerrequisitos del cual se pueden sacar varios objetos que conforman al sistema.

En el siguiente diagrama mostrado en la ilustración diez y ocho (18) se pueden evidenciar los elementos, sus atributos y clases a las que pertenecen, se puede ver que hay dos grupos de estos los cuales son los del personaje jugable así como los del jefe final, el

personaje jugable genera un campo de fuerza que esta evidenciado en los prerrequisitos además este sostiene una vara mágica que genera una partícula con la cual se ataca.

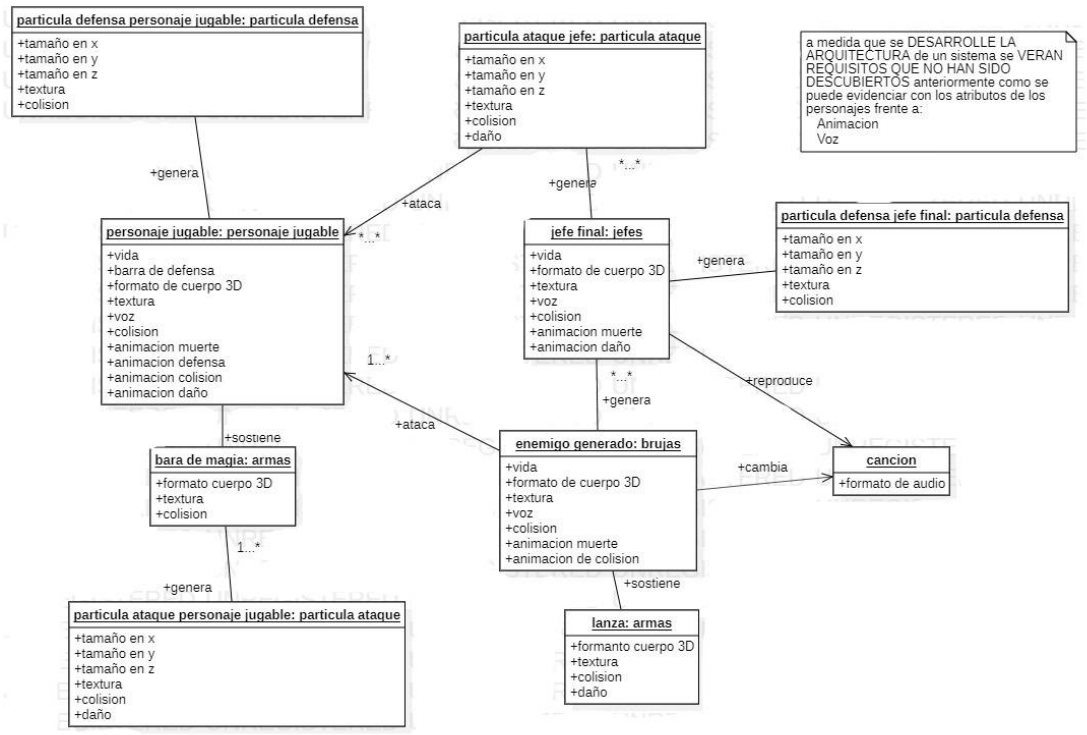

Ilustración 18: Ejemplo diagrama de objetos. Fuente: Propia.

El jefe final por su parte genera un campo de fuerza además de las partículas de ataque para el personaje jugable, también a los enemigos generados, estos tendrán una lanza con los cual será que ataquen al personaje jugable, además el jefe final genera una melodía que cuando este genere varios enemigos la canción cambia por otra.

En el diagrama se puede evidenciar que el personaje jugable pertenece a la clase de personaje jugable, tiene su vida, barra de defensa, formato para su cuerpo 3D, textura, voz, colisión con los demás, animación de su muerte, defensa, colisión, pero también

daño, en cuanto al arma que sostiene pertenece a las armas, tiene un formato para el cuerpo 3D, la textura además de una colisión.

La partícula de ataque del personaje jugable pertenece a partícula de ataque el cual tiene su tamaño en tres dimensiones ya que es tridimensional, una textura, una medición de colisión y un daño que provoca cuando alcanza a algún enemigo; la partícula de defensa del personaje jugable comparte muchas características de la partícula de ataque, debido a que tienen una naturaleza similar cambiando en su tipo la cual pertenece a partícula de defensa.

Debido a las capas de abstracción se puede ver que el jugador a través del personaje jugable cual a través de su varita puede lanzar entre una a varias partículas. Por otra parte entre uno o varios enemigos generados atacaran al personaje jugable así como varias a muchas partículas lo atacaran; el jefe final sin importar si genera enemigos o partículas generara varios o muchos de estos.

Se puede apreciar que los objetos que comparten clase tienen las mismas características o unas muy similares, esto se debe a que siempre tiene las propiedades o un valor nulo de ellas, como se puede apreciar con los de las “armas” en los cuales la vara el personaje jugable no tiene un valor de daño; para el valor de las partículas tanto de defensa como de ataques sin importar cual sea mantiene sus propiedades.

A medida que se desarrollen los requisitos del sistema se encontraran que algunos requisitos pedidos al inicio necesitan de otros requisitos que hasta ese momento estuvieron ocultos, como se puede apreciar con las armas del personaje jugable y del

enemigo generado, para el personaje jugable se evidencia que el ataque no sale de él sino de su vara además se sabe con qué cosa atacaran los enemigos generados; aparte se le dan animaciones que no fueron previamente documentadas como las de colisión, daño y defensa o solo la de colisión respectivamente.

Diagrama de actividad

Definición

Según James Rumbaugh, Ivar Jacobson & Graddy booch en el libro El lenguaje unificado de modelado manual de referencia (2007), lo define a grandes rasgos como la representación de la ejecución de los estados de cómputo y no como la del estado de un objeto.

En el libro UML 1.3 se describe lo siguiente: “Un diagrama de actividades es la notación para un grafo de actividades (…). Incluye algunos símbolos especiales abreviados por conveniencia. Estos símbolos pueden usarse en cualquier diagrama de estados, aunque mezclar la notación puede ser molesto.” (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

En una ontología para la representación de conceptos de diseño de software se señala que: “Representa los flujos de trabajo paso a paso de negocio y operacionales de los componentes en un sistema. Un Diagrama de actividades muestra el flujo de control general.” (Gloria, Acevedo, & Moreno, 2011).

De una manera poco detallada se describe en Ingeniera de software I:

“Representación del comportamiento dinámico del sistema mediante actividades

• Un diagrama de actividad representa el comportamiento mediante un modelo de flujo de datos y flujo de control

• Actividad: especificación de un comportamiento parametrizado que se expresa como un flujo de ejecución por medio de una secuencia de unidades subordinadas

• Acción: especificación de una unidad fundamental de comportamiento que representa una transformación o procesamiento

• Las acciones están contenidas en actividades que le proporcionan su contexto

• Los diagramas de actividad capturan las acciones y sus resultados

• Los pasos de ejecución dentro de una actividad pueden ser concurrentes o secuenciales

• Una actividad involucra constructores de sincronización y de bifurcación". (García, Moreno, & García, 2018).

Funcionamiento

Se representan las acciones sin ninguna interrupción externa, los grafos contienen un estado (actividad) el cual es representado por una sentencia en la fluctuación de un procedimiento, en pocas palabras una actividad correspondiente a un punto de trabajo y una vez termina pasa a la siguiente dentro del mismo.

Elementos

Acción: Es la **presentación** de una **actividad** en un sistema la cual puede incluir a un actor de caso de uso o no.

Ilustración 19: Elemento Acción. Fuente: Propia.

Inicio: El inicio es la presentación **del inicio del sistema** esta **antes que cualquier acción** y se dirige hacia la primera de estas para el caso de uso del que se desarrolle el diagrama.

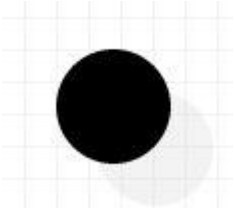

Ilustración 20: Elemento Inicio. Fuente: Propia.

Final: todo diagrama debe de presentar un inicio y un fin, el final del diagrama se presentará **después de la última acción**, se debe de tener uno o varios finales ya que no se debería de poder ejecutar eternamente.

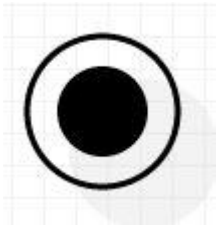

Ilustración 21: Elemento Final. Fuente: Propia.

Flujo de control: El flujo de control muestra de una **manera como se ejecuta**, el dominio de este permite **señalar la dirección del sistema**, auto repeticiones y como se ejecutará acción por acción.

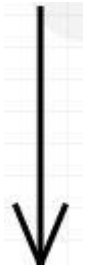

Ilustración 22: Elemento Flujo de control. Fuente: Propia.

Bifurcacion/Union: Hay ciertos sistemas como casos de sistemas en las que una acción conlleva a dos acciones o más en paralelo, para estos casos se usa la bifurcación o **fork** con la cual se evidencia que tendrá dos o más **actividades paralelas** y después se volverán a unir en caso de que presente tal característica. Mientras el fork vuelve una línea en varias el join vuelve varias en una.

Ilustración 23: Elemento Bifurcación. Fuente: Propia.

Decisión/unión: La decisión y unión de esta a diferencia de la bifurcación cual está hecha para que se tomen acciones cuando la separación no es paralela, es decir que el sistema o usuario **toma una acción u otra**, pero **nunca las dos** al mismo tiempo. Es mal visto que una Decisión sea cerrada con una unión de bifurcación.

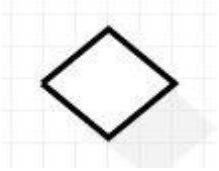

Ilustración 24: Elemento Unión. Fuente: Propia.

Actor: Es una **partición** del sistema en la cual se **relatan las acciones** que realiza este o alguno de los **actores** identificados en los casos de uso, se pueden poner de manera tanto horizontal como vertical. Tiene la forma de una tabla sin embargo esta no está cerrada por abajo o por su derecha dependiendo de la posición.

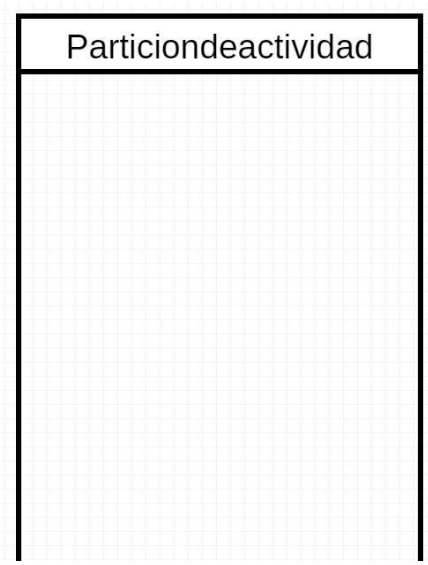

Ilustración 25: elemento Actor. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en los diagramas de caso de uso en el cual se da el desarrollo de un prototipo para un videojuego, en el cual se tienen que desarrollar los prerrequisitos del cual se pueden sacar varias actividades que lo conforman.

Como se puede evidenciar en la ilustración veintiséis (26) referente al presente diagrama hay tres actores, el personaje jugable, el jefe final y el sistema; el jefe final

comienza al activar la canción sin embargo no solo él puede finalizar con tal, cual se dará cuando uno de los dos muera.

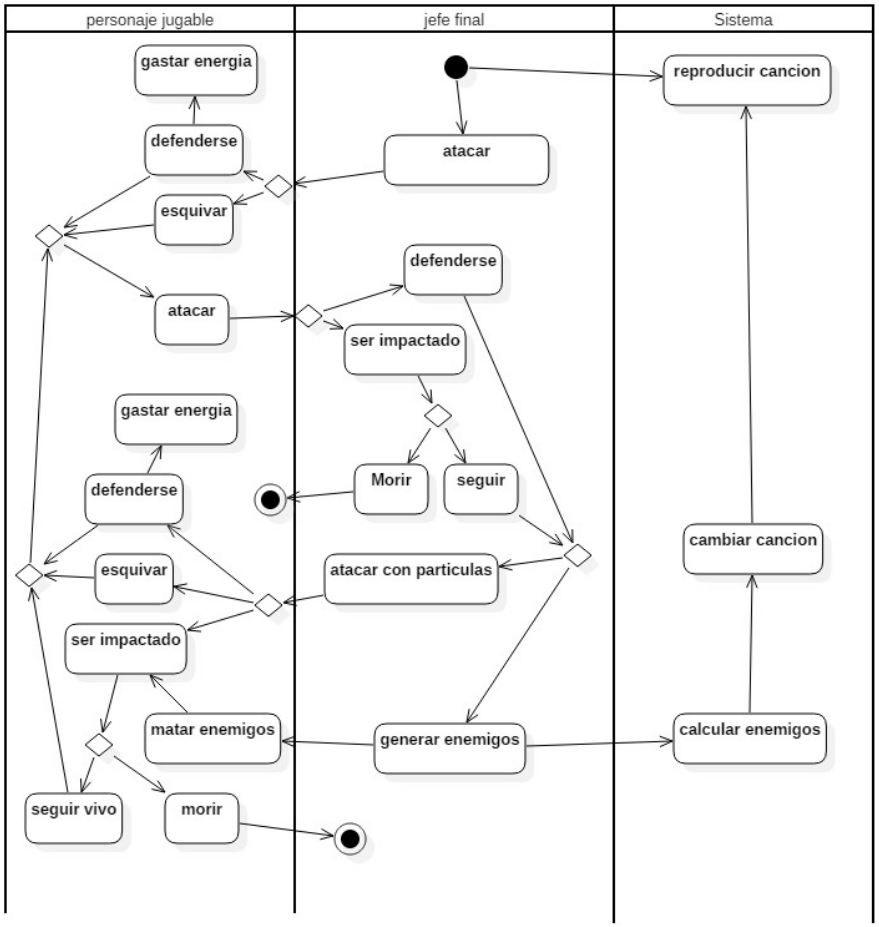

Ilustración 26: Ejemplo diagrama de Actividad. Fuente: Propia.

Después de que el jefe final ataque el personaje jugable tendrá la decisión de defenderse o esquivar el ataque para posteriormente realizar el mismo, el jefe también podrá defenderse o podría llegar a ser impactado lo que podría llevar a que muera o no, en caso de que no muera podrá decidir si atacar con partículas o generar enemigos, el personaje jugable por su parte podrá atacar a los enemigos generados, o defenderse además de esquivar las partículas que se le lancen.

En caso de que siga vivo el jugador podrá volver a atacar; si se defiende se gasta energía como se ha venido evidenciando desde prerrequisitos. El sistema por su parte calculara la cantidad de enemigos cambiando la canción cuando se generen lo que hará que se reproduzca por la canción que acaba de cambiar.

Diagrama de estado

Definición

Son técnicas para conocer el estado de un programa, describen un objeto y como puede cambiar a otro y está basado en solo una clase en particular.

En el libro UML 1.3 se lee en el primer párrafo sobre este apartado: “La vista de la máquina de estados describe el comportamiento dinámico de los objetos, en un cierto plazo, modelando los ciclos de vida de los objetos de cada clase. Cada objeto se trata como una entidad aislada que se comunica con el resto del mundo detectando eventos y respondiendo a ellos. Los eventos representan las clases de cambios que un objeto puede detectar: la recepción de llamadas o seriales explicitas desde un objeto a otro, un Cambio en ciertos valores, o el Paso del tiempo. Cualquier cosa que pueda declarar a un objeto se puede caracterizar como evento. Los sucesos del mundo real se modelan como señales del mundo exterior al sistema.” (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

En Ingeniería de software I se muestra como características: “Las máquinas de estados describen los estados que un objeto puede tener durante su ciclo de vida, el comportamiento en esos estados y los eventos que causan los cambios de estado

• UML define dos tipos de máquinas de estados:

• De comportamiento: capturan los ciclos de vida de los objetos, subsistemas y sistemas.

• De protocolo: se usan para especificar las transformaciones legales que pueden ocurrir en un clasificador abstracto como una interfaz o un puerto (protocolos de uso).” (García, Moreno, & García, 2018).

Funcionamiento

Si un evento no tiene una etiqueta en su transición significa que termina tan pronto como se termine la comprobación: una condición lógica solo es verdadera o falsa, luego dado que el estado cambia si el guardia es verdadero.

En una ontología para la representación de conceptos de diseño de software se señala que: “Es un diagrama utilizado para identificar cada una de las rutas o caminos que puede tomar un flujo de información luego de ejecutarse cada proceso. Identifica bajo qué argumentos se ejecuta cada uno de los procesos y en qué momento podrían tener una variación, además permite visualizar de una forma secuencial la ejecución de cada uno de los procesos.” (Gloria, Acevedo, & Moreno, 2011).

Elementos

Estado: se representa con un **verbo gerundio o participativo** se describe el **estado de un objeto** cuando se da **determinada acción o actividad.** Está representado por un cuadro redondeado el cual tiene escrito dentro de él con el verbo la intención a representar.

Ilustración 27: Elemento Estado. Fuente: Propia.

Inicio: El diagrama siempre comienza con el inicio, está representado por una **bola negra rellenada** la cual se pone **antes del primer estado.**

Ilustración 28: Elemento Inicio. Fuente: Propia.

Fin: Todo diagrama debe de tener un final para que no se ejecute indeterminadamente por tanto **representando el final**, es hecho por un círculo negro relleno con una aureola.

Ilustración 29: Elemento Fin. Fuente: Propia.

Bifurcación/unión: Representa que **se divide en dos estados o más** separados que se dan al mismo tiempo o **en paralelo**, al finalizar estos dos estados separados puede

que se dé la condición para que se vuelvan a unir por tanto se un cierre con la unión paralela. Se representa como una barra negra.

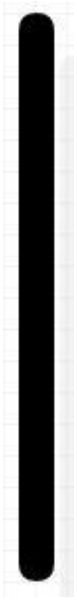

Ilustración 30: Elemento Bifurcación. Fuente: Propia.

Decisión: La decisión de estados es cuando lleva a dos o más actividades del sistema que no se presentan en paralelo haciendo que se use este recurso. Su forma es un romboide simple.

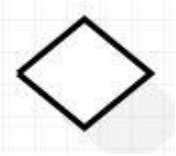

Ilustración 31: Elemento Decisión. Fuente: Propia.

Transición: es lo que lleva de un estado a otro cuales **llevan escritas la actividad** que hizo que **cambiara** tal, está representada como una simple flecha abierta y se usa apuntado a lo que transmutará el objeto a continuación.

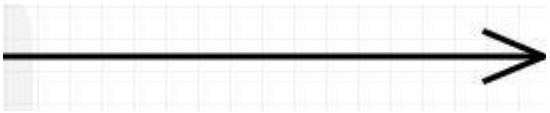

Ilustración 32: Elemento Transición. Fuente: Propia.

Auto transición: Se da cuando un estado llevas a una acción la cual **termina en él mismo**, esta **acción** puede llegar a **ser distinta** con la que se entró **o igual**, está representado como una flecha abierta que se dirige a sí misma.

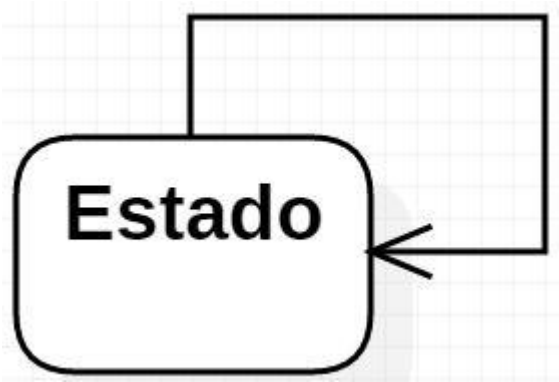

Ilustración 33: Elemento Auto transición. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de actividades cual se da el desarrollo de un prototipo para un videojuego, en el cual se tienen que desarrollar los prerrequisitos del cual se pueden sacar varios estados que conforman al sistema.

En el grafico presentado en la ilustración treintaicuatro (34) se evidencia como no puede diferenciar entre objetos o actores haciendo que el intercambio dentro del sistema se tenga que percatar, comenzando entonces con el jefe final quien ataca al personaje jugable terminando con “mortuorio” de alguno de los dos.

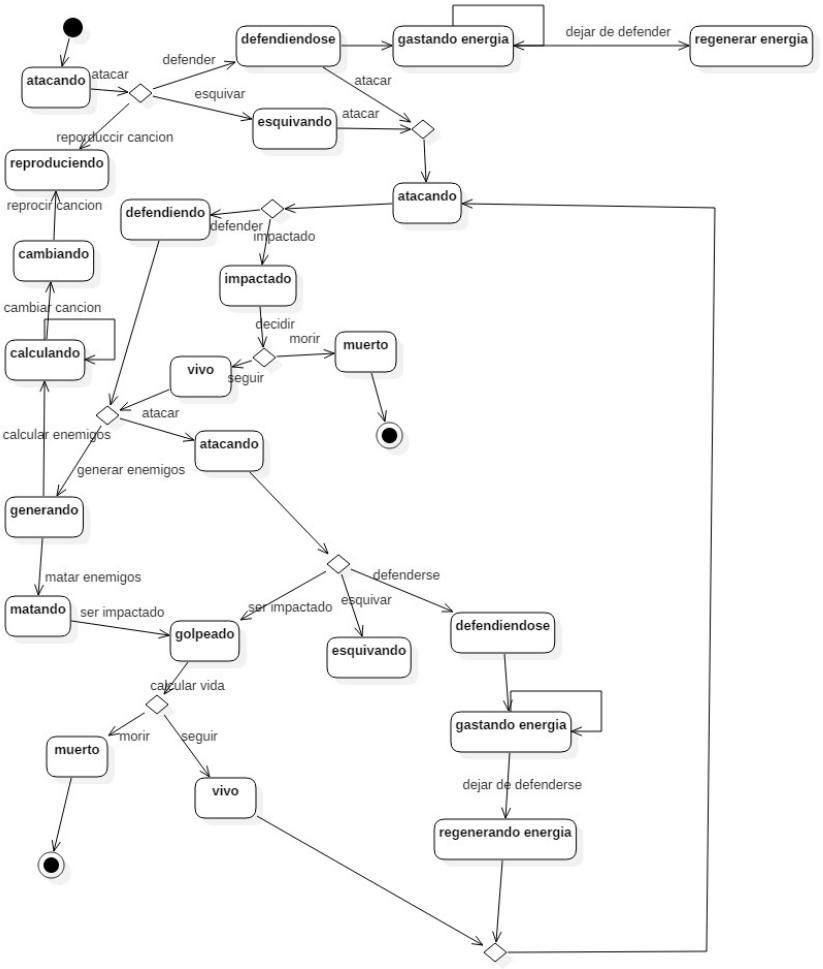

Ilustración 34: Ejemplo Diagrama de estado. Fuente: Propia.

Al jefe estar atacando altera al objeto personaje jugable el cual cambia sus estados a desentendiéndose o esquivando, si se defiende está gastando energía dando que al no hacerlo comienza a regenerar esa energía. El PJ puede estar atacando cuando lo haga el líder se estará defendiendo o será impactado de serlo se decide si sigue vivo o muerto y en caso de que muera el juego termina.

El jefe final (JF) podría estar generando monstruos o atacando, si ataca el jugador tiene tres posibles fases: ser golpeado, estar esquivando o defendiéndose, al ser golpeado se calculará la vida así sabiendo si está muerto o no, en caso de que no lo esté el juego continuara haciendo que pueda volver a atacar. El personaje jugable (PC por sus silabas en inglés) tendrá un estado de matando el cual se dará cuando mate monstruos.

El sistema estará calculando a los enemigos en tiempo real, al contar más de un enemigo cambiará la canción y la reproducirá en vez de la que ya se estaba produciendo al notar al jefe final en escena.

Diagrama de clases

Definición

Describe las diversas clases estáticas que existen entre objetos de un sistema, estas pueden ser por asociación y por sub tipos. En los diagramas también se muestran los atributos y operaciones, así como sus restricciones esto debido a la forma de interconexión entre ellas.

En el libro UML gota a gota se describe en el segundo párrafo dedicado: “El diagrama de clase, además de ser de uso extendido, también está sujeto a la más amplia gama de conceptos de modelado. Aunque los elementos básicos son necesarios para todos, los conceptos avanzados se usan con mucha menor frecuencia. Por eso, he dividido mi estudio
de los diagramas de clase en dos partes; los fundamentos (…) y los conceptos avanzados (…).” (Flower & Scott, 1999).

En una ontología para la representación de conceptos de diseño de software se señala que: “Es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el desafío conceptual de la información que se manejará en el Sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro.” (Gloria, Acevedo, & Moreno, 2011).

En Ingeniería de software I se muestra como características: “Una clase es un clasificador que describe un conjunto de objetos que comparten la misma especificación de características, restricciones y semántica • Una clase describe las propiedades y comportamiento de un grupo de objetos (…). Diagramas de clases: describen la vista estática de un sistema en forma de clases y relaciones entre ellas.” (García, Moreno, & García, 2018).

Funcionamiento

Hay tres perspectivas desde las que se pueden manejar los gráficos: conceptual, de especificación e implementación, en el primero se dibuja un diagrama que represente los conceptos que se están estudiando, en el segundo se deben de observar las interfaces de software no su implementación, en el tercero y último al ya tener las clases se expone por completo la implementación estas en el diagrama.

Al dibujarse un diagrama se debe de usar una sola perspectiva y al leerlo tal perspectiva debe ser tenida en cuenta.

Elementos

Clase: Está dada por los atributos y propiedades de un **conjunto de objetos** que pudieron ya haber sido descritos en el diagrama, en estos se nos muestra la abstracción de los objetos y por lo general tienen **nombre de sustantivo colectivo,** aunque pueden tener cualquier nombre que sea capaz de describir los elementos.

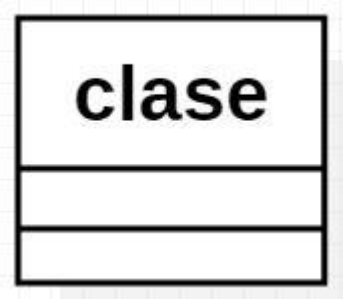

Ilustración 35: Elemento Clase. Fuente: Propia.

Atributo: Está dentro de la clase y se usa para dar una **descripción física o conceptual** que se describe.

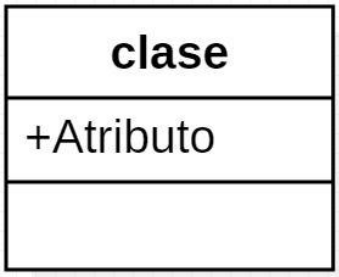

Ilustración 36: Elemento Atributo. Fuente: Propia.

Propiedad: es la **acción** que ejecuta una clase, al igual que el atributo pueden ser una o varias acciones las cuales se pueden ejecutar en paralelo o no y están definidas por un **verbo** cuando no representa una **función** en un programa real.

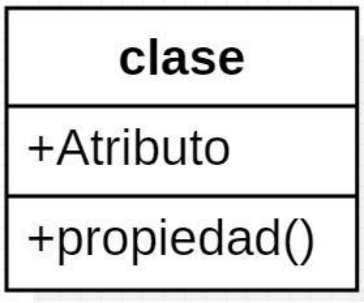

Ilustración 37: Elemento propiedad. Fuente: Propia.

Interfase: Es la representación de la interfase y **únicamente la interfase de un programa**, se le anida la clase de interfase correspondiente y esta clase si conecta con el resto del programa.

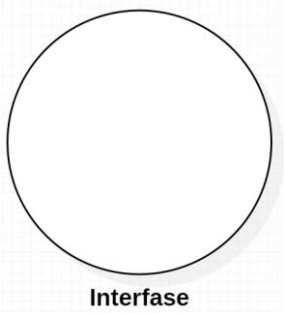

Ilustración 38: Elemento Interfase. Fuente: Propia.

Asociación: Muestra que dos clases o más **comparten sus atributos y propiedades al mismo tiempo**, es la forma más sencilla de describir la conexión entre clases, se dibuja como una simple línea.

Ilustración 39: Elemento Asociación. Fuente: Propia.

Asociación directa: Se muestra que una clase está directamente **siempre se asocia con otra clase** y se dibuja como una flecha con la punta abierta, la señalización declara que clase A se asocia a clase B de tal forma que es B quien es asociada.

Ilustración 40: Elemento Asociación directa. Fuente: Propia.

Dependencia: se da cuando **una clase usara de la que se señala**, está dada como una flecha con la punta abierta la cual está entre cortada en varias partes iguales.

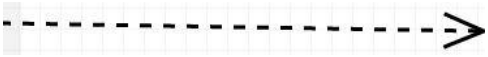

Ilustración 41: Elemento Dependencia. Fuente: Propia.

Agregación: Se da cuando **una clase usa otra**, pero esa clase **no depende de ella** por tanto si desaparece o sufre cambios, la señalada que le usa no desaparecerá; se muestra como un romboide sin rellenar con una línea.

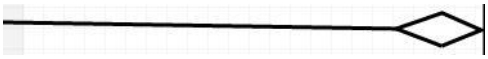

Ilustración 42: Elemento Agregación. Fuente: Propia.

Composición: se da cuando **una clase usa otra y depende de ell**a por tanto si desaparece o sufre cambios, la señalada que le usa también desaparecerá; se diagrama como un romboide rellenado con una línea.

Ilustración 43: Elemento Composición. Fuente: Propia.

Generalización: Se da cuando una clase **hereda** los atributos y propiedades de otra por tanto todos los valores y funciones dentro de la clase madre pasaran a la clase hija. Se dibuja como una flecha con la punta cerrada.

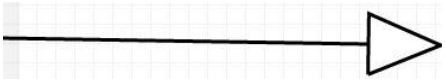

Ilustración 44: Elemento Generalización. Fuente: Propia.

Relaciones de cantidad: hay ciertas clases que se multiplican a si mismas al tener relación con otras clases, para describir este fenómeno se usa la relación de cantidad el cual **puede ir de 0 a muchos** en caso de que la relación sea de 1 a 1 no se declara la relación.

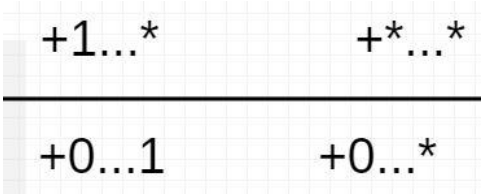

Ilustración 45: Elemento Relación cantidad. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de objetos y actividades en los cuales se da el desarrollo de un prototipo para un videojuego, en el cual

se tienen que desarrollar los prerrequisitos del cual se pueden sacar varios estados que conforman al sistema.

En el grafico mostrado en la ilustración Cuarenta y seis (46) se nos presenta las relaciones de las clases principales personaje jugable, jefes y brujas, los cuales son hijas de personaje en la cual se nos muestra que allí se programará la vida, formato 3D, textura, voz, colisión, animación de daño además de muerte, esto podrán perder vida, hablar, morir, moverse y volar.

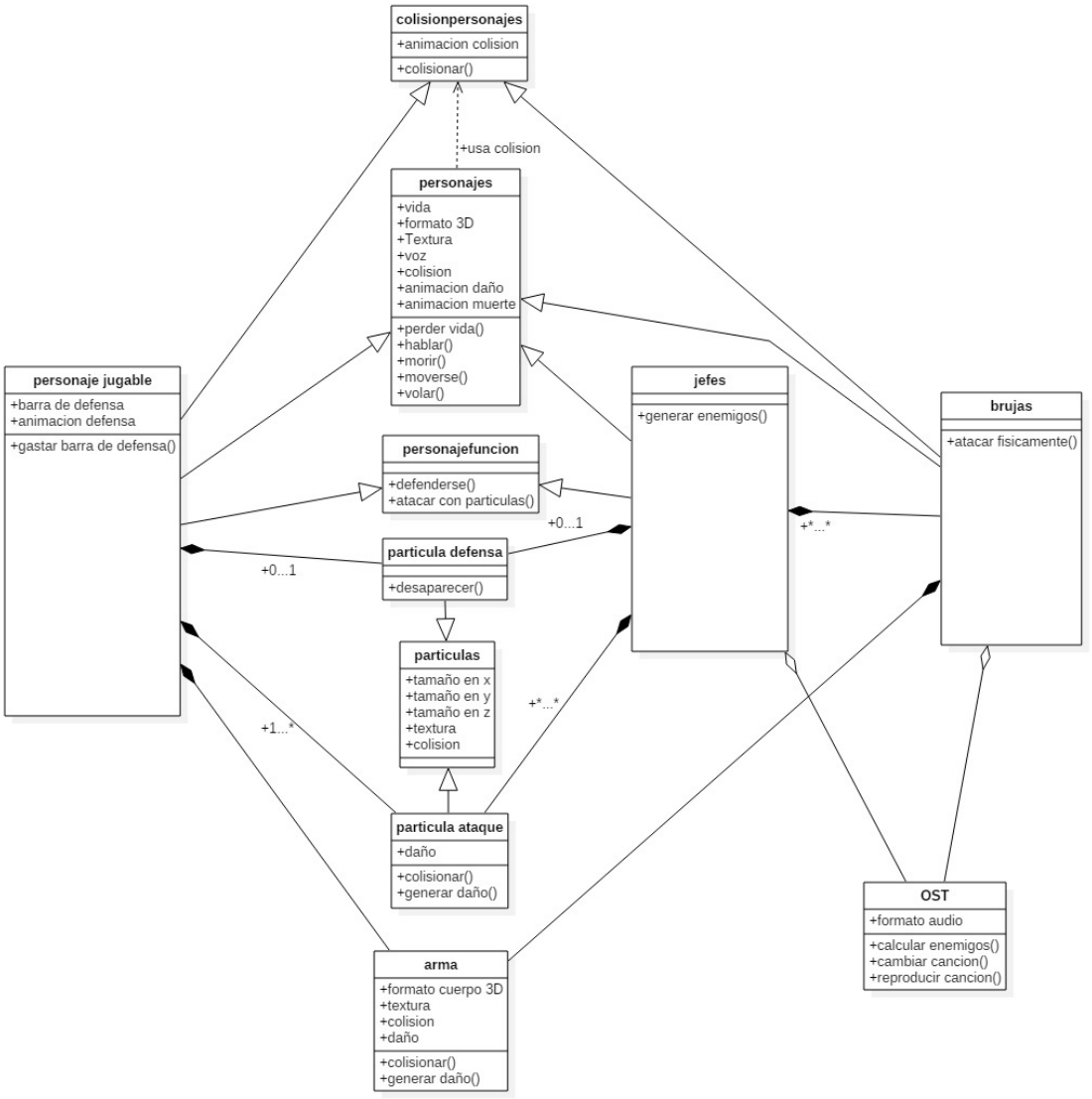

Ilustración 46: Ejemplo Diagrama de Clase. Elaboración propia. Fuente: Propia.

Las clases mencionadas en el párrafo anterior tendrán funciones propias las cuales son para el caso del jefe generar enemigos, para la bruja atacar físicamente, el personaje jugable tendrá barra de defensa, animación de defensa también gastará la barra de defensa.

Las partículas de defensa y ataque toman de partículas todos sus atributos por otra parte las armas tendrán un formato 3D, textura, colisión, daño además generarán el daño tal como podrán colisionar. La relación demuestra que todas las clases con las que tiene relación el personaje jugable, jefes así como brujas desaparecen, pero las clases se mantienen.

OST existirá así no hayan enemigos de ningún tipo, este tendrá un formato de audio, calculara los enemigos, cambiara la canción también reproducirá esta. Personajefuncion tendrá la función defenderse, pero también atacar con partículas Colisionpersonajes tendrá programada la colisión de personajes e igualmente colisionar.

Diagramas de componentes

Definición

Estos componentes se denotan en interfaces las cuales son una notación en donde se expresa este, entradas o servicios del mismo aun así, dejando libres a otros iguales dentro del sistema y clases.

El libro UML 1.3 lo describe más detalladamente: “La vista de implementación muestra el empaquetado físico de las partes reutilizables del sistema en unidades sustituibles llamadas componentes. Una vista de implementación muestra la implementación de los elementos del diseño (tales como clases) mediante componentes, así como sus interfaces y dependencias entre componentes. Los componentes son las piezas reutilizables de alto nivel a partir de las cuales se pueden construir los sistemas.” (Rumbaugh, Jacobson, & Booch, Lenguaje Unificado de Modelado Manual de Referencia UML 1.3, 2000).

Funcionamiento

En el escrito de Geoffrey Spark sobre una introducción a UML (2018) se nos revela en tal apartado que estos gráficos son la demostración de comunicación, ubicación, así como otras condiciones sobre los componentes de un sistema.

En una ontología para la representación de conceptos de diseño de software se señala que: “Representa como un Sistema de software es dividido en componentes y muestra las dependencias entre estos componentes. Los componentes físicos incluyen

archivos, cabeceras, bibliotecas compartidas, módulos, ejecutables o paquetes.” (Gloria, Acevedo, & Moreno, 2011).

López y Ruiz lo mencionan de esta forma: “-Describen la estructura del software mostrando la organización y las dependencias entre un conjunto de componentes.
-Pueden representar la encapsulación de un componente con sus interfaces, puertos y estructura interna (posiblemente formada por otros componentes anidados y conectores).
-Cubren la vista de implementación estática del diseño de un sistema.” (Riuz & López, 2020).

Elementos

Componente: la mayoría de las veces es representado como una especie **de clase o sub clase** la cual **demuestra** por lo general un **conjunto de objetos** dentro del sistema.

Ilustración 47: Elemento Componente. Fuente: Propia.

Artefacto: es la representación de **ficheros** cono los que trabaja el sistema, estos deben tener una connotación un tanto externa al sistema como una factura o un archivo de texto.

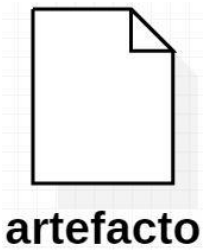

Ilustración 48: Elemento Artefacto. Fuente: Propia.

Interfase: hace parte del **módulo** con la que un componente se **comunica** con otros iguales.

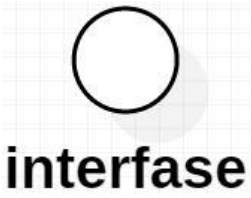

Ilustración 49: Elemento Interfase. Fuente: Propia.

Puerto: es la **entrada y salida de comunicaciones** de los **objetos** de un componente con otros iguales.

Ilustración 50: Elemento Puerto. Fuente: Propia.

Objeto: es la representación de que objetos **están dentro** de diferentes **componentes** estos objetos por lo general han sido descritos en el esquema de objetos, con varios dentro pueden llegar a tener **relación entre ellos.**

Ilustración 51: Elemento Objeto. Fuente: Propia.

Dependencia: demuestra que un componente tiene una **existencia dependiente** de al que señala.

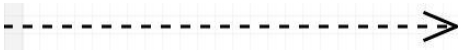

Ilustración 52: Elemento Dependencia. Fuente: Propia.

Realización: demuestra que un componente, interfase o artefacto **realiza acciones no especificas** con otro.

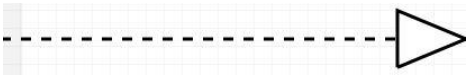

Ilustración 53: Elemento Realización. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de objetos y clases en los cuales se da el desarrollo de un prototipo para un videojuego, para eso se tienen que desarrollar los prerrequisitos donde se sacaron varios componentes que conforman al sistema.

En la ilustración cincuenta y cuatro (54) se puede ver el diagrama el cual consta de cinco de ellos: objetos personaje jugable, objetos bruja, personajes, partículas jefe y voces, tiene un total de ocho artefactos que son usados por estos además de siete objetos cuales tienen comunicación entre ellos.

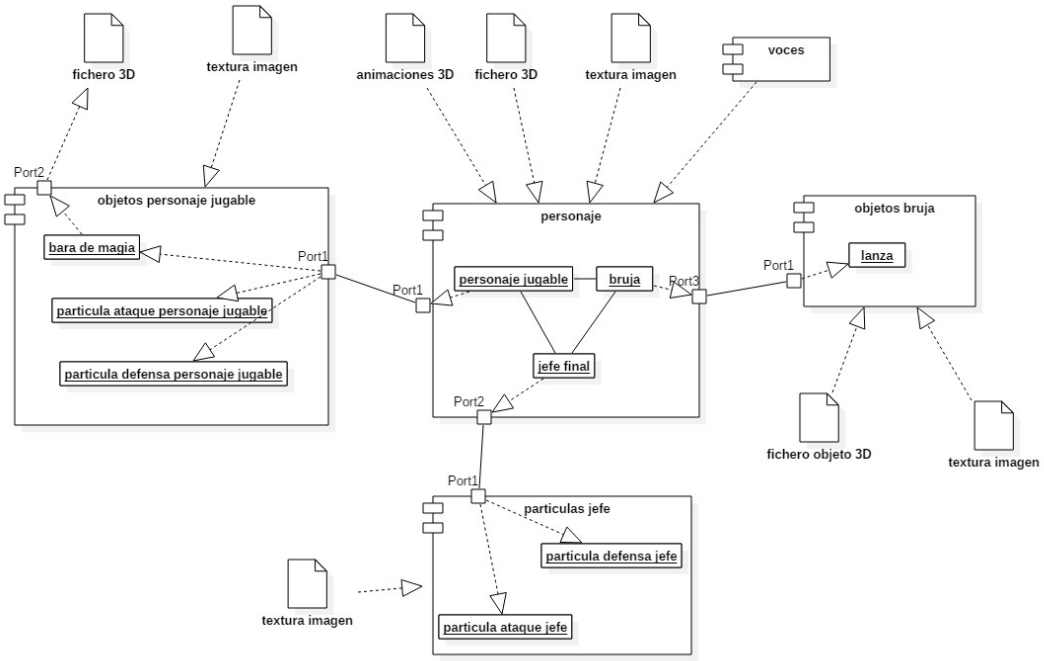

Ilustración 54: Ejemplo Diagrama de componentes. Fuente: Propia.

El personaje tiene dentro de el al personaje jugable, jefe final y bruja, esta se conecta con objeto bruja al hacer uso de la lanza, el jefe final por su parte usa de partícula de defensa y ataque propias, al PJ le pasa algo similar cambiando por la barra de magia

cual no se debe confundir con una posible interfase “barra de magia”, solo este objeto usa un fichero 3D, pero todos usan texturas.

Personaje usa animaciones en 3D, ficheros 3D, una imagen de textura y un componente no aclarado llamado voces el cual estará lena de las voces de los personajes, objetos bruja también usa formato 3D y textura, aunque partículas de jefe solo usa una textura.

Diagramas de despliegue

Definición

El agregar nodos dentro de un paquete de componentes lo vuelve un diagrama de despliegue, Cada uno de estos representa en su mayoría piezas de hardware, siendo este un dispositivo, sensor simple o un mainframe.

En el El Lenguaje Unificado De Modelado Manual De Referencia UML 2.0 se da la siguiente definición: “La vista de despliegue muestra la disposición física de los nodos. Un nodo es un recurso computacional de ejecución, como computadoras u otros dispositivos. Durante la ejecución, los nodos pueden contener artefactos, entidades físicas como los archivos. La relación de manifestación muestra la relación entre un elemento de diseño, como un componente, y los artefactos que los plasman en el sistema software. La vista de despliegue puede resaltar los cuellos de botella en el rendimiento debidos a la ubicación de los artefactos que manifiestan componentes independientes en diferentes nodos.” (Rumbaugh, Jacobson, & Booch, El Lenguaje Unificado de Modelado Manual de Referencia 2.0, 2007).

Funcionamiento

En Ingeniería de software I se muestra como características: “La vista de despliegue representa el despliegue de artefactos de tiempo de ejecución sobre nodos.
• Los diagramas de despliegue, junto con los diagramas de componentes, forman parte de la arquitectura física.

• La arquitectura física es una descripción detallada del sistema que muestra la asignación de artefactos de software a nodos físicos.” (García, Moreno, & García, 2018).

En el libro de UML gota a gota de Martin Fowler y Kendall Scott (1999), se relata que es aquel en el que se muestran las relaciones físicas de los componentes, así como a nivel de software puesto es bueno para demostrar cómo se comunican y conectan diversos sistemas.

En una ontología para la representación de conceptos de diseño de software se señala que: “Es un tipo de diagrama del Lenguaje Unificado de Modelado que se utiliza para modela el hardware utilizando en las implementaciones de sistemas y las relaciones entre Sus componentes.” (Gloria, Acevedo, & Moreno, 2011).

Elementos

Nodo: Es la representación de un **sistema** dirigido al usuario o autónomo y más frecuentemente de un **hardware**, está representado como una caja a la cual se le agrega el nombre.

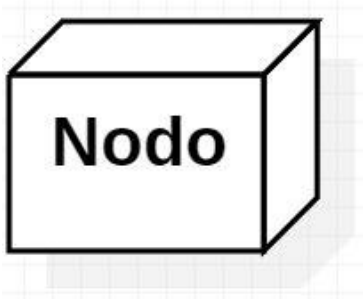

Ilustración 55: Elemento Nodo. Fuente: Propia.

Desplegar: Deploy es una indicación la cual da a entender que un **nodo usa** de una manera **dependiente y no aclarada** a otro.

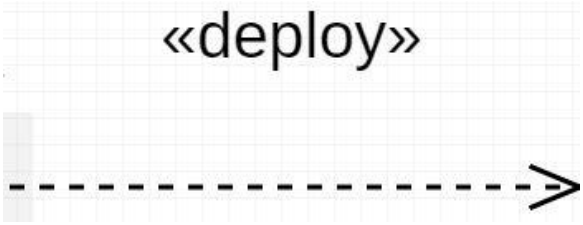

Ilustración 56: Elemento Desplegar. Fuente: Propia.

Comunicador: Por lo general se usa cuando se explica de **una manera explícita** como **usa un nodo a otro** y con qué **cantidad**.

Ilustración 57: Elemento Comunicador. Fuente: Propia.

Interfase: hace parte del **módulo** con la que un componente se **comunica** con otros iguales y se pone dentro de un nodo.

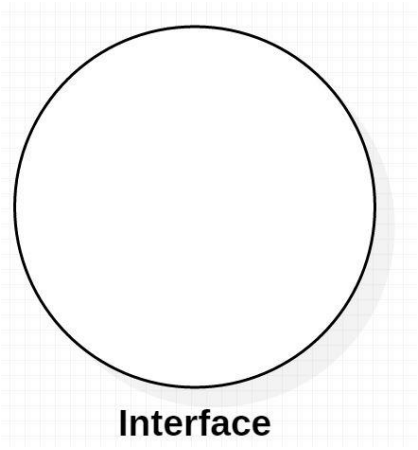

Ilustración 58: Elemento Interfase. Fuente: Propia.

Artefacto: es la representación de **ficheros** cono los que se trabaja, estos deben tener una connotación un tanto externa, como una factura o un archivo de texto cuales se coloca dentro del nodo.

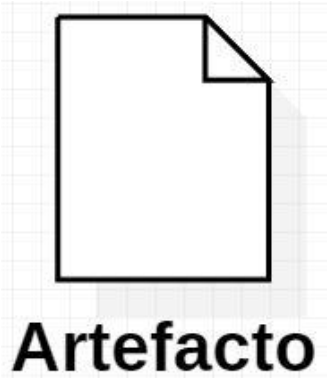

Ilustración 59: Elemento Artefacto. Fuente: Propia.

Componente: la mayoría de las veces es representado como una especie **de clase o sub clase** la cual **demuestra** por lo general un **conjunto de objetos** dentro, se coloca dentro del nodo.

Ilustración 60: Elemento Componente. Fuente: Propia.

Dependencia: demuestra que un componente tiene una **existencia dependiente** del componente al que señala, nunca debe de salir del nodo.

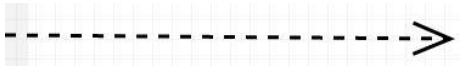

Ilustración 61: Elemento Dependencia. Fuente: Propia.

Realización: demuestra que una interface, componente o artefacto **realiza acciones no especificas** con otro.

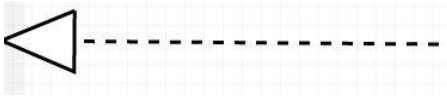

Ilustración 62: Elemento Realización. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de componentes en el cual se da el desarrollo de un prototipo para un videojuego, para eso se tienen que desarrollar los prerrequisitos donde se sacó el despliegue que conforman al sistema.

El prototipo se ejecutará en una computadora la cual contara con RAM, CPU, tarjeta gráfica (GPU), Mouse y Teclado además de todos los buses de datos que comunican los componentes entre ellos, como se muestra en la ilustración sesenta y seis (63) se ejecutara en el disco duro, dentro del disco se ejecuta el sistema operativo dentro de este el motor gráfico y adentro el prototipo.

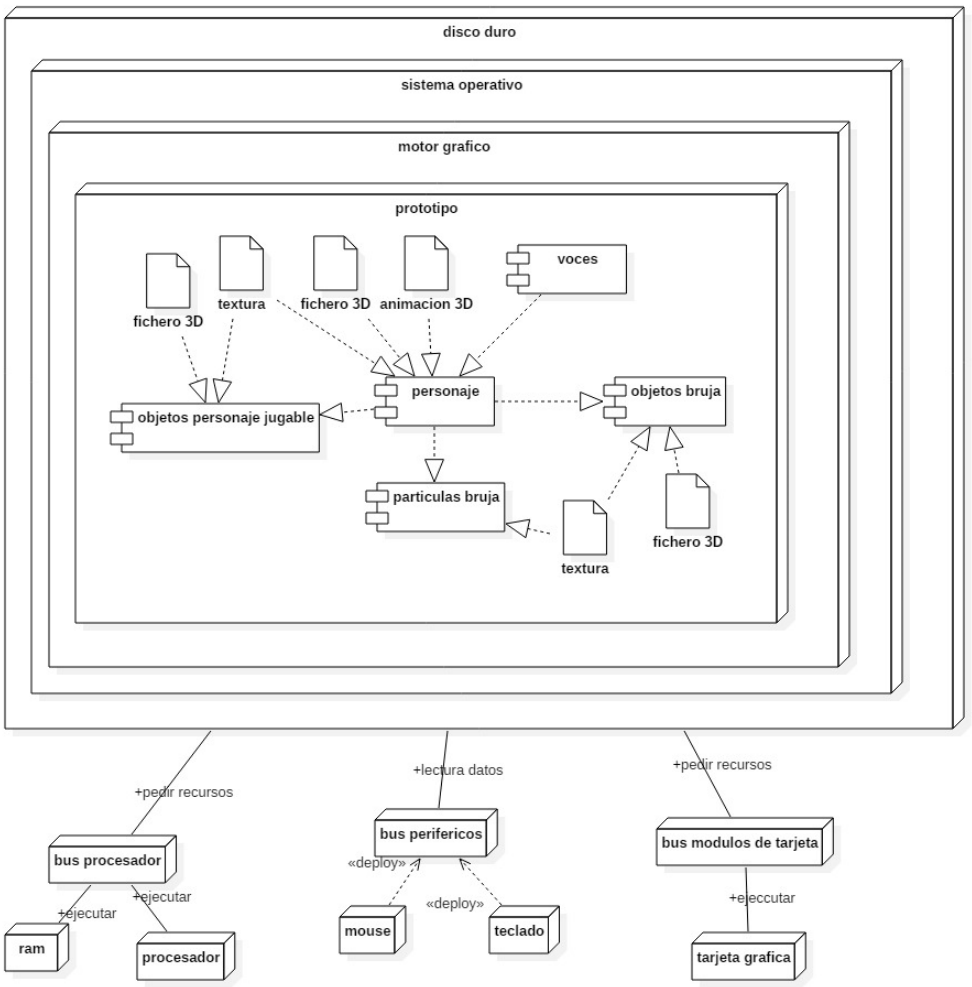

Ilustración 63: Ejemplo Diagrama de despliegue. Fuente: Propia.

Pues así el prototipo por medio del motor grafico cual usa al sistema operativo que está en el disco duro pide recursos a la memoria de acceso aleatorio, procesador y gráfica, estos sin depender de nada ejecutaran los recursos para el videojuego, mientras que al mismo tiempo se hará lectura de datos para el mouse y teclado.

El diagrama de componentes entro del prototipo expresa lo mismo que el grafico de componentes sin entrar en escala por tanto es una versión más simplificada del mismo.

Diagramas de estructura compuesta

Funcionamiento

En el diagrama de estructura compuesta está compuesto por partes y conectores de un clasificador estructurado o una colaboración, los cuales son muy parecidos a los de colaboración además de los de clase.

Funcionamiento

En una ontología para la representación de conceptos de diseño de software se señala que: “Un diagrama de Estructura Compuesta refleja la colaboración interna de clases, interfaces o componentes para describir una funcionalidad. Los diagramas de estructura compuesta son similares a los diagramas de clase, a excepción de que estos modelan un uso especifico de la estructura. Los diagramas de clase modelan una vista estática de las estructuras de clase, incluyendo Sus atributos y comportamientos. [Un diagrama de Estructura Compuesta se usa para expresar arquitecturas en tiempo de ejecución, patrones de uso, y las relaciones de los elementos participantes, los que pueden no estar reflejados por diagramas estáticos” (Gloria, Acevedo, & Moreno, 2011).

Relata con gran detalle que variables, así como operaciones tiene un sistema en todas sus estancias para así saber cómo deberían ser aplicadas las funciones y variables dentro de este.

Elementos

López y Ruiz lo mencionan de esta forma: “Muestran la estructura interna (incluyendo partes y conectores) de un clasificador estructurado o una colaboración. -Muy parecidos a los diagramas de componentes.” (Riuz & López, 2020).

En El Lenguaje Unificado De Modelado Manual De Referencia UML 2.0 se da la siguiente connotación: “Una vez que comienza el proceso de diseño, las clases se deben descomponer en colecciones de partes conectadas que, posteriormente, se deben descomponer por turnos. Un clasificador estructurado modela las partes de una clase y sus conectores contextuales. Una clase estructurada puede ser encapsulada forzando a que las comunicaciones desde el exterior pasen a través de los puertos cumpliendo con las interfaces declaradas.” (Rumbaugh, Jacobson, & Booch, El Lenguaje Unificado de Modelado Manual de Referencia 2.0, 2007).

Clase: Dentro del grafico demuestra una **clase** que debería de **tener el sistema**.

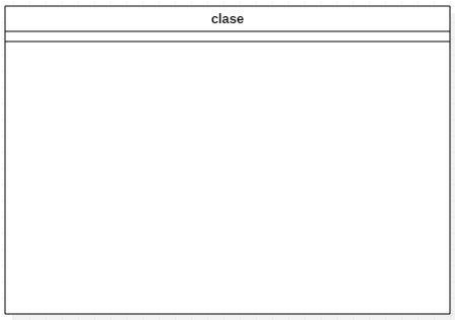

Ilustración 64: Elemento Clase. Fuente: Propia.

Función: Aparecen dentro y representan una **función real de la clase.**

Ilustración 65: Elemento Función. Fuente: Propia.

Atributo: Las funciones podrán tener **objetos o variables** con las cuales se desarrollen.

Ilustración 66: Elemento Atributo. Fuente: Propia.

Operación: Las funciones podrán tener **actividades** con las cuales se **demuestra la ejecución.**

Ilustración 67: Elemento Operación. Fuente: Propia.

Puerto: Muestra la **salida o entrada lógica de las clases** que deben llevar a una función en otra clase.

Ilustración 68: Elemneto puerto. Fuente: Propia.

Realización: Demuestra la comunicación de una o varias funciones con otras clases o con otras funciones de igual modo con los puertos.

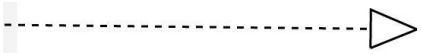

Ilustración 69: Elemento Realización. Fuente: Propia.

Conector: Demuestra la **conexión entre dos puertos** de diferentes clases sin aclarar de qué forma se transmiten los datos.

Ilustración 70: Elemento Conector. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de clases en el cual se da el desarrollo de un prototipo para un videojuego, para eso se tienen que desarrollar los prerrequisitos donde se sacó la estructura compuesta que conforman al sistema.

En el diagrama mostrado en la ilustración setenta y uno (71) se nos muestran: colisión de personajes, personaje, personajefuncion, particuladefensa, partícula, particuladaño, arma, personajejugable, jefes, brujas y ost cada uno tendrá varias funciones en el cual estarán los valores y operaciones con las que se ejecutara el programa.

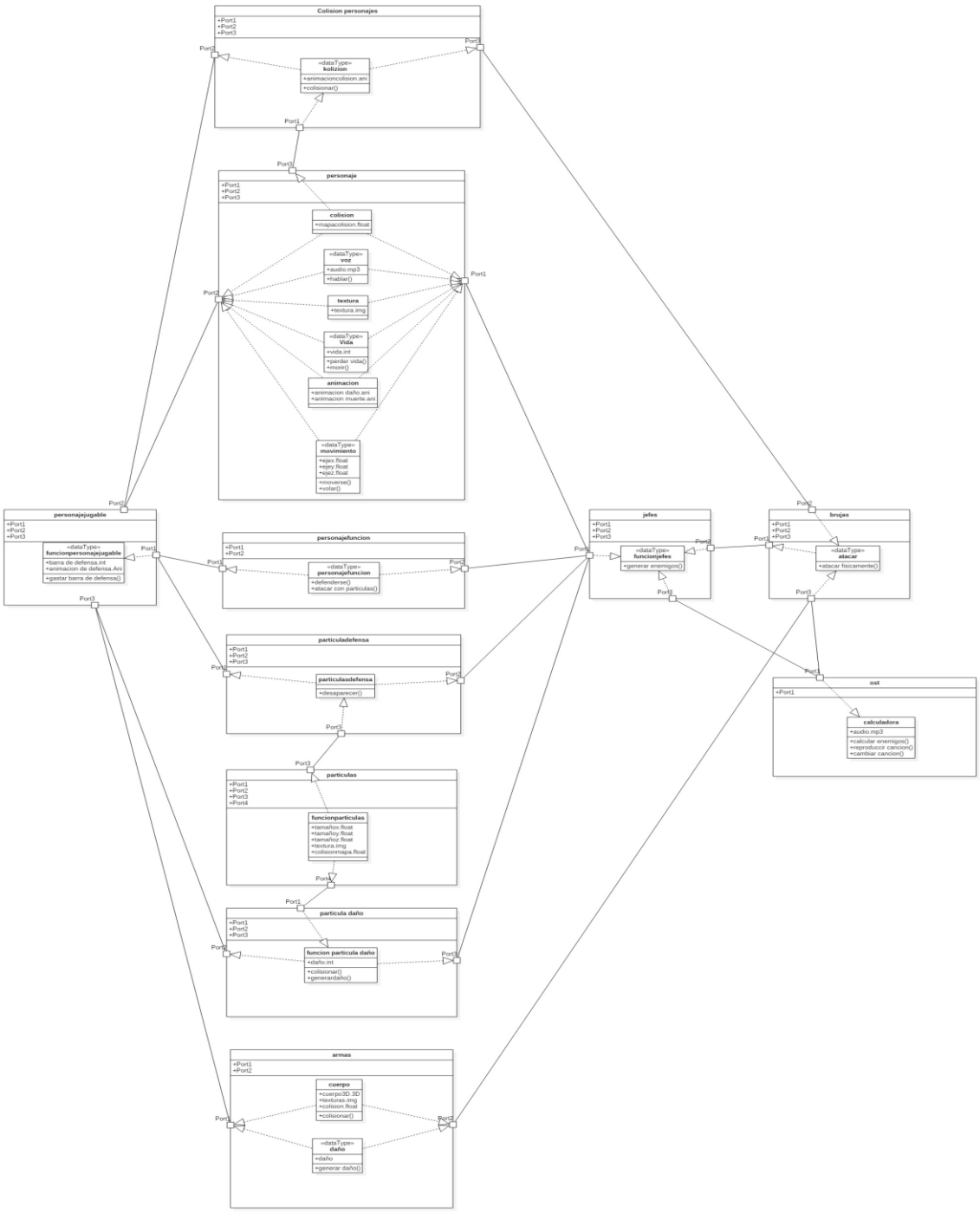

Ilustración 71: Ejemplo Diagrama Estructura Compuesta. Fuente: Propia.

Algunas de las variables que tiene el diagrama no son explicitas como .ani, .3D y .img ya que no se conocen cuáles son los requisitos de los formatos de imagen, modelo 3D y animación. La mayoría mantienen una sola función en la cual se especifica lo

mismo que el diagrama de clases, siendo las únicas que tiene más de una función personaje y arma.

Como se puede apreciar por cómo se dirigen las flechas todo está conectado al personaje principal y jefe estando en menor medida brujas.

Diagramas de secuencia

definición

Es la demostración de la interacción secuencial entre los objetos del sistema con otros o componentes de otros indoles como las bases de datos.

En El Lenguaje Unificado De Modelado Manual De Referencia UML 2.0 se da la siguiente connotación: “Un diagrama de secuencia muestra un conjunto de mensajes ordenados en una secuencia temporal. Cada rol se muestra como una línea de vida —es decir, una línea vertical que representa al rol a lo largo del tiempo a través de la interacción completa. Los mensajes se muestran con flechas entre líneas de vida. Un diagrama de secuencia puede mostrar un escenario —una historia individual de una transacción. Las construcciones de control estructurado, como los bucles, las condiciones y las ejecuciones en paralelo, se muestran como rectángulos anidados con palabras clave y una o más regiones.” (Rumbaugh, Jacobson, & Booch, El Lenguaje Unificado de Modelado Manual de Referencia 2.0, 2007).

En una ontología para la representación de conceptos de diseño de software se señala que: “Un tipo de diagrama usado para modelar interacción entre objetos en un sistema según UML. Muestra la interacción de un conjunto de Objetos en una aplicación a través del tiempo y se modela para cada Caso de uso.” (Gloria, Acevedo, & Moreno, 2011).

Funcionamiento

Representa una interacción de dos o más objetos en un caso determinado de un sistema.

En Ingeniería de software I se muestra como características: “Es un diagrama de interacción que resalta la ordenación temporal de los mensajes intercambiados durante la interacción.

-Presentan un conjunto de roles y los mensajes enviados y recibidos por las instancias que interpretan dichos roles.

-Habitualmente, sirven para mostrar como interaccionan unos objetos con otros en un caso de uso o un escenario de ejecución.” (García, Moreno, & García, 2018).

Elementos

Objeto: representa el **objeto de un sistema** el cual tendrá interacción con otros casos.

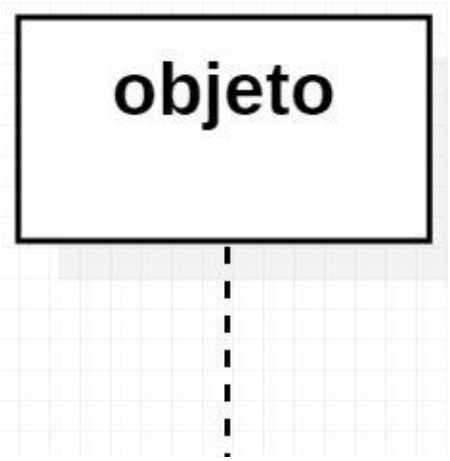

Ilustración 72: Elemento Objeto. Fuente: Propia.

Mensaje: Es la **interacción, petición o acción** que tendrá un objeto con otro.

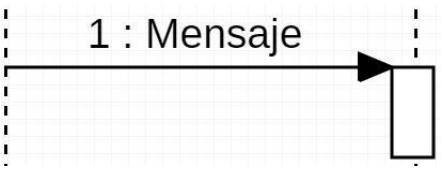

Ilustración 73: Elemento Mensaje. Fuente: Propia.

Replica: Se da cuando se espera que se muestre una **interacción en respuesta con la acción que se le ha dado**, tal se da con el mismo que manda el mensaje.

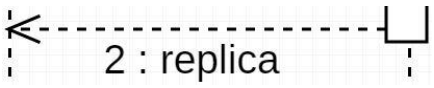

Ilustración 74: Elemento Replica. Fuente: Propia.

Auto mensaje: Es la **acción que se da del objeto consigo mismo**, esta tiende a estar ligada a otra que la hace ser.

Ilustración 75: Elemento auto mensaje. Fuente: Propia.

Borrar: Se da cuando **termina su vida en el sistema** y se requiere que sea **borrada por el sistema u objeto** dependiendo de la acción que se tome.

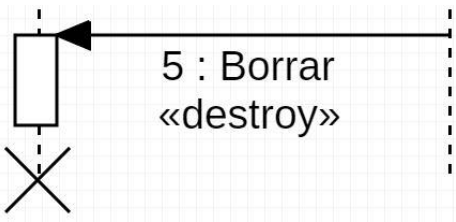

Ilustración 76: Elemento Borrar. Fuente: Propia.

Marco de acción: Dentro de este se darán **todas las interacciones y o desapariciones**.

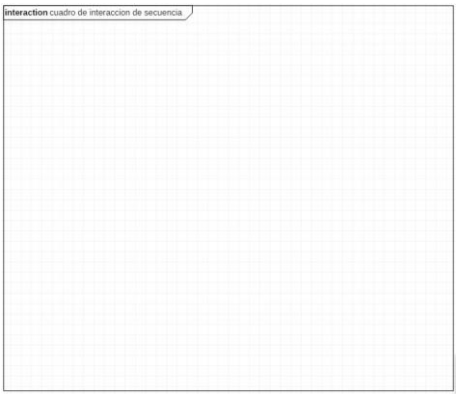

Ilustración 77: Elemento Marco. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de objetos y actividades en el cual se da el desarrollo de un prototipo para un videojuego, para eso se tienen que desarrollar los prerrequisitos donde se sacaron las secuencias que conforman al sistema.

Como se puede evidenciar en el grafico mostrado en la ilustración Setenta y ocho (78) hay un total de diez (10) objetos los cuales tienen un total de treinta y cinco (35), estos al llegar un punto se eliminan exceptuando por la canción la cual seguirá existiendo, el personaje jugable utiliza de cuatro elementos y el jefe de tres, los enemigos generados usará una lanza y canción calculará al jefe así como enemigos generados.

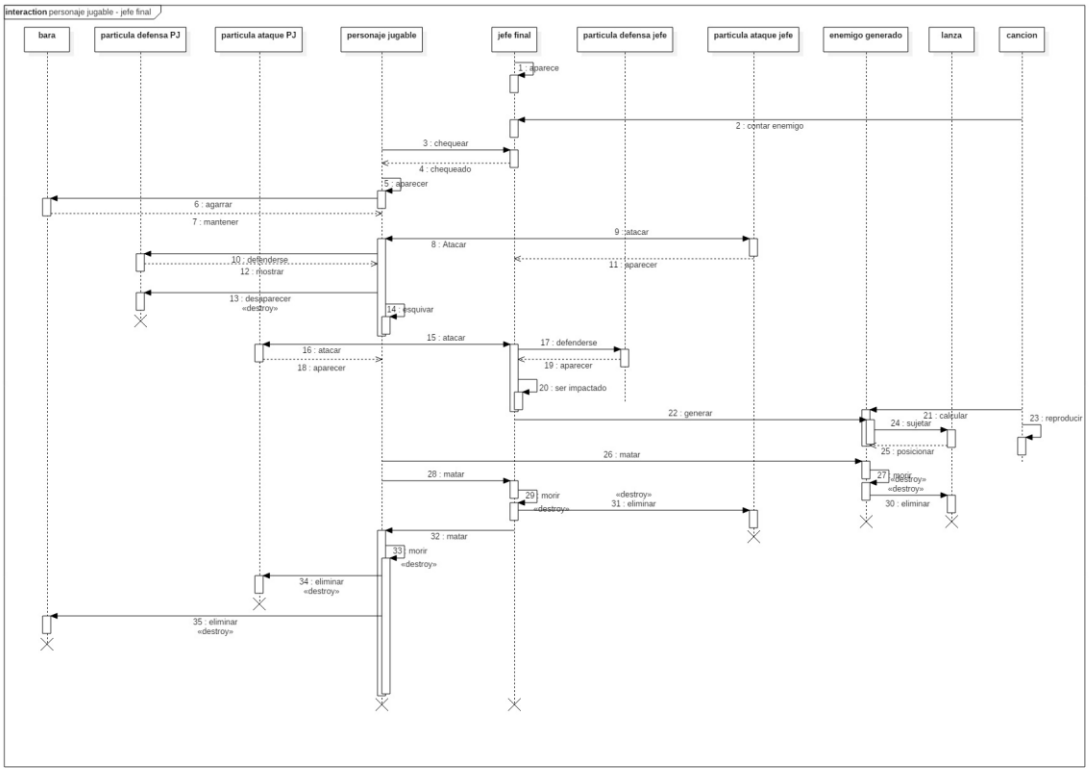

Ilustración 78: Ejemplo Diagrama de Secuencia. Fuente: Propia.

El primero en aparecer será el jefe seguido por el personaje jugable mismo que pausara el sistema de darse las condiciones mostradas, sin embargo, el jefe final también puede usar este recurso tanto el PJ como el enemigo generado agarraran sus armas apenas aparezcan y esta arma desaparecerá con ellos.

Diagramas de paquetes

Definición

En El Lenguaje Unificado De Modelado Manual De Referencia UML 2.0 se escribe en el primer párrafo de este apartado: “UML está definido utilizando un metamodelo, es decir, un modelo del propio lenguaje de modelado. La modificación del metamodelo es complicada y peligrosa. Además, muchas herramientas son construidas a partir del metamodelo estándar, y no funcionarían correctamente con un metamodelo diferente. El mecanismo de los perfiles permite cambios limitados sobre UML sin modificar el metamodelo subyacente. Los perfiles y las restricciones permiten que UML sea adaptado a dominios o plataformas específicas mientras mantiene la interoperabilidad entre herramientas.” (Rumbaugh, Jacobson, & Booch, El Lenguaje Unificado de Modelado Manual de Referencia 2.0, 2007).

En las diapositivas UML diagramas de paquetes por Demian Gutiérrez (2009), se escribe que es un elemento para agrupar elementos en un UML.

Funcionamiento

En una ontología para la representación de conceptos de diseño de software se señala que: “Los Diagramas de Paquetes se usan para reflejar la organización de los paquetes y sus elementos, y para proveer una visualización de sus correspondientes nombres de espacio.” (Gloria, Acevedo, & Moreno, 2011).

Permiten organizar dichos elementos para facilitar el desarrollo de diagramas en sistemas complejos, este define un espacio de nombre el cual puede ser compartido mientras estén en paquetes distintos.

En Ingeniería de software I se muestra como características: "• Los paquetes constituyen un mecanismo de agrupación para organizar elementos UML

• Proporcionan un espacio de nombres a los elementos agrupados

• Los paquetes se organizan jerárquicamente, siendo el paquete raíz el que contiene todo el sistema.

• Un paquete se representa como un rectángulo grande con un rectángulo pequeño sobre su esquina superior izquierda. El nombre puede aparecer en cualquiera de los dos rectángulos. Si se muestra el contenido del paquete, entonces el nombre aparece en el rectángulo pequeño." (García, Moreno, & García, 2018).

Elementos

Carpeta: Este elemento demuestra una carpeta en la cual **se albergarán objetos y clases.**

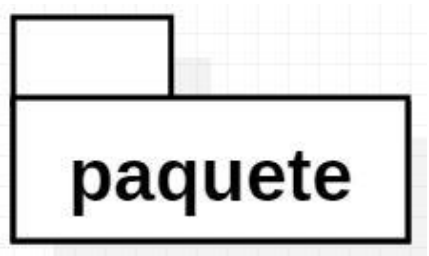

Ilustración 79: Elemento Paquete. Fuente: Propia.

Contiene: Esta relata que la **carpeta contiene dentro** de ella a aquella que se encuentra del lado abierto de la línea.

Ilustración 80: Elemento Contiene. Fuente: Propia.

Dependencia: Esto muestra que una **carpeta depende de los objetos o clases** a la quien se le señala.

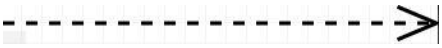

Ilustración 81: Elemento Dependencia. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de objetos en el cual se da el desarrollo de un prototipo para un videojuego, para eso se tienen que desarrollar los prerrequisitos donde se sacaron los paquetes que conforman al sistema.

En la ilustración ochenta y dos (82) se puede apreciar que en prototipo es donde se desarrolla el sistema el cual consta de seis carpetas cuales contienen los objetos y clases elocuentes al nombre en cuestión. Apreciando así que el jefe tal como personaje jugable tienen una dependencia por las partículas, el ultimo mencionado además de brujas tienen dependencia por las armas, terminando por OST cual depende de los enemigos.

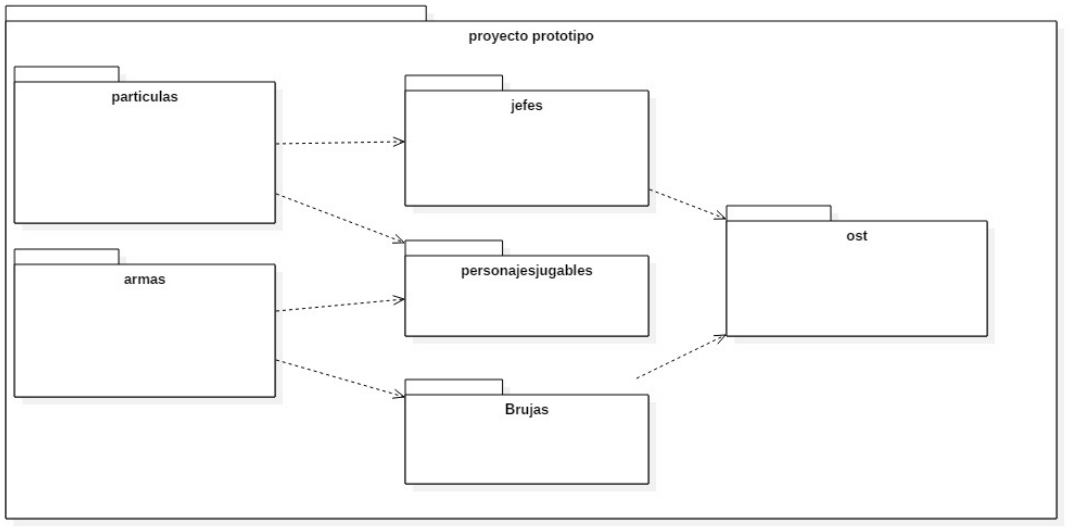

Ilustración 82: Ejemplo Diagrama de Paquetes. Fuente: Propia.

Diagramas de colaboración

Definición

Un diagrama de colaboración es un diagrama que contiene calificadores y roles de asociación en vez de solo una de estas características.

En El Lenguaje Unificado De Modelado Manual De Referencia UML 2.0 se escribe en el primer párrafo de este apartado: “Un diagrama de comunicación muestra roles en una interacción con una disposición geométrica (Figura 3.10). Cada rectángulo muestra un rol —una línea de vida que representa la vida de un objeto a lo largo del tiempo. Los mensajes entre los objetos que desempeñan los roles se muestran como flechas vinculadas a los conectores. La secuencia de mensajes se indica mediante números de secuencia que preceden a la descripción de los mensajes.” (Rumbaugh, Jacobson, & Booch, El Lenguaje Unificado de Modelado Manual de Referencia 2.0, 2007).

Funcionamiento

Estos definen la configuración del objeto y los enlaces que pueden ocurrir cuando se da la ejecución de una de las instancias.

En una ontología para la representación de conceptos de diseño de software se señala que: “Muestra interacciones organizadas alrededor de los roles. A diferencia de los diagramas de secuencia, los diagramas de comunicación muestran explícitamente las relaciones de los roles.” (Gloria, Acevedo, & Moreno, 2011).

En Ingeniería de software I se muestra como características: "• Los diagramas de comunicación se centran en las interacciones y en los enlaces entre los objetos que colaboran, siendo secundario el orden de envío y recepción de mensajes

• Se representan dentro de un marco con el nombre del diagrama precedido del prefijo sd dentro del símbolo que aparece en la esquina superior izquierda del marco

• Sólo se representa el rectángulo de la línea de vida

• Los mensajes se colocan cerca de los enlaces. Se representan con una flecha y una etiqueta que contiene el nombre del mensaje y otra información adicional" (García, Moreno, & García, 2018).

Elementos

Cuadro de interacción: Se pondrán **todas las interacciones** de los objetos.

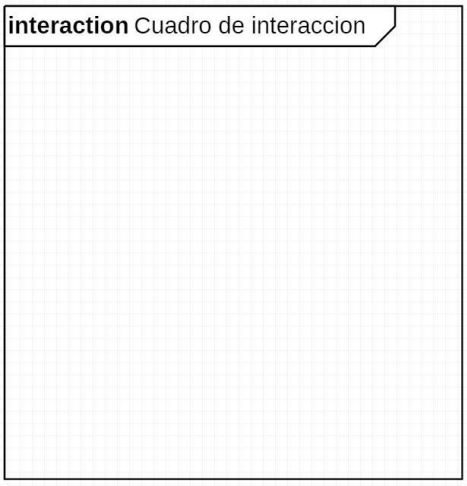

Ilustración 83: Elemento Cuadro de Interaccion. Fuente: Propia.

Objetos: Los objetos son **objetos del sistema o actores** que interactúan.

Ilustración 84: Elemento Objeto. Fuente: Propia.

Conexión: Se muestran la **conexión entre dos objetos** y que tipo de conexión poseen.

Ilustración 85: Elemento Conexión. Fuente: Propia.

Mensaje: Es la **acción de un objeto con otro** el cual muestra una acción del caso de uso, actividad o secuencia.

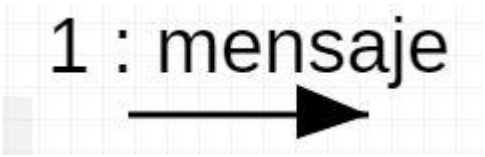

Ilustración 86: Elemento Mensaje. Fuente: Propia.

Respuesta: Se da como **respuesta al mensaje** y se usa cuando se espera que el mensaje tenga una respuesta del suso dicho.

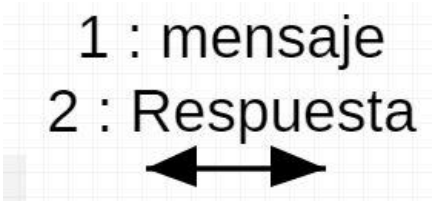

Ilustración 87: Elemento Respuesta. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de secuencia en el cual se da el desarrollo de un prototipo para un videojuego, para eso se tienen que desarrollar los prerrequisitos donde se sacaron las colaboraciones que conforman al sistema.

En la ilustración ochenta y ocho (88) se muestra como el grafico consta de cinco objetos de los cuales uno es usuario, se muestran catorce (14) interacciones el cual muestra que el objeto música calcula si hay un jefe final cual ataca al personaje jugables, el jugador esquivo ya que siempre maneja al personaje jugable, posteriormente ataca si es impactado el jefe final pierde vida, podrá genera enemigos cuales son calculados por la música.

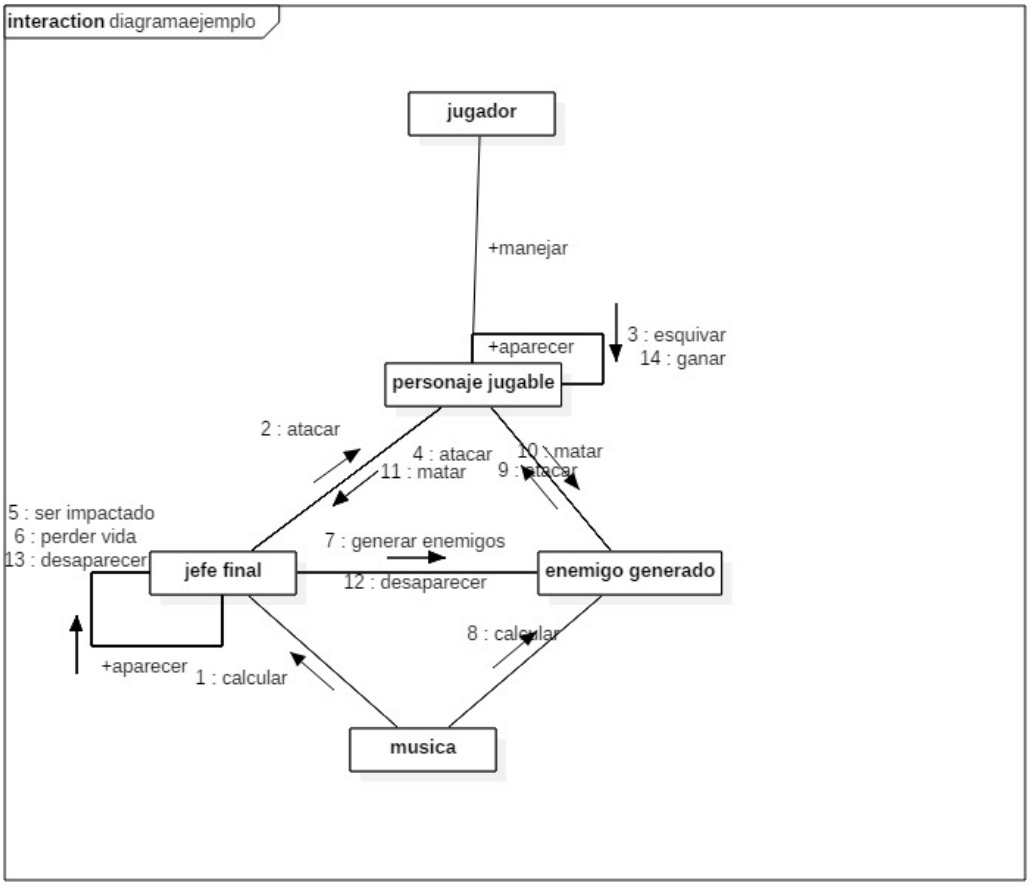

Ilustración 88: Elemento Diagrama de Colaboración. Fuente: Propia.

Los enemigos generados atacan al personaje jugable así que los mata, en caso de que mate al jefe final los enemigos generados y el mismo desaparecerán, por tanto, el personaje jugable habrá ganado la partida.

Diagramas de tiempo

Definición

Es la descripción del estado de un componente u objeto a lo largo de la vida en interacciones específicas.

Funcionamiento

En una ontología para la representación de conceptos de diseño de software se señala que: “El diagrama de tiempo define el comportamiento de los diferentes objetos con una escala de tiempo. Provee una representación visual de los objetos cambiando de estado e interactuando a 10 largo del tiempo. Puede usar diagramas de tiempos para definir componentes de software dirigidos por hardware o embebidos; por ejemplo, aquellos usados en un sistema de inyección de combustible, un controlador de microondas. También puede usar diagramas de tiempo para especificar procesos de negocio dirigidos por tiempo” (Gloria, Acevedo, & Moreno, 2011).

En Ingeniería de software I se muestra como características: “• Los diagramas de tiempo (timing diagrams) proporcionan una forma de mostrar los objetos activos y sus cambios de estado durante sus interacciones con otros objetos activos y con otros recursos del sistema

• Para representarlos se utiliza el marco de los diagramas de interacción.

• El eje X muestra las unidades de tiempo y el eje Y muestra los objetos y sus estados

• Se puede representar el cambio en el estado de un objeto a lo largo del tiempo en respuesta a eventos o estímulos.

• Permite la representación de diferentes tipos de mensajes. Los mensajes se pueden dividir mediante etiquetas para mejorar la legibilidad de los diagramas.” (García, Moreno, & García, 2018).

“Los diagramas de tiempo forman herramientas que sirven para plantear y resolver problemas financieros, pues son ütiles para identificar los desplazamientos simbólicos del capital en el tiempo. Por medio de los desplazamientos se pueden llevar las cantidades de dinero que forman parte de un problema, hasta una fecha en común, la cual se conoce como Fecha focal o Fecha de referencia.” (Cortez, 2012)

Son usados para marcar el estado de un componente u objeto en un sistema tanto de hardware como software en el cual se describe cuál es ese a lo largo de un ciclo o vida de tiempo.

Elementos

Componente: Es aquel **objeto en el sistema** del cual se espera un comportamiento a lo largo de su vida útil.

Ilustración 89: Elemento componente. Fuente: Propia.

Estado: Un objeto a lo largo de su línea de vida este hecho por **tres estados, apagado, en espera o activo.**

Ilustración 90: Elemento Estado. Fuente: Propia.

Línea de tiempo: Se muestra de manera horizontal y demuestra en una **cantidad de tiempo pre establecida** lo que pasara en él con el componente.

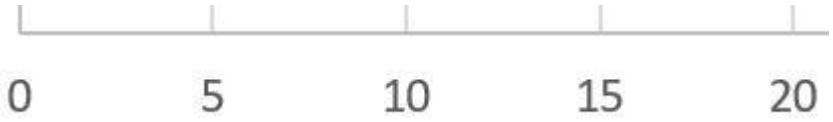

Ilustración 91:Elemento Línea de tiempo. Fuente: Propia.

Línea de vida: Mostrará el **estado de un objeto** o componente a lo **largo de su vida** diagramada.

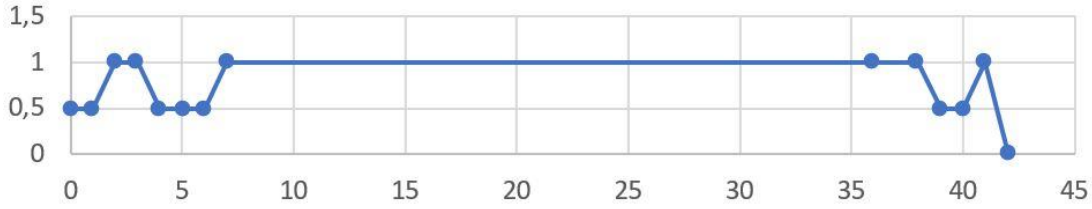

Ilustración 92: Elemento Línea de Vida. Fuente: Propia.

Ejemplo

Para este ejemplo se tomará el que se evidencio en el diagrama de colaboración en el cual se da el desarrollo de un prototipo para un videojuego, para eso se tienen que desarrollar los prerrequisitos donde se sacaron los tiempos que conforman al sistema.

En la Tabla uno (1) se puede apreciar que el numero cero demuestra que el componente esta apago o fue eliminado, cero punto cinco es para un objeto en espera de ser activado que ya está activo y uno es para un objeto que está activo, se reconocen catorce actividades que cambiaran de estado los objetos personaje jugable, jefe final, enemigo y música el cual se ejecutara en un lapso aproximado de cuarenta y dos (42) segundos o menos de un minuto.

Tabla 1: Estados Componentes Diagrama de Tiempo. Fuente: Propia.

tiempo segundos	personaje	jefe final	enemigo	musica	accion
0	0,5	0,5	0,5	0,5	carga
1	0,5	1	0,5	1	ataque jefe
2	1	0,5	0,5	1	esquive
3	1	0,5	0,5	1	ataque personaje
4	0,5	1	0,5	1	golpe jefe
5	0,5	1	1	1	generacion enemigos
6	0,5	0,5	1	1	ataque a personaje jugable
7	1	0,5	1	1	matar enemigos
36	1	0,5	1	1	matar enemigos tiempo final
38	1	1	0,5	1	matar jefe
39	0,5	0,5	0	1	eliminar enemigos
40	0,5	0	0	1	eliminar jefe
41	1	0	0	1	ganar jugador
42	0	0	0	0	terminar

En la ilustración noventa y tres (93) se puede apreciar que todos los objetos del sistema están esperado la entrada de una señal al cargar el sistemas, cuando este termina de cargar se llevan a cabo las actividades de los objetos, el personaje jugable es el segundo más activo seguido por los enemigos generados y acabando por el jefe final quien la mayor parte del tiempo tiene un estado pasivo, la música es la que tiene más actividad ya que siempre se está ejecutando sin importar los demás objetos escenificados.

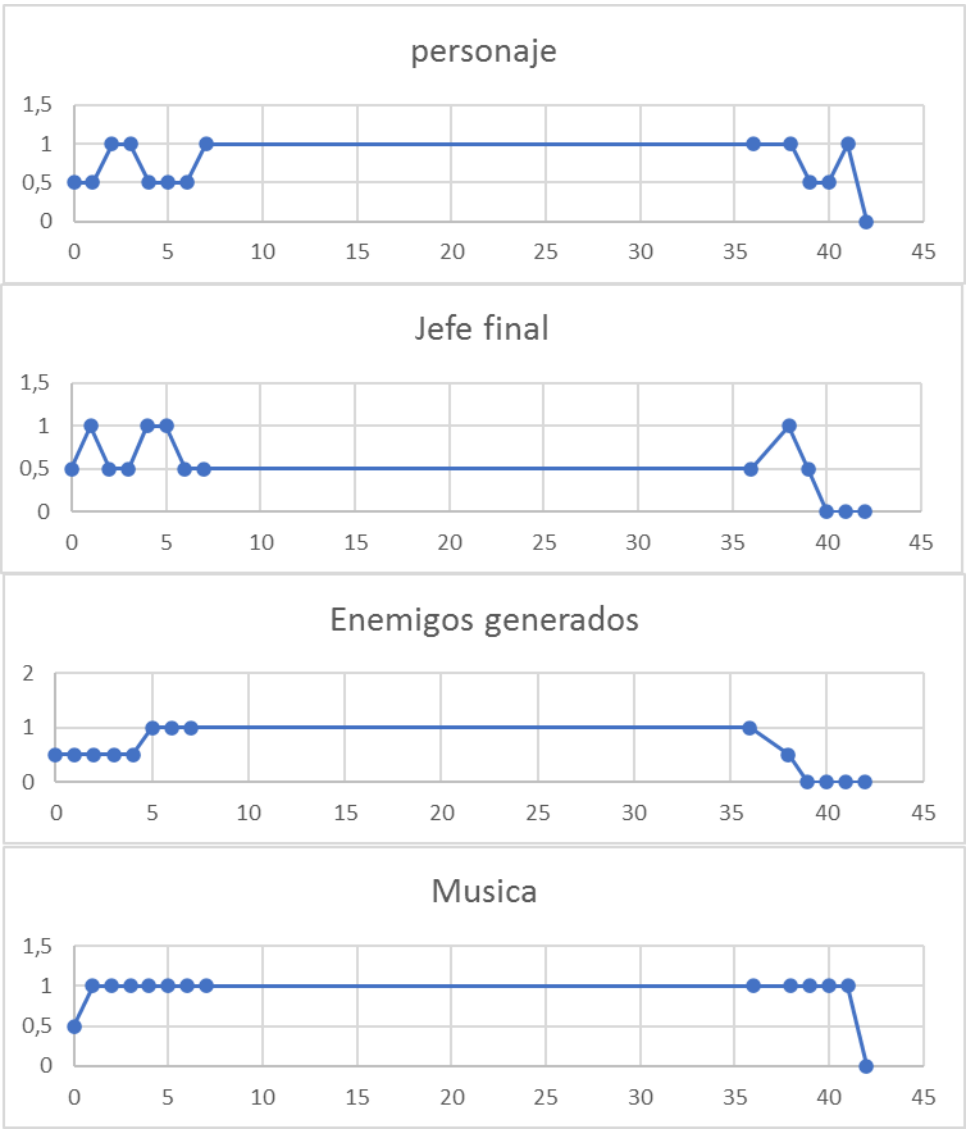

Ilustración 93:Ejemplo Diagrama de Tiempo. Fuente: Propia.

En el diagrama que se muestra que el primero en actuar es el jefe final sin tener en cuenta la música, el personaje responde esta señal y a continuación vuelve a responder el jefe final mandando enemigos generados quienes combaten contra el personaje jugable hasta que este los elimina a todos en el segundo 36, entonces los enemigos generados pasan a una posición pasiva donde el personaje jugable remata al jefe, haciendo que se eliminen los enemigos generados para posteriormente desaparecer el jefe final.

De allí la música se mantiene para dar la señal de victoria y se activaran las mecánicas de victoria del personaje donde acaba el sistema.

Casos de éxito

Para los casos de éxito se tratará de forma más o menos explícita los implementos de tecnología que se han desarrollado con UML, cuales gráficos han sido implementado y para que función han servido.

En el documento Generación automática de código a partir de diagramas de gráficos de estado escriben Sunitha E. V. y Philip Samuel que la idea detrás de la implementación del sistema se pude modelar usando notaciones como las dadas en LMU, cual luego se compilo y ejecuto para probar el funcionamiento incluso antes de su escritura completa; estos datos les resultaron interesantes en la fase de codificación y pruebas de desarrollo los cuales son muy costosos.

El lenguaje de modelado unificado ayuda a compilar el modelo lo cual reduce el esfuerzo de codificación y pruebas, mientras al mismo tiempo se mejora la calidad del software; el UML también pudo reducir los errores de los productos que se necesitaron desarrollar, les ayudaron a definir la especificación de requisitos, se mantiene la correlación entre el diseño y la ejecución. En el mantenimiento se cambia el lenguaje fuente, pero no la arquitectura.

Por tanto, cada mantenimiento realizado reduce la correlación con la estructura y el mismo código fuente, haciendo que a lo largo de su vida útil la escritura no tenga ninguna relación con el proyecto real; por tanto, el lenguaje estructural da una solución a este problema: durante el mantenimiento se pueden cambiar los modelos y no la programación real.

El código fuera del modelo del sistema, esta idea proviene del UML ejecutable, el cual dice que puede ser modelado entre la comunicación de objetos, donde cada objeto tiene su propia de transición de estado, lo cuales sirven para entrar en eventos y actuar en consecuencia.

En este mismo documento de desarrollo se describe que: “Una máquina de estados se puede definir como un gráfico de estados y transiciones (…). El diagrama del diagrama de estado se puede adjuntar a clases casos de uso y colaboraciones para describir la dinámica de un objeto individual. Modela todas las historias de vida posibles de un objeto de una clase. Cualquier influencia externa al objeto es llamada como evento. La respuesta al evento puede incluir la ejecución de una acción y transición a un nuevo estado. Eventos puede tener parámetros que caracterizan cada evento individual ejemplo. La herencia y la simultaneidad se pueden modelar en estado máquinas.” (Sunitha & Samuel, 2019).

Este se puede modelar usando diagramas de clase, estado y actividad; estos gráficos pueden ayudar mucho a la generación automática de escritura ya que con estos se puede dar programación de implementación usando herramientas de generación de código: los de clase ayudan con la generación de escrituras estructurales, muestras que los otros dos ayudan a la generación de programación de comportamiento.

El de estado es el más popular para el diseño incrustado y el modelado impulsados por eventos, la concentración del esquema puede generar un programa de 100%. Los de LMU también pueden representar de una manera muy acertada el sistema de los métodos controlados por eventos, así como los reactivos. El sistema de los métodos controlados por eventos cambia con las interacciones según el entorno.

Los eventos representados por el grafico anteriormente mencionado pueden representar el ciclo de vida de un objeto. Luego el desafío de la metodología consiste en elaborar un método eficiente para convertir estos gráficos en un programa debido a la inexistencia de programación programada para mostrar los elementos de manera directa en los esquemas. En el documento se presenta un método para convertir los estados jerárquicos, concurrentes e históricos en el lenguaje de programación java.

Mostrándose lo anteriormente mencionado en este párrafo: “Aunque es un método simple de implementación de gráficos de estado, no puede admitir estados concurrentes en un gráfico de diagrama de estados. Además de eso, los estados compuestos no pueden ser implementado usando este método, ya que las jerarquías estatales no se pueden representar en sentencias de cambio de caso.” (Sunitha & Samuel, 2019).

En ese método se sigue un patrón de arquitectura de referencia, estos últimos se adhieren a un diagrama de clases para generar una implementación en línea. Las principales contribuciones del diseño son las siguientes: presentar una estructura de planeamiento muy comprensible.

Que este sea reutilizable para la escritura de la máquina de estado, el patrón del plan es expansible y puede manejarlos se forma jerárquica, proporcionar un método eficaz para implementar tales de compo-sitio incluyendo regiones paralelas orientada a objetos, método simple para mantener el historial superficial y profundo en una máquina de etapas.

El patrón de arquitectura propuesto también proporciona modularidad e incomprensión; se mantiene la semántica jerárquica estos, la actualidad y fases de historia; debido a modularidad el patrón es en eficiencia fácilmente expandible. Se hace posible que haya una pauta estructural y señal de tormenta, el método cumple a cabalidad con lo planteado para este.

En el estudio de caso de éxito examina: “Pais y Oliveira proponen algunos tipos de estéreo para UML para especializar los elementos de frontera, control y entidad o clases. El diagrama de robustez representado usando estos los tipos estéreo se pueden convertir a diagramas de gráficos de estado UML, diagrama de secuencia y diagrama de clases. Kundu y col propone un método de generación de código a partir de diagramas de secuencia UML. Los diagramas de secuencia primero se convirtieron en secuencia gráficos de interacciones. Estos gráficos contienen información como mensajes, flujo de control y alcance del método de interacciones. Estos gráficos luego se transformarán en código. Este método es más adecuado para las clases de controlador y no para límites y clases de entidad. Sunitha y Samuel presentan la formal asociación entre diagrama de actividad y diagrama de secuencia. Basado en esta definición formal, el artículo propone un método para generar código a partir del diagrama y la secuencia de actividades UML diagramas. Este método es el más adecuado para límites y control clases.” (Sunitha & Samuel, 2019).

El proyecto admite la concurrencia y la historia de La etapa sin comprometer la capacidad de expansión, reutilización y comprensión, es más eficiente en términos de tiempo empleado para el procesamiento de eventos; proporciona un resultado menos complejo y resultados comprometedores.

Dados todos los resultados del documento se da como un hecho que el proyecto desde su planificación hasta implementación se considera bueno usando en el desarrollo UML, por tanto, ha de considerarse un caso de éxito en la ejecución del lenguaje en el proceso.

Yilong Yang, Wei Ke, King Yang y Xiaoshan Li escriben es su artículo Integración de UML con Refinamiento de Servicio para Modelado y Análisis de Requerimientos es descubrir la correspondencia entre UML y el refinamiento de servicios, con el fin de aprovechar las ventajas que da el Modelo de lenguaje unificado y los métodos formales en modelado de requisitos, mientras se pueda realizar una verificación de seguridad formal y refinamiento mediante una herramienta popular de modelado de requisitos.

Comenzando el estudio con este párrafo: “La teoría del refinamiento del servicio describe cómo especificar los servicios de un sistema y las relaciones de refinamiento entre varios conceptos en las especificaciones. Los conceptos incluyen interfaces para describir las firmas de servicios y las variables, especificaciones para describir la funcionalidad y comportamiento, y contratos para describir los protocolos de interacción, en particular, las secuencias de invocación previstas de los servicios en un sistema. Con las especificaciones de comportamientos y los protocolos de interacciones, la consistencia de un sistema se puede definir y razonar, donde libre de interbloqueo y livelock-free son las propiedades con las que lidiar (…).” (Yioling, Wei, King, & Xiaoshan, 2019).

El refinamiento del servicio es un método formal bien diseñado en el cálculo del refinamiento de contrato además tiene la capacidad de razonar propiedades de seguridad de los comportamientos de sistema y correlación de refinamiento; el refinamiento de diseño sienta bases en las teorías unificadas de programación y refinamiento de servicios en el diseño de resguardo, así como fallas en los procesos secuenciales de información y el modelo de divergencia: todo esto apunta a los modelos de UML bajo consideración principal.

El mantenimiento es lo suficientemente simple como para permitir la centrarnos de esfuerzo en el modelado y análisis de requisitos, no entrando tampoco en modelos de construcción como: diagramas de imagen, secuencia o colaboración. En ayuda del UML el refinamiento del servicio está mucho más cerca del puente con el modelado de requisitos del UML que otros posibles métodos basados en autómatas como Event-B y UPPAAL.

Extiende el artículo que: “Se muestran los elementos de UML y el refinamiento de servicios utilizados en el modelado de requisitos, junto con su correlación (…). Un modelo de requisitos de UML contiene un caso de uso diagrama, interfaces del sistema, diagramas de secuencia del sistema y un diagrama de clases conceptual. Las construcciones correspondientes en el refinamiento del servicio incluyen interfaces, especificaciones de interfaces, protocolos de interfaces, fallas de interfaces y divergencias de interfaces.” (Yioling, Wei, King, & Xiaoshan, 2019).

Aun así, el refinamiento de servicios no está diseñado de forma nativa para el modelo y análisis de requisitos, o al menos no en un estilo completamente orientado a

objetos. Ya que una interface de refinamiento de servicios solo consiste en servicios “publico” para poder manejar el UML se debe de especificar los requisitos con una visibilidad.

Volviéndolo pues en un prefijo público o privado, los cuales cabe aclarar se pueden agregar a un servicio; recordando que una operación de interface puede ser puede ser pública o privada, se da entonces una inclusión en la cual se den un conjunto se servicios los cuales pueden ser público o privados en una misma interface.

El sistema se restringe a que sus servicios deben pasar por sus servicios privados invocando los servicios en otro caso de uso de modo que todas las interacciones entre los casos de uso se puedan realizar como invocaciones de los servicios privados en casos de uso no activos. Dada la extensión de visibilidad anterior sobre el refinamiento de servicios se define como innovaciones infinitas de servicios privados; sin la extensión de visibilidad la coherencia en el refinamiento de servicios solo se considera en caso de que haya tres puntos muertos.

Se extiende la consistencia del contrato para tener en cuenta los “livelocks”, el contrato es coherente cuando existen algunos servicios fuera del conjunto de denegación después de un rastreo, en palabras más simples sin puntos muertos, pero que tampoco exista un rastreo que contenga invocaciones infinitas de servicios privados, sin bloque de medios. Los refinamientos se finalizan por contrato y mantiene el comportamiento del sistema, sin embargo, son más seguros en términos de consistencia.

En caso de los servicios privados cuales son invisibles para el sistema se ocultaron al considerar el comportamiento externo de un contrato con el fin de justificar la relación de refinamiento. El propósito del proyecto es proporcionar apoyo formal a UML a través del refinamiento del servicio.

Con el fin de que se lleve a cabo la verificación de los requerimientos de los contratos derivados de un modelo de requisitos y la verificación de la coherencia se presenta el puente desde el modelado de requisitos usando UML a las interfaces y contratos correspondientes en el servicio dado el caso de estudio de un sistema de compras en línea.

Especificando el algoritmo más profundamente como se puede pareciar en este párrafo: “Seguimos mejorando Ctr2 para que sea un contrato sin livelock Ctr3 agregando una variable MaxRepeats de tipo Integer a RDec (ManageAccountI), con una lógica de control que una vez número de invocaciones para repetir Invocación Alcance de pago MaxRepeats, por ejemplo, 3 veces, ¡abandonará la solicitud y devuelve un resultado predeterminado! depósito (falso). La variable wait0 se establece en falso en este caso, por lo tanto, el conjunto de rechazo correspondiente no contendrá todos los servicios públicos. Después de agregar la variable MaxRepeats y limitar el contrato Ctr3 de la interfaz ManageAccountI, ¿cuando Al solicitar el servicio? deposit (inNewBalance), el entorno eventualmente recibe la respuesta deposit (out) after no más de MaxRepeats veces de invocaciones al −repeatInvokingPayment servicio interno.” (Yioling, Wei, King, & Xiaoshan, 2019).

La contribución del proyecto es la siguiente: la extensión del refinamiento del servicio con prefijos de visibilidad y formulación de los ajustes necesarios a la teoría, Proposición de un enfoque sintético entre UML y SR mediante la integración del UML con el refinamiento de servicios.

Demostrar la efectividad de UML-SR para el modelado y la verificación de requisitos a través del estudio del caso. Los métodos formales proporcionan una forma rigurosa de razonar acerca de las propiedades críticas de un sistema, cuales son difíciles de garantizar solo mediante pruebas.

La integración entre métodos formales y herramientas de modelado populares es un buen comienzo para aprovechar las ventajas de ambos lados. Con la extensión de visibilidad de refinamiento del servicio, somos capaces de integrar UML y servicios para mejorar los requisitos de análisis y modelado.

Se muestra cómo se puede asignar el requisito de UML a la construcción en el reajuste del servicio a través de un estudio tópico: un sistema de compras en línea, además de obtiene el modelo de interface razonando sobre las propiedades de seguridad, saber cuáles son puntos muertos y bloqueos de vida, dando como resultado en la utilización del refinamiento del servicio para obtener un contrato sin trabas ni bloque de vida.

El proyecto se puede generalizar para ayudar a los desarrolladores a usar UML y el refinamiento de servicios juntos para presentar un modelo de requisitos coherente en la etapa inicial del desarrollo de tal modo que se puedan evitar desarrollos innecesarios. La

base del análisis formal de los requisitos de los requisitos con el método propuesto demuestra que se pueden realizar más contratos mediante enfoques orientados a objetos, basados en componentes y orientados a servicios.

En el artículo Generación de especificaciones de Mude a partir de diagramas de descripción general de interacción UML: un enfoque basado en la transformación de gráficos de Chafika Djaoui, Khaled Khalfaoui, Elhillali Kerkouche y Allaoua Chaoui se comienza escribiendo que el UML es el lenguaje de modelado escogido por los desarrolladores de software durante las primeras fases del desarrollo de un sistema como lo son: la ingeniería de requisitos, la arquitectura y el diseño detallado.

El articulo comienza de la siguiente forma: “UML 2.0 introdujo nuevos diagramas de comportamiento llamados Diagramas de descripción general de interacción (IOD) como un caso especial de Diagramas de actividad. El propósito de los IOD es llegar al rescate de casos de uso que probablemente sean lo suficientemente complejos como para tienen un escenario de éxito principal (muchos flujos alternativos). IOD flujo de control de visión general entre un conjunto de interacciones. Sin embargo, en lugar de acciones, diagramas de descripción general de interacción solo puede tener interacciones (que representan una secuencia completa Diagrama) o usos de interacción (ref; típicamente anónimo) que indican una actividad u operación a invocar. Interacción diagramas de resumen combinan el flujo de control de un Diagrama de actividad y especificación de mensajes del diagrama de secuencia. Las líneas de vida, los mensajes y combinados los fragmentos encontrados en los diagramas de secuencia (SD) no aparecen en el nivel de descripción general. Sin embargo, aparecen en un nivel inferior adjunto al nombre de la interacción donde ellos participar. Gráficamente, los IOD son gráficos conectados

diagramado con algunos elementos básicos derivados de la actividad Diagramas. Los nodos de interacciones de IOD están representados por marcos conectado con los bordes de flujo de control. El conjunto de nodos de control está compuesto por; Nodo inicial, nodo final, decisión / fusión Nodos y nodos Fork / Join.” (Djaoui, Khalfaoui, Kerkouche, & Chaoui, 2018).

Describe además que el lenguaje ofrece una colección de técnicas de descripción textual y grafica cuales en teoría deben ser fácilmente apreciadas por todos los desarrolladores de software. Las últimas versiones aprovecharon otros formalismos como las redes PN, las tablas de secuencia de mensajes, para introducir nuevos diagramas y conceptos como lo son: los x en la parte dinámica de los UML; los diagramas de información general de interacción son nuevos diagramas de comportamiento aprobados en la versión 2.0.

El propósito de los diagramas de información general de interacción es proporcionar un resumen del flujo de control en el sistema mediante la conexión en el flujo de control del sistema mediante una conexión de conjunto de interacciones a través de una variante de diagrama de actividad.

Incluyendo la lógica de flujo de control para navegar a través de las interacciones. A diferencia de los diagramas de actividad los de información general de interacción no representan el flujo de objetos, aun así, se central el flujo de control donde los nodos son interacciones o de uso de interacción.

Una interacción representa un diagrama de interacción donde el contenido se demuestra explícitamente; en el diagrama información general de interacción, mientras que un uso de interacción es una referencia a un diagrama de interacción existente separado; los diagramas globales de interacción tienen un poder expresivo que permite proporcionar una vista de alto nivel en la ejecución de la programación lógica de ejecución del sistema a través de un conjunto de integración.

En el tema 4 se escribe: “Contiene la declaración de un nuevo tipo llamado CONFIGURACIÓN que representa la corriente configuración de una instancia de Diagrama de descripción general de interacción. La configuración de un diagrama general consta de Nodo, nodo final e interacción (FRAME) que puede ser un uso de interacción o una interacción (SD). Además, este módulo define mensajes y líneas de vida y combina fragmentos dentro de cada fotograma (SD), así como las operaciones utilizadas para manipular elementos de configuración. Por ejemplo, en la operación nula define la configuración vacía. Nosotros también definir la operación START para iniciar una interacción (SD) en el nivel inferior, mientras que la operación END se define para finalizar interacción y volver al nivel de descripción general.” (Djaoui, Khalfaoui, Kerkouche, & Chaoui, 2018).

UML ha experimentado mejoras importantes a través de sus versiones, sus estructuras son de naturaleza sema formal y tiene una variedad de problemas, sin embargo, esta no tiene una matemática precisa, es decir, UML se mantiene lejos de estar completamente formalizado, debido a esto la integración de mentados formales a las notaciones UML pueden afectar significativamente el desarrollo de las semánticas precisas.

Dado el contexto del artículo se explora el uso de Mude como dominio semántico para promover las especificadores del UML, Mude es un lenguaje de alto nivel que admite la programación íntegra y declarativa estilo ocupacional de programación funcional como el cálculo lógico de la re escritura, admite la lógica de ecuación de membresía y la especificación lógica de reescritura de un sistema; el lenguaje de maude está dotado de un intérprete eficiente capaz de ejecutar diferentes tipos de especificaciones.

La lógica de rescritura es una lógica computacional simple que proporciona un marco semántico adecuado para la ocurrencia, dentro de la cual se puede representar una amplia gama de modelos computacionales; la lógica de rescritura esta parametrizada por su lógica de ecuación sub yacente que da lugar a reglas de rescritura. La teoría de rescritura para representar los aspectos dinámicos y estáticos de un sistema.

En el documento se propone un enfoque para formalizar los diagramas globales de interacción UML con el lenguaje Mude; esta formalización permite la especificación formal y el análisis de los comportamientos dinámicos del sistema. Para obtener un proceso automático y correcto de la generación de código maude se usa el sistema y gramática de transformación grafica para implementar, y definir la generación de código Mude.

En el tema 5 se escribe: “Se evalúa la utilidad del enfoque propuesto a través de un simple estudio de caso. La Figura 5 muestra un ejemplo de Diagrama de descripción general de interacción para cajero automático (ATM) que ofrece acceso a cuenta

bancaria. Representa el aspecto de comportamiento del cajero automático y muestra cómo una cuenta el titular puede realizar un retiro de efectivo después de la validación de la tarjeta y ser autenticado por el cajero automático. El diagrama comienza en inicial nodo, el retiro de fondos se realiza a través de una interacción (SD) que encapsula una alternativa combinada fragmento ALT. El posterior negó el retiro si los fondos son insuficientes, de lo contrario se autoriza la retirada. Los el flujo de control termina en el nodo final." (Djaoui, Khalfaoui, Kerkouche, & Chaoui, 2018).

Para la realización del enfoque se utiliza la herramienta de modelado Mude. Para la realización de nuestro enfoque se utiliza la herramienta de modelado de motos AToM3; definiendo primero el metamodelo simplificado para los diagramas de descripción general de integración UML, luego se utiliza esa herramienta de metamodelo para generar automáticamente una herramienta de modelo visual para los diagramas de descripción general de integración del UML de acuerdo con el modelado propuesto.

Para la generación de código Mude se ha definido una gramática de gráficos que traduce los diagramas de descripción general de interacción un creados en la herramienta generada a una especificación de Mude equivalente; la herramienta se puede usar para simulación además de análisis formal. Para el propósito del proyecto se propuso la formalización del IOD al tratar con múltiples actores y fragmentos confinados alternativos de la interacción SD.

El mapeo del proyecto ha permitido el modelo formal de los modelos fuente utilizando capacidades del lenguaje maude para ejecutar y validar la especificación generada por simulación.

Cristian L. Vidal-Silva, Rodolfo H. Villarroel, Xaviera A. López-Cortés y José M. Rubio en su artículo Una Propuesta de Algoritmo Spin / Promela para el Análisis y Diagnóstico de Errores en Diagramas de Secuencia UML: donde la tolerancia a fallas es una característica para mejorar la confiabilidad de un sistema de la información.

Tal como también lo anunciaron Adelmo en 1994 y Sukumar en 2007 una diferencia entre autores de trece años, esta tolerancia permite que el sistema siga funcionando así se den unas o varias fallas en el mismo, al principio esta se aplicaba a hardware pero siguió evolucionando en sus niveles lógicos.

Comenzando la introducción del documento de la siguiente manera: “Tal y como enuncian Adlemo (1994) y Sukumar (2007), la tolerancia a fallas es una característica para mejorar la confiabilidad de un sistema computacional. Según (Dubrova, 2013), la tolerancia a fallas es la propiedad que le permite a un sistema seguir funcionando correctamente en caso de falla de uno o varios de sus componentes. En un principio, la tolerancia a fallas se aplicaba principalmente en cuestiones de hardware, pero ya desde hace años también se aplica en contextos de software (Adlemo, 1994). El desarrollo de sistemas computacionales con un enfoque orientado a objetos requiere el uso de un lenguaje de modelamiento como UML. UML incluye diagramas para especificar, documentar y modelar elementos estructurales y de comportamiento de un sistema de información (Vidal et al., 2013; Visser et al., 2014; Vidal et al., 2014). Justamente, los diagramas de clase UML modelan la estructura del sistema de software (estructura de los principales componentes del sistema de software), mientras que los diagramas de secuencias y los diagramas de estado modelan el comportamiento del sistema de software

(Pender, 2003). Tal y como argumenta Visser et al. (2014), técnicas y herramientas prácticas para capturar y predecir el comportamiento esperado de un sistema de información.” (Vidal, Gutiérrez, & Mendivil, 2018).

Por otra parte el desarrollo de software requiere del uso de lenguaje de modelado como lo puede ser el UML, el UML incluye diagramas para modelar elementos de comportamiento y estructurales así como especificar, y documentar un sistema de información; siendo justamente los diagramas de clase del mismo UML que ayudan a modelar la estructura de un sistema, mientras que los de secuencia y estado modelan el comportamiento del mismo, de igual manera herramientas y técnicas prácticas que ayudan a predecir y capturar el comportamiento esperado de un sistema.

Para que se dé el modelo consistente de un software este se modela estructuralmente de acuerdo a sus modelos estructurales, dando como parte de esto entonces el proceso de modelado de comportamiento desarrollado con una metodología con desarrollo a objetos, UML permite el desarrollo de diagramas de estado y de secuencia. En el artículo se describe globalmente el diagrama de estado y secuencia.

Describiendo más ampliamente que: “En un desarrollo de software orientado a objetos, después de definir actores y casos de uso de la aplicación, y luego de establecer los principales elementos estructurales del sistema de software -clases con sus atributos y métodos- debe definirse un modelo de comportamiento de alto nivel del sistema de software. Por lo tanto, es necesario definir escenarios para describir y conocer las características de comportamiento de cada elemento estructural del sistema. En este contexto, los diagramas de secuencia UML permiten describir escenarios del sistema de

software, y saber cómo los objetos participantes interactúan y reaccionan bajo ciertas condiciones en esos escenarios (Pender, 2003).” (Vidal, Gutiérrez, & Mendivil, 2018).

Estos últimos permiten representar interacciones algorítmicas a través de los diagramas que corresponden a diagramas de secuencia secundarios dentro de un diagrama padre: una composición de elementos de arriba hacia abajo. En el artículo se expresa textualmente “una cuestión importante es la identificación de cada falla.

Así como su tratamiento en etapas iniciales del proceso de desarrollo como lo son la especificación y diseño de software”, dado este sentido spin/Promela representa un lenguaje de programación y una herramienta para el chequeo de modelos, así como la detección de fallas y la verificación formal del software.

Donde Promela es el lenguaje de programación y Spin es el intérprete. El objeto vio del artículo es aplicar y proponer para obtener el código Promela a partir de diagramas de secuencia UML y así mediante el uso de spin verificar la corrección de dicho código en detección de fallas en código no ejecutable, interbloqueos y comunicación.

Se promociona la utilización de diagramas de secuencia UML además de modelado de comportamiento como los diagramas de estado y colaboración UML cuales atribuyen al artículo.

Citando el artículo en el cual: “Se presenta un ejemplo simple de diagrama de secuencia UML como caso de estudio, principalmente ya que este considera o incluye los

elementos básicos de dicho diagrama, esto es, objetos, y mensajes síncronos y asíncronos de comunicación. Así, este ejemplo de aplicación se puede extender a otros diagramas sin la presencia de fragmentos combinados, para el análisis y tolerancia a fallos sobre dichos diagramas. Existen numerosos ejemplos de aplicación de diagrama de secuencias UML tales como la modelación de escenarios en un sistema Adaptativo de Control de Crucero (sistema ACC) (Ebnenasir y Cheng, 2006), o en un sistema de cuentas bancarias (Vidal et al., 2013).” (Vidal, Gutiérrez, & Mendivil, 2018).

Se concluyó que a partir del proyecto se modela y valida el comportamiento critico de un sistema que es de gran relevancia ya que constituye una herramienta de validación del funcionamiento y la semántica, se permite el refinamiento de los diagramas integrados, sin embargo, para una completa validación de la sintaxis de los diagramas de secuencia de UML respecto a los diagramas de clase, el detalle de los componentes de clase de los objetos participantes es absolutamente necesario.

Se establece un conjunto de pasos para la traducción de diagramas de secuencia en código Promela estos pasos se pueden implementar en una solución para para generar código Spin/Promela para garantizar y probar la corrección de los diagramas de secuencia. Finalizando, aunque Spin/Promela está hecho para soluciones distributivas pueden aplicarse de igual modo en no distributivas

El último artículo de los casos de éxito comienza describiendo que las herramientas existentes dirigidas al desarrollo de modelo no brindan el soporte adecuado para del sistema de depuración a nivel de modelo; debido a esto los desarrolladores generalmente deben consultar el código fuente que se genera.

Este comienza de la siguiente manera: “UML para tiempo real (UML-RT) [15, 17] es un lenguaje de modelado para diseñar sistemas integrados en tiempo real (RTE) con software en tiempo real con limitaciones. Ha sido la base de mucho trabajo académico, industrial proyectos y herramientas exitosas (por ejemplo, IBM RSA-RTE [9] y PapyrusRT [6]). UML-RT es un subconjunto de UML con conceptos dedicados para Diseño RTE y solo proporciona dos diagramas: cápsula y estado diagramas de máquina. Ambos diagramas son especializaciones de UML diagrama de estructura compuesta y diagrama de máquina de estado UML respectivamente. Las máquinas de estado UML-RT son versiones simplificadas de UML máquinas de estado, por ejemplo, no hay regiones de estado AND (ortogonal) en Máquinas de estado UML-RT.” (Bagherzadeh, Seekatz, Hili, & Dingel, 2018).

utilizar un depurador para su uso en las aplicaciones, este uso contradice los objetivos y principios del desarrollo dirigido por modelo porque se pierden los beneficios de la abstracción, y se introducen una nueva complejidad innecesaria, la generación del código generado puede ser propenso a errores y consumir mucho tiempo.

En especial para aquellos que no estén familiarizados con el objetivo del lenguaje o la generación de código, en otras palabras, el lenguaje de programación que se generó. Como estado de arte se escribe que la mayor parte de la literatura sobre depuración se basa en la técnica de depuración de servicios y programas de la depuración de mapas con sus resultados desde el nivel de modelo hasta el código.

Este enfoque puede tener los siguientes inconvenientes donde el más importante está relacionado con la necesidad de integrar el depurador a nivel de modelo con diferentes depuradores de programas, si estos admiten diferentes idiomas de destino; esta dependencia causa problemas de portabilidad y disminuye la reutilización de la instrumentación. El uso de un depurador de programas para la depuración de modelos también expone brechas semánticas entre el idioma de destino y el lenguaje de modelado.

"Creación de programas depurables. En lugar de usar el programa depurador y otras técnicas relacionadas, MDebugger utiliza el programa depurable que se genera a partir del modelo instrumentado. (…) utilizamos la transformación de modelo a modelo (M2M) técnicas para crear una versión instrumentada de un usuario definido modelo. El programa generado a partir del modelo instrumentado, por lo tanto, proporciona los servicios de depuración por sí mismo de la siguiente manera: (1) Servicio de control de ejecución para controlar la ejecución de un sistema en proceso. depurado usando comandos para establecer puntos de interrupción, suspender, reanudar, intervenir en la ejecución, etc. (2) Ver y cambiar atributos de servicios para inspeccionar y modificar el estado del sistema; (3) Generación del servicio de seguimiento de ejecución que proporciona una base para rastrear y analizar la ejecución; (4) Interfaz de depuración para permitir que los depuradores a nivel de modelo interactúen con el sistema depurado. El proceso de instrumentación se ha desarrollado utilizando Epsilon Object Language (EOL) [14]." (Bagherzadeh, Seekatz, Hili, & Dingel, 2018).

Esta brecha es sustancial y significa que algunos modelos del concepto no se pueden asignar fácilmente a los conceptos del lenguaje de programación correspondiente y en el mismo sentido inverso, la traducción no se puede ocultar. La solución propuesta

en Depurador: un depurador de nivel de modelo para UML-RT por Mojtaba Bagherzadeh, David Seekatz, Nicolas Hili y Juergen Dingel.

Para superar las limitaciones existentes es proponer un enfoque novedoso para realizar un de nivel de modelo independiente de la plataforma cual no depende de ningún depurador de programa específico de arquitectura.

Usando la trasformación del modelo para instrumentar un modelo a depurar con información que le permita soportar actividades de depuración. Como resultado de la instrumentalización del modelo el código generado a partir del modelo instrumentado es un programa depurarle cual puede proporcionar servicios de depuración. Se desarrolló un depurador a nivel de modelo que interactúa con programas depurables y proporciona servicios de depuración a nivel de modelo.

En el primer párrafo del tema 4 Evaluación se lee lo siguiente: “Tenemos que aplicó la instrumentación a varios modelos de sistema que son enumerados (…) y evaluamos nuestro enfoque utilizando tres métricas: sobrecarga de tamaño, tiempo de instrumentación y sobrecarga de rendimiento. Tiempo de instrumentación. Para medir el tiempo de instrumentación y verificar que nuestro enfoque sea escalable, medimos el tiempo requerido por la herramienta para crear la versión instrumentada del modelo a partir del cual se genera el código. Para cada caso de uso, medimos tiempo de instrumentación 20 veces y calculó el promedio. (…) muestra que este promedio varía entre 445 y 1.523 milisegundos según el modelo del sistema, que es dentro del rango de un segundo y una buena indicación de escalabilidad de acercarse.” (Bagherzadeh, Seekatz, Hili, & Dingel, 2018).

como conclusión se aprecia que se amplía gradualmente para admitir la depuración de modelos parciales, siendo uno de los primeros que aborde ese problema.

Ventajas y desventajas del UML

En las diapositivas Lenguaje Unificado de Modelado – UML de Patricia López y Francisco Ruiz se escriben algunas ventajas y desventajas del UML, en este se comienza escribiendo de las ventajas donde se relata que El UML es estándar y por tanto es fácil la comunicación por este; se basa en metamodelo con una semántica bien definida.

La notación grafica en la que se basa en concisa, fácil de utilizar, así como de aprender; se puede modelar para ser usado en diversos dominios como lo pueden ser sistemas de información empresariales, sistemas web, sistemas críticos y tiempo real, inclusive se puede usar en sistemas que no son de software; es fácilmente extensible.

Dentro de los inconvenientes de el mismo documento tratado anteriormente se escribe lo siguiente: no es una metodología, es decir que además del UML hace falta una metodología proyectada a objetos; no cubre todas las especificaciones requeridas como lo son los documentos textuales o el diseño de interfaces de usuario; faltan ejemplos elaborados en la documentación; llega a ser complejo alcanzar un conocimiento completo del lenguaje.

Estos mismos especifican que hay otros errores en otras metodologías como la lectura de código de la cual señalan: "• Representa sólo la lógica e ignora el resto.
• El ser humano lo interpreta muy lentamente.
• No facilita la reutilización ni la comunicación." (Lopez & Ruiz, 2019).

También escriben algunas desventajas del método textual de la factorización de código:

“• Es ambigua y confusa

• Es lenta de interpretar

• Difícil de procesar.” (Lopez & Ruiz, 2019).

Fernando Berzal en su documento El lenguaje unificado de modelado especifica como ventaja la unificación de distintas notificaciones previas como lo son las de Rumbaugh, Jacobson, Meyer, Harel, Wirfs—Brock, Fusion, Embly, Gamma, Shlaer-Mellor, Odell, Booch, así como algunos otros no nombrados.

Como desventaja describe: “Falta de integración con otras técnicas (p.ej. diseño de interfaces de usuario).
UML es excesivamente complejo (y no está del todo libre de ambigüedades): "el 80% de los problemas puede modelarse usando alrededor del 20% de UML"”. (Berzal, 2005).

En la tabla No.1 se tratan las ventajas y desventajas del UML en el cual de detallan seis ventajas, así como desventajas que tiene el modelo frente a otros que también podrían utilizarse o la implementación en el desarrollo de un proyecto informático.

Tabla No.2: ventajas y desventajas del UML

Ventajas	Desventajas
Es estándar esto facilita la comunicación	No es una metodología en si misma de desarrollo

Se basa en meta modelos con semánticas bien definidas	No cubre todos los requisitos
La notación es fácil de aprender y utilizar	No se pueden hacer ejemplos elaborados en la documentación de los requisitos
Es fácilmente extensible	Es complejo llegar a conocer todo el lenguaje
Se puede usar en cualquier tipo de proyecto	Falta de integración con otras técnicas
Unificación de las notaciones previas de los autores del UML	Ambigüedad en el desarrollo de los diagramas

Herramientas para el UML

Para este apartado se sacó información de las paginas ionos, ingenieriadesoftware, aprendeaprogramar, Stadium UNAD y programas para pc, además se encontró que las herramientas más populares para el desarrollo de UML de manera virtual son ArgoUML, Lucid Char, StarUML, Magic Draw, Microsoft Visio, Papyrus UML, Modelio, presentando la información en la ilustración No.94:

ionos.es	ingenieriadesoftware.es	ingenieriadesoftware.es	aprenderaprogramar.com	stadium.unad.edu.co	programasparapc.net	frecuencia	
Gliffy						1	
ArgoUML	ArgoUML			ArgoUML		3	
MagicDraw	MagicDraw					2	
Lucidchart		Lucidchart	Lucidchart			3	
IBM Rational Rhapsody						1	
Microsoft Visio			Microsoft Visio			2	
	Papyrus UML				Papyrus UML	2	
	Modelio				Modelio	2	
	StarUML			StarUML	StarUML	3	
		Gennymodel				1	
		Draw.io				1	
		Creately				1	
		Cacoo				1	
		Umletino				1	
		Diagramo				1	
		Editor Jsuml2				1	
		BPMN				1	
		vertabelo				1	
			Astah community			1	
			Rational Rose			1	
				Día		1	
				Frame UML		1	
				Tiny UML		1	
				Eclipse		1	
					UMLet	1	

Ilustración No.94: tabla de frecuencia en la mención de herramientas UML.

Para la escritura de herramientas UML se tomarán todas las herramientas que se mencionaron en más de una ocasión en todos los sitios consultados.

ArgoUML:

Es una aplicación programada en java que sirve para el diagramado del UML, el cual fue publicado bajo la licencia EPL en abril de 1999, debido a que es programada en java está disponible en cualquier plataforma que lo soporte en 2003 fue una de las

finalistas en la categoría de Design and Analysis Tools en el El Magazine de Desarrollo de Software, también recibió un premio "runner-up".

Este no está completamente conformado de acuerdo al estándar UML dejando de recibir actualizaciones en la versión 0.20, y carece por completo de herramientas para la diagramación de colaboración y secuencias.

La página de Ionos la defina así: “ArgoUML ha sido durante mucho tiempo una de las herramientas UML gratuitas de código abierto más populares para el escritorio. Aunque ya no se mantiene, muchos modeladores continúan usando el programa para tareas más pequeñas. Software multiplataforma, el requisito mínimo es Java 5 ArgoUML soporta todos los tipos de diagramas de la versión 1.4 de UML y perfiles UML. El programa también ofrece algunas formas decorativas que no forman parte del estándar UML: si utilizas estos formularios, te desviarás del estándar UML. Por lo tanto, asegúrate de que esto no cause ningún problema de comprensión posteriormente. Utiliza OCL (Object Constraint Language) para asignar información restrictiva a un modelo.”(Ionos, 2019).

Cabot expresa en la página ingeniería de software: “Ok, ahora pensaréis que me he vuelto loco. ¿Qué hace ArgoUML aquí si es una m… de herramienta? Cierto, hoy en día, hay pocas herramientas más feas que ArgoUML y hace muchos años que no se actualiza. Pero ArgoUML gana en la categoría “mejor herramienta por razones sentimentales”. Fue la primera herramienta que usé y era de lo mejorcito (y además en software libre) de lo que había en esos momentos. Creo que se merece que la recordemos aunque sólo sea por eso.”(Cabot, 2019).

LucidChart:

Permite su acceso en navegadores que usan el estándar HTML 5, por consiguiente, no requiere de plataformas de terceros como flash, debido a esto ha de saberse que se ejecuta en la web; su uso es para permitir a los usuarios colaborar en compartir, revisar y dibujar diagramas y cuadros.

Para el 2010 se integró en google apps Marketplace; logro recaudar un millón de dólares en "ángeles inversionistas"; para 2017 esta empresa había recaudado 72 millones de dólares adicionales de Meritech Capital e ICONIQ Capital.

"Herramienta que permite crear muchos tipos de diagramas, entre ellos UML." (Krall, 2019).

"Basado en HTML5. Con soporte para UML. Permite también la colaboración online en tiempo real. Puede importar ficheros Visio con lo que es una buena alternativa para aquellos equipos que estén buscando una alternativa más ágil a Visio y que sea basada en web. Además de UML, también incluye plantillas para crear modelos ER, procesos de negocio, diagramas de red y muchos otros tipos de modelos." (Cabot, 2019).

"Lucidchart es una herramienta UML a la que se puede acceder en el navegador, así como a través de Android e iOS. La cuenta gratuita te da paso a un paquete de herramientas UML muy completo. Incluye 7 tipos de diagramas UML y lenguajes de modelado de procesos de negocio como BPMN 2.0, plantillas de iconos de red, maquetas

de dispositivos móviles e integración de vídeo. Una de las ventajas de Lucidchart es su funcionamiento intuitivo. También permite compartir y editar simultáneamente diagramas en equipo e integrar comentarios directamente en la herramienta. Como herramienta de modelado UML compatible con MacOS, es una buena alternativa a Microsoft Visio para usuarios de Apple.” (Ionos, 2019).

Star UML:

Es una herramienta desarrollada por MKLab para el desarrollo de UML, este fue licenciado bajo una versión modificada de GNU GPL hasta 2014 cuando se lanzó la versión 2.0 la cual fue re programada para betas bajo licencia patentada. Este paso de Delphin a Eclipse solo para abandonarse nuevamente; aunque la versión de código abierto sigue actualizándose.

El objetivo de esta aplicación es el remplazo de programas como Borland Together y Rational Rose los cuales son más grandes que StarUML; este admite la mayoría de diagramas del UML 2.0 faltándole diagramas de resumen, sincronización e interacción. Esta aplicación se descargó en Delphin lo que es una de las razones por las cuales se abandonó durante mucho tiempo, haciendo que no se actualice desde 2005.

La versión más reciente de los autores se puede descargar como StarUML2. La versión beta publica está disponible, aunque no bajo la licencia de GPL, aunque el nuevo tipo de licencia y el precio final aún se desconocen, esta versión incluye muchas características nuevas como los son: el soporte para extensiones, compatibilidad para el sistema operativo X de mac, y una nueva interface de usuario. Siendo re escrita desde 0.

“Si Grady Booch mismo habla bien de ella, tenía que poner StarUML en la lista. StarUML es la mejor opción si buscas una herramienta rápida, fácil de usar y razonablemente barata en comparación a otras herramientas UML.” (Cabot, 2019)

“StarUML es buenísima para los que recién entran a este mundo es fácil de usar, intuitiva, rápida y sus extensiones son netamente económicas. Podrás trabajar cómodamente obteniendo grandes resultados.” (programasparapc, 2019).

MagicDraw:

Herramienta para el desarrollo de diagramas UML desarrollado por NoMagic, catalogada como una herramienta CASE “Ingeniería de Software Asistida por Computadora”. Esta es compatible con la versión 2.3 de UML permitiendo el desarrollo de código para diversos lenguajes como los son c++, c#, java, entre otros; así como para modelar datos. La herramienta cuenta con capacidad para trabajar en equipos y es compatible con los siguientes entornos de desarrollo integrados.

“Me encanta su usabilidad. Pero aún me gusta más su motor de ejecución de modelos UML. NoMagic (la empresa detrás MagicDraw) ha sido comprada por Dassault Systems. Es de prever que, como consecuencia, MagicDraw siga mejorando en todo lo que se refiere a la ingeniería de sistemas donde este tipo de simulaciones a partir de modelos es clave.” (Cabot, 2019).

"MagicDraw de No Magic es la primera versión completa para el modelado profesional que ofrecemos en este listado. Esta aplicación de escritorio destaca por su diseño moderno y claro, así como por su variedad de funciones y la facilidad de su uso. Esta herramienta de diagramas UML ofrece además SysML, representación gráfica de procesos de negocio con BPMN (Business Process Model and Notation) y el marco de arquitectura UPDM (United Profile for DoDAF/MODAF). En MagicDraw se trabaja con los diagramas actuales según el estándar UML 2.5, cuyos perfiles se pueden adaptar a tus propias necesidades. MagicDraw también ofrece lenguaje de especificación OCL (Object Constraint Language), y XMI, que puedes usar para exportar diagramas a otros programas sin pérdidas de información." (Ionos, 2019).

Papyrus UML:

Es una herramienta para el diagramado de UML de código abierto bajo la licencia EPL y basado en eclipse, este ha sido desarrollado por LISE (Laboratorio de Ingeniería Modelo para Sistemas Embebidos) el cual forma parte de omisión Francesa de Energías Alternativas y Energía Atómica. Esta herramienta sirve como complementación de eclipse Proporciona soporte para lenguaje de dominio y SysML. Está diseñado para ser fácilmente extensible ya que se basa en el principio de los perfiles UML.

"Papyrus UML tiene una característica que lo diferencia del resto y es que este programa es el aliado perfecto para aquellos que les agrada trabajar en Eclipse, además de que te ofrece la posibilidad de integrar plug-ins como parte del desarrollo." (programasparapc, 2019).

"El entorno de modelado estándar "de facto" en Eclipse. Gratuito y open source, Papyrus es sin duda la mejor opción si trabajas con Eclipse o necesitas integrar tus modelos con otros plug-ins de Eclipse como parte de tu proceso de desarrollo. Te acepto que Papyrus no es la herramienta más intuitiva ni fácil de usar, pero se está esforzando para revertir la situación. Por ejemplo, recientemente ha sacado versiones especializadas para escenarios de uso concretos (e.g. Papyrus for Information Modeling o Papyrus for real-time)." (Cabot, 2019).

Modelio:

Es una herramienta UML de código abierto desarrollada por ModelioSoft el cual tiene su sede en Francia en la ciudad de parís, este no solo es compatible con UML sino además con BPMN, fue lanzado a publico el 5 de octubre de 2011 bajo la GPLv3, las API clave se otorgan bajo la versión apache 2.0 más permisiva.

Esta es una de las 6 herramientas que participaron en la demostración de interoperabilidad realizada por el grupo de trabajo de intercambio de modelos en diciembre de 2009, además Mades piensa usar esta herramienta para el desarrollo de nuevas anotaciones para aplicaciones de aviónica y vigilancia.

"La interfaz dinámica que tiene Modelio te da infinitas posibilidades para que trabajes UML, tales como recursos y herramientas descargables para ampliar tu trabajo. También, agregar soporte de requisitos y análisis de datos directamente desde el programa gratuito de Modelio." (programasparapc, 2019).

“Herramienta muy potente, organizada en un núcleo open source al que se le pueden añadir funcionalidades mediante un sistema de extensión modular. Algunos de los modelos son también gratuitos pero muchos son ya extensiones comerciales, disponibles en la modelio store. Esta estructuración te permite adaptar la herramienta a tus necesidades de modelado UML. Por ejemplo, puedes empezar modelando gratis tu sistema y si luego decides utilizar esos modelos para generar código para la plataforma que sea, comprar la extensión correspondiente.” (Cabot, 2019).

Metodologías Agiles

Scrum

Ilustración No.95: tablero Scrum.

Representación de una tabla de Scrum en la ilustración No.27 en la cual se evidencian las actividades no planeadas, no iniciadas, en ejecución y las que ya han sido hechas divididas en siete columnas horizontales.

Scrum es una metodología ágil para la gestión de proyectos informáticos relacionado con la construcción de este. Se da pues la primera definición de esta forma de gestión llamado sprint el cual tiene dos características: la duración debe ser fija de una a cuatro semanas y cada sprint se ejecuta de manera consecutiva. El objetivo general es

trasformar un conjunto de “objetos” requeridos por un cliente en un software incremental y 100% operativo.

Se sostiene por parte de uno de los creadores la implementación de tres pilares: transparencia, inspección y adaptación. En el marco de trabajo de Scrum se componen una serie de reglas que definen las ceremonias que deben respetarse, dan bloques de tiempo pre establecidos, los artefactos necesarios para los procesos y los roles que integran los equipos; los roles bien diferenciados de esta metodología son el Scrum master, el dueño del producto y el equipo de producción.

Como se puede leer en el libro Scrum y extreme para programadores de Eugenia Bahit: “Según Jeff Sutherland, uno de los creadores de Scrum y del método de control empírico de procesos en el cual se basa dicha metodología, éste se sostiene en la implementación de tres pilares: transparencia (visibilidad de los procesos), inspección (periódica del proceso) y adaptación (de los procesos inspeccionado).” (Bahit, 2011).

Hay una cantidad de bloques de tiempo interactivo los cuales duran de dos a cuatro semanas en las cuales se destina la creación de continuidad y regularidad, basadas en seis ceremonias cuales aseguran el cumplimiento de objetivos estas son: reunión de retrospectiva, revisión, diaria, Sprint, planificación de Sprint y planificación de entrega. Se emplean cuatro artefactos como herramientas: back log del producto, back log Sprint, Scrum task board, diagrama de Burdow.

Se definen más específicamente los roles de Scrum comenzando por el dueño del producto quien es la única persona para definir cuales son las características funcionales y

funcionalidades del producto, representado por el cliente usuarios de software y alguna otra parte interesada en el producto, entre sus funciones y responsabilidades se establece que debe canalizar las necesidades del negocio, escuchando a las partes interesadas y al equipo que desarrollara el producto.

En el libro Kanban y Scrum – Obteniendo lo mejor de ambos se puede leer: "Scrum prescribe 3 roles: dueño de producto (establece la visión del producto y las prioridades), equipo (implementa el producto) y ScrumMaster (elimina los impedimentos y proporciona liderazgo en el proceso)." (Kingberg & Skarin, 2010).

Como aptitudes se espera que se tenga una excelente facilidad de comunicación en relaciones interpersonales, un excelente conocimiento del negocio, facilidad para analizar el costo beneficio y tener vías de negocios. El Scrum master es el alma de la metodología, un error frecuente es llamarlo líder, ya que este no es un líder típico siendo más bien un servidor neutral cual será encargado de intuir y fomentar los principios agiles del Scrum.

Entre sus responsabilidades están la aplicación correcta del Scrum, resolver los conflictos que entorpezcan el progreso del proyecto, motivar e incentivar al Scrum team, entre las aptitudes se deben aplicar excelentes conocimientos del Scrum, amplia vocación a servicios, tendencia altruista, amplia capacidad para la resolución de problemas, ser observador y analítico, saber motivar e incentivar, capacidad instructiva y docente, buen carisma para las negociaciones.

Las actitudes negativas que debe evitar son: entrometerse en la gestión del equipo sin dejar definir su propio proceso de trabajo, asignar tareas a uno o más miembros al

equipo de trabajo, colocarse a actitud de autoridad frente al equipo, negociar con el dueño del producto la calidad del producto a desarrollar, delegar la resolución de un conflicto al miembro del equipo o a un tercero.

En las diapositivas Desarrollo ágil con Scrum se describe que los roles de un scrum master son el ser: “responsable del proceso de Scrum.

• Incorporación de Scrum en la cultura de la organización.

• Asegura el cumplimiento de los roles y responsabilidades.

• Formación y entrenamiento en el proceso.” (Álvarez, 2015).

El equipo de desarrollo de Scrum es multidisciplinario, integrado por programadores, el cual puede estar integrado por programadores, diseñadores de software, arquitectos del sistema, testers, entre otros, cuales en forma auto organizada serán los encargados de desarrollar el producto, entre sus funcionalidades y responsabilidades esta convertir el backlog en un software potencialmente entregable, ser profesionales expertos y avanzados en su disciplina, tener vocación, capacidad de auto gestión.

Algunas actitudes personales que se deben considerar, y cuáles deben ser evitadas dentro del equipo son la solidaridad y colaboración, motivador y no pretender sobre salir y evitar la competencia. El Scrum propone tres herramientas o artefactos para mantener organizados los proyectos que hagan uso de su metodología.

Estos ayudan a planificar y revisar cada uno de los Sprints, aportando medios para efectuar cada una de las ceremonias, se escribe sobre el backlog del Sprint y el producto, el Scrum del taskboard y los diagramas de burndown.

Eugenia describe que: “Scrum, propone tres herramientas o "artefactos" para mantener organizados nuestros proyectos. Estos artefactos, ayudan a planificar y revisar cada uno de los Sprints, aportando medios ineludibles para efectuar cada una de las ceremonias que veremos más adelante. Ahora, nos concentraremos principalmente, en el backlog de producto, el backlog de Sprint y el Scrum Taskboard, para luego hablar brevemente sobre los diagramas de Burndown.” (Bahit, 2011).

El backlog del producto se describe como un listado públicamente visible y dinámico para aquellos involucrados en el proyecto, en este el dueño mantiene una lista de requerimientos funcionales para el software, en esa lista se representa que es lo que se pretende, pero sin mencionar como hacerlo debido a que de esto se encarga el equipo de desarrollo.

Para el backlog del producto es una lista de ítems que representa los requerimientos funcionales esperados para el software para cada uno de estos ítems será necesario de la especificación de los criterios de aceptación, granularidad, esfuerzo que demanda y el grado de prioridad.

Estos ítem mencionados anteriormente se obtienen sus características del cual se escribe en primera instancia de la priorización de los ítems cual especifica: los ítems deben guardar un orden prioritario teniendo en cuenta la base de cuáles son los valores

diferenciales con respecto al producto de la competencia, coherencia con los intereses del negocio del dueño, los riesgos de la implementación, costo o perdida que demande posponer la implementación de una funcionalidad, y los beneficios de implementar una funcionalidad.

Scrum a diferencia de metodologías tradicionales propone la estimación de esfuerzos y complejidad en el desarrollo de funcionalidades, esto solo para aquellas que sean prioritarias, luego no se estima sobre aquello poco prioritarios, o que estén en incertidumbre, dando a lugar evitar pérdidas de tiempo en estimaciones irrelevantes postergándolas en los momentos en el cual realmente sea necesario comenzar a desarrollarlas.

Los ítems del producto no tienen por qué tener una granularidad pareja, los ítems de baja gradualidad suelen agruparse en el formato historias de usuario mientras que los altos como epics o temas, una historia de usuario puede describirse en tal forma que "como usuario, puedo hacer tal acción o funcionalidad para tal beneficio".

En muchas ocasiones puede resultar redundante o incluso carecer de algún sentido indicar el beneficio de una funcionalidad, debió a esto es frecuente describir las historias de usuario sin incorporar el tercer elemento mencionado, como nota adicional se da la frecuencia de encontrar acciones como quiero o necesito en vez de puedo cuando se describe una historia de usuario.

Álvarez simplifica el backlog de la siguiente manera: "Listado con los requisitos del sistema

• Mantenido y priorizado por el Prod el Product Owner

• Documento dinámico que incorpora constantemente las necesidades del sistema

• Se mantiene durante todo el ciclo de vida.” (Álvarez, 2015).

Se recomienda que cada ítem backlog del producto de especificación de cuáles son los criterios de aceptación para considerar como cumplido un requisito, luego los criterios de aceptación no son más que pequeñas reglas o pautas que una historia de usuario debe respetar para ser considerada cumplida.

El backlog Sprint es una lista reducida de backlog de producto, cuáles han sido negociados entre el dueño del producto y el equipo de desarrollo durante la planificación del sprint, la lista se genera al comienzo de cada sprint y representa las características que el equipo se compromete a desarrollar durante la interacción actual.

Estos ítems se dividen en tareas las cuales por lo general no demandan una tarea superior a un día a un solo miembro del equipo, esta se actualiza diariamente y muestra las tareas pendientes e cursos y terminadas, el nombre del miembro del equipo y al que se le ha asignado dicha tarea, y la estimación del esfuerzo pendiente de cada tarea sin terminar, cada tarea debería ser etiquetada.

La división de usuarios en tareas cosiste en desmembrar el ítem a la mínima expresión encuadrada en un mismo tipo de actividad, este desmembramiento se hace de lo general a lo particular, y de allí al detalle: para especificar más estos aspectos se da a conocer que como análisis general se puede responder a la pregunta que es, en el

particular se responde el cómo hacerlo, y en el detallado que tareas se necesitan hacer para lograrlo.

Eugenia escribe textualmente: “La estrategia consiste en desmembrar el item a la mínima expresión posible, encuadrada en un mismo tipo de actividad. El desmembramiento debe hacerse "de lo general a lo particular, y de lo particular al detalle".” (Bahit, 2011).

Estas tareas conseguidas se plasman en diferentes posts it, los mismos miembros del equipo deciden que tareas se asignara cada uno y se colocara los posts it en el tablero; al finalizar cada Sprint, el equipo hará entrega de un incremento de funcionalidad para el software, este incremento debe asemejarse a un software funcional el cual deberá ser implementado en un ambiente de producción a su cien por ciento.

Es frecuenta oír hablar de ceremonias cuando se hace referencia a las cuatro reuniones que realizan de forma interactiva cada sprint, las cuales son retrospectiva, revisión, reunión diaria y planificación de sprint. La planificación es lo primero que debe hacerse con cada sprint, en esta participan el equipo de desarrollo, el Scrum master y el dueño del producto.

El objetivo es que el dueño pueda presentar al equipo la historia de los usuarios prioritaria comprendidas en el backlog del producto, el equipo comprenda el alcance del mismo mediante preguntas, y luego se pase a una negociación de cuales se pueden desarrollar en el sprint, cuando se define el alcance del sprint es cuando el equipo se

divide cada historia del usuario en tareas las cuales serán necesarias para desarrollar la función que se requiere.

Tales tareas tendrán un esfuerzo de desarrollo estimado, tras lo cual serán pasadas por el backlog sprint y de allí se visualizará en el tablero una vez cada miembro se haya auto asignado como apto, la dinámica de la planeación basa en presentar los ítems, estimar esfuerzos, definir el alcance, negociar los requisitos, la definición de tareas y armar el tablero.

Una característica que se define de esta metodología es que: “Un tablero Scrum es propiedad de solamente un equipo Scrum. Un equipo Scrum es multi-funcional, contiene todos los conocimientos necesarios para completar todos los elementos de la iteración. Un tablero Scrum suele ser visible para cualquiera que esté interesado, pero sólo el equipo Scrum al que pertenece puede editarlo - es su herramienta para administrar su compromiso para esta iteración.” (Kingberg & Skarin, 2010).

Las reuniones diarias son conversaciones de entre 5 y 15 minutos las cuales se darán cada día con cada miembro del equipo, en esta, cada miembro del equipo deberá ponerse al día con lo que cada miembro ha desarrollado, lo que hará en la fecha actual, cuales impedimentos le estén surgiendo, con el fin de resolverlos y que el Scrum pueda continuar, como dinámica adicional se debe dar la actualización del tablero, el artefacto involucrado será el backlog Sprint.

Durante la ceremonia de revisión en Scrum, el equipo presentara al dueño las funcionalidades desarrolladas, hará una demostración de ellas y las explicara, con el fin

de que el dueño y la eventual audiencia puedan experimentarla, el dueño podrá sugerir mejoras a las funcionalidades, aprobarlas o rechazarlas si se considera que no cumple con el objetivo, esta celebración se lleva a cabo el último día del sprint, y no tiene una duración fija, puesto que se utiliza el tiempo que se crea necesario, entre las dinámicas se da además la actualización del backlog de producto y el acurdo de un próximo encuentro.

Como ultima ceremonia se da la búsqueda de la perfección en la cual los objetivos son detectar la fortaleza del equipo y oportunidades de mejorar y acordar opciones concretas de mejoras, como participantes se incluye el equipo de desarrollo, Scrum master y opcionalmente el dueño del equipo, este se da al finalizar el último día del sprint, la duración de esta ceremonia es fija y no debe durar más de dos o tres horas, no se ven artefactos del mismo Scrum involucrados.

La dinámica se ensaña en la identificación de oportunidades de mejora, la definición de acciones concretas y la revisión de acciones pasadas. Sobre la estimación de esfuerzo en Scrum establece que a diferencia de las técnicas de estimación tradicionales cuales se centran en el tiempo, el cual demanda la realización de tareas inciertas las metodologías agiles en general proponen la generalización de esfuerzos ya que seguramente es mucho más certero, y fácil indicar el esfuerzo de una actividad que el tiempo estimado del mismo basado en una sola variable de la actividad.

“Estimar el -esfuerzo- es algo que resulta independiente de la cantidad y calidad de los factores del entorno. Pues para cualquier pintor, pintar el edificio de 14 pisos demandará mucho esfuerzo. Incluso aunque los pintores designados a realizar la tarea, sean seis. Sin embargo, estimar el tiempo, es un factor que además de poco certero, es

inherente a los elementos del entorno. Pues es fácil decir que pintar el edificio demandará mucho esfuerzo, pero calcular el tiempo que demande hacerlo, implicarán factores como la cantidad de pintores involucrados, la calidad de la pintura, las herramientas que se utilicen, el estado climatológico, entre decenas de otros factores" (Rivadeira, Vilanova, Miranda, & Cruz, 2013).

El agilísimo se avoca a estimar esfuerzos certeros basándose en un principio de honestidad, mediante el cual, los miembros del equipo se comprometes a ser sinceros, teniendo una humildad suficiente y necesaria, la cual es requerida para conocer sus propias limitaciones.

Existen varias técnicas para estimar esfuerzos en el libro se hablan de tres: la estimación por columnas, la estimación por póker y el T-Shirt sizing; la última se basa en la tabla de medidas americanas que utiliza las prendas de medir, las cuales van desde el XL pasando por L, M, S y finalizando por XS, debido a esto mientras más pequeña es la medida menor es el esfuerzo en cumplir con los valores que requiere la historia de un usuario.

Eugenia escribe que existen diversos tipos de técnicas de planificación: "planificación que hace que estimar, sea algo divertido.

Aquí hablaremos de 3 técnicas:

• T-Shirt Sizing

• Estimación por Poker y

• Estimación por Columnas" (Bahit, 2011).

La aplicación de la metodología se desarrolla de la siguiente manera: un miembro del equipo lee una historia de usuario, cada miembro del equipo lee lo que considera un esfuerzo requerido para su desarrollo, luego todos al mismo tiempo giran el papel haciéndolo visible, se indica la media del esfuerzo requerido, el miembro que mayor, así como el que menor estimación haya dado argumenta la estimación.

Se puede elegir la estimación desarrollada por la mayoría o dando capacidad al Scrum master para que escoja una estimación, o haga que se vote de nuevo, también podrá pedir una nueva argumentación, lo ideal se indica cómo dar un consenso.

El Scrum con póker se juega con una baraja de cartas numeradas siguiendo una serie de Fibonacci, mientras menor sea el número menor será el esfuerzo que demanda una historia del usuario, pero mientras más elevado sea el número mayor esfuerzo necesitara, el Scrum póker cuenta con dos cartas más una la cual significa no dar más "una taza de café" y no estar seguro del esfuerzo que demanda una actividad.

Un entendimiento no completo de los requerimientos dado por un signo de interrogación, se dan reglas de juego como definir el tiempo de descanso previsto si un miembro del equipo de cansa, decidir cuánto será el tiempo de exposición tendrá cada miembro por cada estimación y el Scrum master puede actuar como moderador.

Para poder jugar se debe de hacer la exposición en la que se vuelva a votar repitiendo la discusión del alcance, cada miembro debe tener su propio juego de cartas, una vez todos los miembros hayan bajado una carta sobre la mesa le darán la vuelta y si todos los miembros han arrojado la misma carta debe indicarse el mismo.

Se añade de Álvarez que: “En Scrum, los equipos tienen que estimar el tamaño relativo (= cantidad de trabajo) de cada elemento al que se comprometen. Sumando el tamaño de cada elemento completado al final de cada sprint, obtenemos la velocidad. La velocidad es una medida de la capacidad - la cantidad de cosas que podemos ofrecer por Sprint. He aquí un ejemplo de un equipo con una velocidad media de 8.” (Álvarez, 2015).

El esfuerzo ha sido estimado, después de esto se levanta la carta de la mesa y se pasa al siguiente backlog, si uno de los miembros da un número mayor estará viendo un impedimento que otros no y expondrá su argumento, también expondrá el que menor esfuerzo haya estimado.

Si no se llega a un acuerdo en la tercera ronda el Scrum master elegirá la estimación final, este finalizará cuando no haya más historias de usuario. La estimación en columna está pensada para estimar grandes cantidades de historias de usuario en bloque y su objetivo es agrupar las historias de los usuarios con el mismo grado de granularidad. La cantidad de esfuerzo requiere es directamente ligada a la posición izquierda de la columna, puesto así, mientras más a la derecha se encuentran los requisitos con menos demanda, los miembros del equipo deben estar sentados en una mesa.

Uno de los miembros debe tomar las historias de usuario, extrae una de las historias de usuario del mazo, este la lee en voz alta ubicándola en la mesa, formando una columna imaginaria; se vuelve a extraer una historia de usuario, la ubica en la mesa bajo

las condiciones de que si la historia demanda menor esfuerzo de la que se encuentra en la mesa deberá colocarla a la izquierda.

En caso de que demande menos esfuerzo se ubica abajo, por ultimo pasa la fila al ubicado en la izquierda, a partir del tercer jugador se tiene en cuenta que, el jugador que posee la pila debe escoger entre pasar de fila una historia ya escogida y pasar la pila al siguiente jugador además de estos el extrae una nueva historia de usuario la lee en voz alta, y la coloca en la mesa, cuando en el juego ya no queden más historias de usuario en la pila, se asignan todas la historias de una misma columna, el mismo grado de esfuerzo dando finalización al juego.

Otro juego de estimación de esfuerzos es la estimación por columnas y póker de forma combinada, este se describe con un máximo de cinco columnas, después se realiza una estimación por póker para cada columna debajo de las siguientes pautas: las estimación comienza en la fila del medio, como baraja se propone contar con las cartas del medio al infinito, dándose la serialización de Fibonacci, la taza del café, la interrogación y exclamación, si la carta es infinito significa entonces que el esfuerzo es mayor a 21, si se da el signo de exclamación el esfuerzo es demasiado grande para ser medido por una sola persona.

Se toma cualquier historia del usuario y se estima mediante Scrum póker, para estimar la columna del medio se dejará fuera las cartas de más alto nivel, así como las de más bajo nivel, el valor que se asigne se dará para todas las historias de usuario, se repite el procedimiento con las columnas restantes, cuando todas las columnas sean estimadas se acaba el juego.

De forma similar se describe que el Spring Planning 2: “Sprint Planning 2 – Reunión previa al Sprint en donde el Product Owner muestra las actividades contenidas en el Product Backlog, ya priorizadas, el Scrum Team en conjunto con el Scrum Master determinan las actividades que contendrá el siguiente Sprint Backlog.
– Si el Scrum Team acepta la viabilidad de la d de la meta definida previamente, se puede iniciar el Sprint, en caso contrario se comunica para la toma de dediciones (incrementar recursos, reducir el alcance).
– El Scrum Team define la plataforma y el diseño a utilizar
– El Scrum Team puede realizar pregunta a fin de determinar la complejidad de las tareas presentadas.” (Álvarez, 2015).

El Scrum kit es un kit para el juego de Scrum “estimación de esfuerzo para historias de usuario”, este puede ser descargado desde internet y contener un mazo de cartas para Scrum póker, diagrama de Burndown para Sprint, Post-it para tareas del Scrum Taskboard y Fichas para Historias de Usuario.

Programación eXtrema

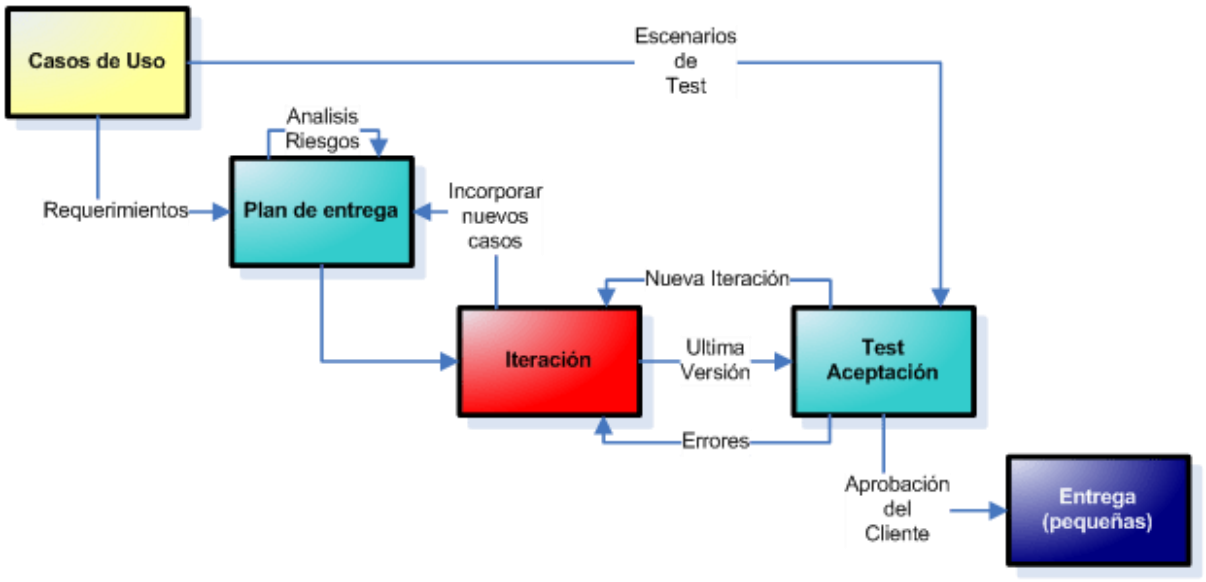

Ilustración No.96: ciclo de desarrollo en XP

En la ilustración No.96 podemos apreciar como la programación extrema afecta el ciclo de desarrollo de software donde algunas ventajas podrían ser los análisis de riesgos, la incorporación de nuevos casos, las nuevas interacciones, y arreglo de errores, como se podría pasar de los casos de uso a test de aceptación y por último pequeñas entregas que puedan ser incrementales.

XP o programación extrema es una metodología que tiene su origen en 1996 (mil novecientos noventa y seis) nacida de Ron Jeffries, Ward Cunningham y Kent Beck (Bahit, 2011), a diferencia del Scrum XP propone un conjunto técnicas prácticas, cuales son aplicadas de manera simultánea y pretenden enfatizar los efectos positivos en un proyecto de desarrollo de software.

Bahit describe: “A diferencia de Scrum, XP propone solo un conjunto de prácticas técnicas, que aplicadas de manera simultánea, pretenden enfatizar los efectos positivos de en un proyecto de desarrollo de Software.” (Bahit, 2011).

Este método se apoya en cinco valores cuales dan presencia colaborativa al equipo como lo son la comunicación ya que todo es trabajado en equipo, desde el análisis y relevamiento hasta la implementación del código de programación, debido a esto todo se debe hablar de cara a cara, procurando hallar soluciones a problemas que puedan surgir.

Otro valor propuesto es la retroalimentación esto debido a que uno de los objetivos en entregarle al cliente lo necesario en el menor tiempo posible, para esto “se demanda la retroalimentación al cliente” ayudando a conocer los requerimientos e implementando los cambios; además de estos dos valores descritos se relata también el del respeto ya que si “se respeta la idoneidad del cliente”.

Este “regresa tal respeto”, se da un intercambio moral que ayudara con el desarrollo del proyecto; se debe de tener coraje ya que “un equipo debe tener el valor para decir la verdad” sobre estimaciones y avances del proyecto, planificando con éxito en vez de dar excusas sobre errores.

Otros de los apartados de XP son las practicas técnicas cuales garantizan un mejor resultado del proyecto, y de fácil comprensión además que se estiman mejor al programar practicas conjuntas: la primera de estas es Client in situ en la cual el cliente se requiere que el cliente esté dispuesto a participar activamente del proyecto, eso hace que se cuente con la disponibilidad suficiente y necesaria.

En el libro Extreme Programing Explained se escribe lo siguiente: “Los objetivos individuales a corto plazo a menudo entran en conflicto con los objetivos sociales a largo plazo. Las sociedades han aprendido a lidiar con este problema desarrollando conjuntos de valores compartidos, respaldados por mitos, rituales, castigos y recompensas. Sin estos valores, los humanos tienden a volver a sus propios mejor interés. Los cuatro valores de XP son:

- Comunicación.
- Sencillez.
- Comentarios.
- Coraje.” (Benck & Andres, 2015).

Para interactuar con el equipo de todas las fases del proyecto, debido a esto la comunicación es esencial ya que así el equipo avanzara más rápidamente debido a que se evacuaran todas las dudas sobre el proyecto y se podrán establecer prioridades a desarrollar, las personas asignadas por el cliente deben ser conocedores expertos de lo que se quiere requerir, se pretende que sean usuarios de software y, personas que cuenten con la información suficiente sobre los objetivos del negocio y la aplicación.

Como segunda practica se da la semana de cuarta horas, se debe asegurar que ningún miembro del equipo pueda realizar un esfuerzo mayor al que pueda disponer, esto marca una diferencia considerable con otras metodologías cuales las estimaciones ineficientes: un equipo descansado logra mayores resultados.

Como practica tres se presenta la metáfora esta se da con el fin de evitar problemas de comunicación entre técnicos y usuarios, buscando un punto de referencia que permita hallar la representación de un concepto técnico con una situación en común con la vida real y cotidiana, para cumplir con esto se debe de realizar paralelismo entre el sistema y la vida real, si este no se entiende inmediatamente se debe de buscar otra metáfora.

Como escribe en un comentario Eugenia Bahit en el libro Scrum-y-eXtrem-Programming-para-programadores: “Una metáfora es la forma de ser didácticos para explicar a nuestro receptor, un concepto técnico y que éste, lo comprenda con facilidad.” (Bahit, 2011).

En ocasiones de debe de hacer uso de más de una metáfora para explicar una función del sistema, pero incluso así puede surgir que no se entienda lo que se debe hacer, momentos para la aplicación de creatividad e ingenio. El diseño simple es una práctica que deriva del famoso principio técnico del desarrollo de software Kiss derivación del “mantenlo simple estúpido”.

Por su propio nombre se trata de mantener el diseño sencillo, de fácil refactorización y comprensión, así como estandarizado, en resumen, es hacer lo mínimo organizadamente. La refactorización es otra técnica de XP la cual consiste en modificar el código fuente de un software sin afectar su aspecto externo, dado que se propone diseñar lo mínimamente indispensable para el añadimiento de funciones de debe de dar refactorización dando mayor cohesión impidiendo redundancias.

La programación a pares es otra técnica, en estos dos programadores se sientan frente a la misma computadora donde cada uno cumple un rol diferente, los roles que se desempeñan en este punto son infinitas dando libertad creativa de desarrollo, entre estas combinaciones se da que uno escriba el código mientras que otro lo revise, el programador más avanzado programa mientras va explicando lo desarrollado al menos experto, dos programadores piensan como resolver el código y uno lo escribe.

Si se busca hacer entregas breves e ir incrementando las pequeñas funcionalidades en cada interacción se aplica la práctica de entregas cortas, esto conlleva a que el cliente pueda tener una mejor experiencia con el software, esto debido a que lo tendrá que probar como nuevo, fácilmente asimilable y nuevo el cual puede sugerir mejoras con mayor facilidad de implementación.

Se puede ampliar la programación por parejas con el párrafo de Bench y Andrés: “El emparejamiento es dinámico. Si dos personas se emparejan por la mañana, por la tarde podrían emparejarse fácilmente con otras personas. Si tiene la responsabilidad de una tarea en un área que no le es familiar, puede pídale a alguien con experiencia reciente que se empareje con usted. Más a menudo, cualquier miembro del equipo actuará como compañero.” (Benck & Andres, 2015).

En esta práctica se pueden encontrar tres tipos de test como lo son el de integración, aceptación y unitario, los unitarios consisten en testear el código de manera unitaria mientras se programa, los de aceptación están más avocados a las pruebas de funcionalidad cuales son definidas por los clientes y levadas a cabo por herramientas que determinan si lo desarrollado cumple con lo requerido, y el integrado integra todos los

test que conforman la aplicación validando el correcto funcionamiento del sistema, esto evita que nuevos desarrollos cambie los anteriores.

Los estándares de programación son esenciales a la hora de programar, da legibilidad y pulcritud al código a la vez que provee a torso programadores un rápido entendimiento y visualización, esta práctica se da para lenguajes que no cuentan con una definición de código como lo puede ser PHP, de allí el equipo se pone de acuerdo para definir y documentar sus propias reglas.

La práctica de propiedad colectiva se da a base de que no existe un dueño de un código o funcionalidad, la propiedad colectiva pretende que todo el equipo conozca cómo está desarrollado el sistema, esto evita sobre esfuerzo intentando entender un código que solo una persona ha programado.

Eugenia escribe que: “Para eXtreme Programming no existe un programador “dueño” de un determinado código o funcionalidad. La propiedad colectiva del código tiene por objetivo que todos los miembros del equipo conozcan “qué” y “cómo” se está desarrollando el sistema, evitando así, lo que sucede en muchas empresas, que existen “programadores dueños de un código” y cuando surge un problema, nadie más que él puede resolverlo, puesto que el resto del equipo, desconoce cómo fue hecho y cuáles han sido sus requerimientos a lo largo del desarrollo.” (Bahit, 2011).

La integración continua de XP propone que todo código encuentre un nuevo punto de alojamiento en el cual deban enviarse los nuevos desarrollos, y correr los test de integración con el fin de que no de errores, estos puntos sueles ser repositorios cuales

pueden ampliarse en la ventaja de la integración mediante el software de control de versiones.

Otro apartado de XP es la programación a pares y código dojo, se describe su nombre haciendo referencia a que dojo es un lugar designado al aprendizaje, sabiduría y meditación, y quien ocupa el ligar de guía es el sensei; el dojo sin embargo se emplea en Japón y el mundo para definir lugares donde se practican artes marciales, dando un sentido semántico de la búsqueda de la perfección, dando el nacimiento del coding dojo donde los programadores se reúnen para la búsqueda de la perfección profesional.

Para un programador el participar en el código dojo es una experiencia vital para su carrera, la finalidad de esta práctica es adquirir nuevas técnicas y aprender de otros programadores, en el dojo se está extenso de competitividad, el cual se sostiene sobre la base de un espíritu de colaboración mutua, concepto fundamental que marca cualquier otro tipo de eventos para programadores. La duración de esta práctica dura entre nueve y cuatro horas.

Otra forma de apreciar esta técnica es descrita en Extreme Programming Explained: “Lo que la mayoría de la gente considera gestión se divide en dos roles en XP: el entrenador y el rastreador (estos pueden o no ser llenados por la misma persona). El coaching se ocupa principalmente de la ejecución técnica (y evolución) del proceso. El entrenador ideal es un buen comunicador, no fácilmente presa del pánico, técnicamente habilidoso (aunque esto no es un requisito absoluto) y confiado. A menudo, como entrenador desea utilizar a la persona que en otros equipos habría sido el líder

programador o arquitecto de sistemas. Sin embargo, el rol del entrenador en XP es muy diferente.” (Benck & Andres, 2015).

Se emplea la modalidad randori y codekata: el ultimo busca perfeccionar una técnica de programación y su significado en japonés es absurdamente similar, entre su consistencia está el presentar una solución ya probada y redefinida a un problema concreto, los pasos son la solución, detalle, prueba y perfección mediante ensayos; el randori busca encontrar una solución a un problema planteado.

Este da un planeamiento apuesto al kata, en este se usa la programación a pares, propuesta por la XP, en este se van turando las parejas de programadores que rotan su rol entre ambos, esta pareja ira analizando y codeando alternativas mientras son explicadas a la audiencia, y esta puede colaborar en la resolución del problema planteado, tras un lapso de tiempo se cambian de lugar el programador y el analista, y esta sede finalmente a una nueva pareja de programadores, repitiéndose la acción hasta que hayan parejas en la sala.

Las principales ventajas de implementar en dojo son mantener activo y motivado al grupo de programadores, reducir la competencia entre miembros del equipo y fomentar la colaboración entre el equipo, poder capacitar a los programadores, la posibilidad rápida, efectiva y certera de encontrar una solución eficiente al conflicto, que la empresa pase a ser algo más que el lugar del trabajo.

El mejor momento para aplicarlo es el último día del mes en una interacción mensual, hacer al finalizar la interacción incrementa las ventajas, se espera que los

programadores se motiven con las nuevas técnicas aprendidas, que posiblemente refuercen el fin de semana.

El TTD

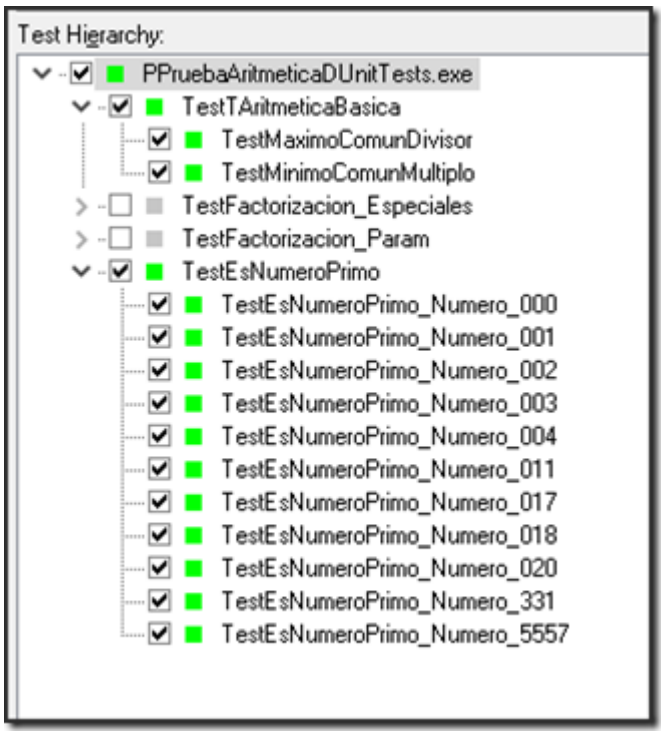

Ilustración No.97: ejemplo real test Unitario

Eugenia Bahit inicia este apartado en su libro de la siguiente manera: “A muchos asusta esta práctica por el simple hecho de ser “desconocida” y resultar su descripción, algo confusa: ¿Qué es acaso, aquello de hacer un test antes de programar?”

Esta es una técnica de programación que consiste en guiar los desarrollos de la aplicación mediante test unitarios, y estos no son más que algoritmos que emulan lo que la aplicación debería de hacer, convirtiéndose así en un modo simple de probar lo que se piensa programar y si esto realmente funciona, una forma de visualizar esto de forma

sencilla seria imaginando el desarrollo de una aplicación, en la ilustración No.29 se puede apreciar un árbol de carpetas de test unitarios.

Los test sirven para saber cuántos, cuáles y como deben de ser los algoritmos a desarrollar para que la aplicación cumpla con las historias de usuario. Entre las ventajas que se encuentran son las siguientes: un test es la mejor documentación técnica que se puede consultar a la hora de misión cumple cada parte del programa, escribir lo que se debe testear antes del código obliga a escribir el mínimo de funciones necesarias.

Evitando el sobre diseño, permite entrar en confianza con programadores que posean menos experiencia, el trabajo en equipo de hace más fácil y une a las personas, además de que la calidad del software aumenta.

Los test unitarios son el alma de la programación que es dirigida por pruebas, estos se encargan de verificar de manera rápida y simple el comportamiento de una parte del código sin alterar el comportamiento y de forma independiente, entre si características principales se encuentra que es: rápido, inocuo ya que este debe ser inofensivo para el sistema, independiente un test de otro test y atómico en referencia a que se prueba una parte mínima del código.

Beck habla de este proceso en la “prueba niña” de la siguiente manera: “¿Cómo se puede ejecutar un caso de prueba que resulta ser demasiado grande? Escriba un caso de prueba más pequeño que represente la parte rota del caso de prueba más grande. Ejecute el caso de prueba más pequeño. Reintroduzca el caso de prueba más grande”. (Beck K. , 2002).

Estos test se realizan en cualquier lenguaje de programación, con formato de frameworks, estos frames tienden a tener una abreviatura del lenguaje de programación, todos usan paradigmas orientado a objetos, tanto en su anatomía como en la implementación, por tanto se agrupan en clases denominadas test case, los métodos dentro de la test case pueden ser bien o no serlo test unitarios, estos deben contener el nombre prefijo Test_ dado en el nombre del método con el fin de que frameworks lo identifique como un test.

Con el XUnit se pueden crear dos métodos especiales que no son test, los cuales están destinos a la preparación del escenario para correr los test de clase y eliminar lo que se considere de liberar, Cuando el test finalice estos métodos serán tearDown() y setUp().

Anatomía dual que denomina el framework y la utilización o implementación, esto divide cada test en tres fases por siglas AAA las cuales representas tres acciones cuales son necesarias para llevar a cabo, estas en su significado en español serian preparar, actuar y afirmar.

Preparar los test consiste en establecer todo aquello necesario para cada uno de los métodos cuales necesiten ejecutarse, si estas preparaciones son comunes se hará uso del método setUp () con el fin de ser creadas, de no ser así se crearán métodos helper, las cuales no pueden ser denominadas test, estas serán acudidas por cada uno de los test cuando les sea de conveniencia.

En el documento Estudio de Test-Driven Development en el proceso de desarrollo de Software se expresa que: “Al escribir las pruebas en primer lugar, el autor del código no se encuentra condicionado por el código a probar, por otra parte permite además especificar el comportamiento esperado sin restringirse a una implementación en particular. También permite a los programadores aprender más de la funcionalidad a implementar, las pruebas pueden tomarse como un criterio de aceptación [4]. Lech Madeyski y Lukasz Szala en su trabajo [14] empíricamente encontraron mejoras además en la productividad medida como una combinación de líneas de código escritas, user stories completadas y pruebas de aceptación correctas al utilizar TDD.” (Vaca, y otros, 2014).

Actuar se refiere a hacer llamadas al código del sistema cubierto por test que desea probar, conocido también como cobertura de código. Para afirmar el resultado de un test se refiere a invocar los métodos del assert, cuales son necesarios para afirmar el resultado obtenido durante la actuación sea el esperado.

Los algoritmos para las TTD consisten en varios pasos como lo son escribir el test y hacer falle, escribir la mínima cantidad de código requerido, escribir u nuevo test y hacer que falle de nuevo, escribir el algoritmo necesario para hacer que pase el test.

Hay varios frameworks XUnit para Unit testing en PHP, sin embargo el único que demuestra contar con gran documentación, estabilidad y cobertura de código es PHPUnit, este pose una gran variedad de métodos assert, y de estas características comunes entre los assert son: el primer parámetro será el valor dado y el segundo el recibió, se le puede pasar como parámetro opcional un mensaje personalizado.

Cada assert recibirá como mínimo un parámetro que será el resultado de ejecutar el código SUT, en generalización existe un opuesto al método assert como lo pueden ser assertNotContains() y assertContains().

PyUnit es el framework elegido oficialmente por Python, la existencia de otros por lo general está hecha para ampliar los beneficios de esta, no necesita ser instalado ya que viene en la librería estándar de Python a través del módulo unitest, una diferencia de este con PHP es que permite efectuar afirmaciones con una sintaxis bastante simple sin necesidad de recurrir a assets específicos.

Otra diferencia es el método assert, entre sus características esta que: el primer parámetro de la función será el valor recibido y el segundo el valor esperado, requieren de dos parámetros obligatorios, se debe de recibir mínimamente un parámetro el cual será el resultado de ejecutar el código SUT y opcionalmente puede recibir un mensaje personalizado para ser arrojado en caso de error.

> Se escribe en el libro Scrum-y-eXtrem-Programming-para-programadores lo siguiente: “discover, “descubrirá” todos los test, identificándolos por el nombre del archivo: debe comenzar por el prefijo “test” (discover utiliza la expresión regular test*.py para identificar Test Cases).” (Bahit, 2011).

Integración continua

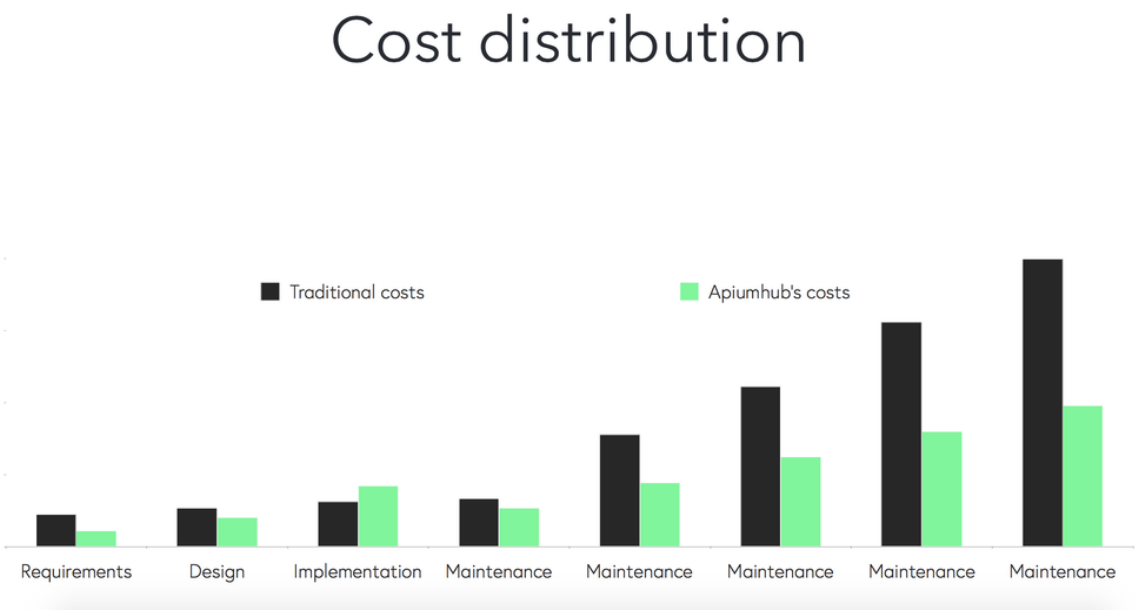

Ilustración No.98: comparación de costos en la metodología y sin la metodología

En la ilustración No.30 se puede apreciar como frente a los costos tradicionales la integración continua es más baratas en los requerimientos, diseño y mantenimiento, sin embargo es más caro en su implementación.

Eugenia Bahit en su libro Scrum y eXtrem Programming para programadores da el siguiente objetivo: “El objetivo de la integración continua, consiste en integrar las nuevas funcionalidades desarrolladas a las existentes, asegurando que lo nuevo, no arruine lo viejo. La respuesta a ¿cómo lograr cumplir con este objetivo? Es la que nos definirá de forma exacta, de qué se trata esta técnica. Veamos cuáles son los pasos a seguir para lograr el objetivo.” (Bahit, 2011).

Dándose entonces el equipo a seguir estas técnicas: llevar un control histórico, unir los repositorios que hayan pasado los test en un repositorio local, correr los test de integración asegurando que todos pasen, generar los test de integración, correr todos los test unitarios asegurándose que todos pasen, y escribir los test unitarios.

Se describe que los pasos uno y dos ya se han descrito lo que resta ahora es seguir con el paso tres el cual está integrado por varas clases de test como lo son el test de sistema en el cual se prueban los casos de uso del sistema, relacionado con la experiencia del usuario con cuestiones de programación y funcionales; test funcionales que prueban el conjunto de criterios evaluados mediante test unitarios, los cuales integran la funcionalidad completa.

Test de aceptación que son test unitarios que aprueban los criterios de aceptación definidos por el dueño del producto; además de test unitarios los cuales ya han sido visto. Los test de aceptación no dejan de ser test unitarios que corren bajo la responsabilidad de un programador, estos test se encargan de verificar la aceptación el cliente se cumple, estos son especificados por el cliente, generando un detalle mayor a cada criterio que es aceptado.

Los test funcionales también creen por cuenta del programador, estos no dejan de ser test unitarios, asemejándose a los test de aceptación ya que estos prueban la funcionalidad completa debido a que los criterios de aceptación corresponden en si a una historia de usuario, el test funcional puede ser exactamente igual a uno de aceptación, estos tienden a probar en mayor medida requisitos no funcionales.

Los test del sistema suelen correr por cuenta de testers, o personas que no necesitan contar con conocimientos de programación, estos tienen a probar la respuesta del usuario frente al software, estos se realizan después de desarrollar la historia del usuario, en estos se puede usar el software manualmente.

Casallas describe las prácticas de esta metodología de esta forma: “1. Utilizar un depósito para el código central para los artefactos del proyecto.
2. Automatizar la construcción del ejecutable (el “build”).
3. Construir el ejecutable y luego hacer que se pruebe automáticamente.
4. Cada integrante hace commit cada día.
5. Cada commit debe producir un nuevo build y un proceso de pruebas.
6. Probar en un clone del ambiente de producción.
7. Hacer fácil la obtención de los últimos entregables.
8. Cada uno puede ver los resultados del último build.
9. Automatizar el despliegue.” (Casallas, 2019).

Usar una herramienta automática es otra alternativa como selenium: el cual permite probar el software una sola vez e ir capturando los resultados en tiempo real y finalmente realizar generar la automatización de código la cual se realizó manualmente, prácticamente se resume en el uso del software manualmente y ver si este responde de la manera como se ha planificado.

Los repositorios son un espacio destinado a almacenar información digital, para el caso estudiado se almacena el código fuente, tarballs, binarios, entre otras cosas del estilo,

sus principales características son: llevar un control histórico de cambios y dar un espacio de almacenamiento centralizado, características que brindan el control de versiones.

Los sistemas de controles de versiones pueden agruparse en dos tipos: centralizados y distribuidos, se dan ventajas de software libres como lo son GIT, Mercurial, Bazzar, estos pueden ser ejecutados en las líneas de comandos, y estos comandos sirven para determinadas funcionalidades.

El bazzar cuenta con una estructura que consta de: repositorio, árbol de trabajo, ramas y revisiones, un repositorio contiene un árbol de trabajo, quien a la vez puede estar integrado por varias ramas, en estas se transitan las versiones del software; este debe instalarse en las maquinas donde se desee usar, habrá repositorios locales y repositorio central.

Entre la sintaxis básica se encuentran las opciones grávelos como lo son –h, -v, y –q que representan ayuda, modo verboso y silenciar de manera respectiva cada uno, help comands muestra la lista completa de comandos, version muestra la versión de la aplicación.

Comando-interno despliega información de ayuda sobre los comandos indicados, se debe de saber quién ha realizado cambios en el proyecto mediante bzr whoami "nombre apellido <nombreapellido@dominio.ext>"; para crear un nuevo proyecto se inicia el repositorio central, luego se crea el branck.

Eugenia complementa en su libro: “Luego, allí mismo, creamos nuestro branch (al que llamaremos trunk): __eugenia_1978_esAR__@mydream:/srv/repos$ bzr init app-curso-xp/trunk Created a repository tree (format: 2a) Using shared repository: /srv/repos/app-curso-xp/” (Bahit, 2011).

La clonación de la información se hace en cada una de las máquinas de cada uno de los miembros del equipo, se puede listar el directorio donde se clono el repositorio, también se puede comprobar si se está conectado al repositorio; se puede dar la creación de archivos de prueba iniciales, y verificar el estado del repositorio, se puede agregar el archivo al repositorio el cual debe ser agregado a bazar con un archivo de prueba.

Todo cambio debe ser informado mediante un mensaje de cambio, se puede enviar este nuevo archivo al repositorio central, el archivo se debe de actualizar en el repositorio central, y cada determinado tiempo todos los miembros traen los cambios desde el repositorio central, se pueden modificar los comandos push y pull modificando el archivo de configuración del repositorio local.

Se pueden presentar diferentes problemáticas que de las que se pueden solucionar por medio de comandos como lo son: ignorar un archivo, recuperar archivo, dejar de lado un archivo temporal, recuperar cambios, encontrar versiones diferentes de un archivo, y merge que no resolvió el conflicto.

Hay comandos que son de uso diario y sirven para muy diversas actividades como lo son: add, check, ci, conflicto-diff, deleted, ignore, ignored, info, log, merge, mv, pull, push, remerge, remove, rename, resolve, revert, shelve, st, tag, tags, incomitive, unshelve,

update, whoami, ver el historia de revisiones y obtener el número de revisiones son otros comandos a los que se puede acceder.

Refactoring

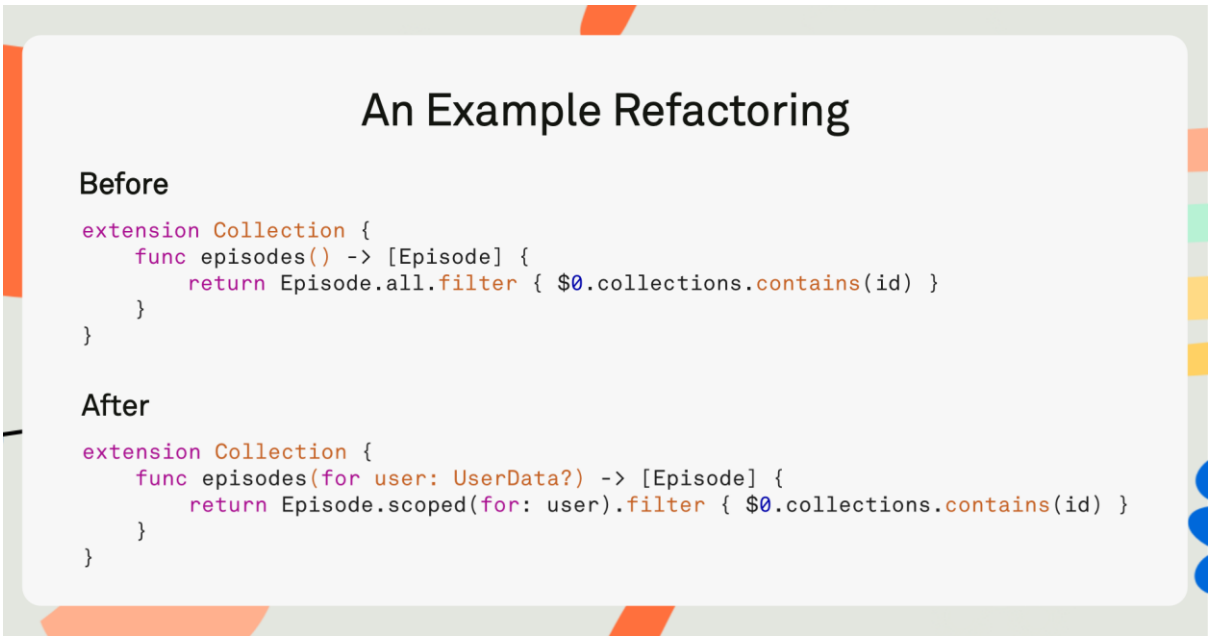

Ilustración No.99: reusó del User Data

Es una técnica que consiste en editar el código fuente sin que dicho código afecte el comportamiento externo del sistema, existen diversos tipos de técnicas de refactorización según lo que necesite el sistema, algunos ejemplos son eliminar el código redundante, estos requieren una técnica diferente a la división de un algoritmo para crear métodos derivados, ejemplo de esto puede apreciarse en la ilustración No.31.

> Eugenia menciona que: “Sin embargo, hablar de técnicas de refactorización puede resultar confuso, ya que la refactorización en sí misma es una técnica, que ofrece diferentes soluciones a cada tipo de problema. Por lo tanto, es preferible pensar la refactorización como una única técnica que propone diferentes soluciones a cada tipo de problema.” (Bahit, 2011).

Se debe de diferenciar error en un código fuente de un problema en el mismo, por lo general el problema está en el lenguaje de programación utilizado, así como en la metodología de orientación, pero si se intentaran abarcar todos los errores de un programa esto se haría muy tedioso, debido a esto solo se abarcan los problemas más comunes; de forma independiente a su lenguaje de programación, pero eso sí, muy estrechamente relacionado con el paradigma de programación orientada a objetos.

La solución debe comenzar por el momento en el que se llevara a cabo. Según TDD no se re factoriza de forma seguida, debe escogerse muy concretamente cuando se debe re factorizar, lo primero que se debe hacer es cumplir el objetivo del código y después re factorizar el código del UST, esto cada vez que se haga una revisión de código, se corrija un bug, se agregue un nuevo método, respetando la regla de los tres strikes se genera la refactorización del programa.

Como se ha hecho anteriormente se ira de lo general a lo particular y de lo particular al detalle; para las variables de uso temporalmente mal implementadas se presentan problemas en casos que defienden una acción concreta para las cuales la solución se presenta en transferir la responsabilidad de la acción a un método.

Méndez expresa en su libro Refactoring de código estructurado que: “Otro punto destacable aportado por el trabajo de [39] es el efecto de los procesos de refactorización sobre la calidad del software. El software posee características que se manifiestan en forma externa y otras que son propias de la estructura interna del mismo, definidas como características internas. En las primeras encontramos conceptos como robustez,

extensibilidad, performance, reusabilidad, etc. Entre las características internas nos encontramos con los conceptos de comprensibilidad, legibilidad, correctitud, redundancia, etc. Los procesos de refactorización pueden afectar a las características internas: al aplicar reducción de código redundante, al aplicar cambios de nombres de métodos o variables. Pero también pueden afectar a factores o características externas que hacen a la calidad del software, por ejemplo, la performance. Si bien se cree que la refactorización de código fuente afecta negativamente en cuanto a la performance [21], existen estudios que demuestran lo contrario. Refactorizaciones tendientes a reemplazar instrucciones if con polimorfismo mejoran la performance de la aplicación gracias a las optimizaciones que hacen los compiladores actuales [39].” (Mendez, 2010).

Las que son requeridas por más de un método son otro ejemplo donde la solución será convertir las variables temporales en propiedades de la clase, las variables de uso temporal que reasignan parámetros donde se puede cambiar el nombre de las variables por un nombre único de las variables.

Los métodos para conseguir parámetros dan distinción entre variables, parámetros y propiedades de la clase, en las cuales las variables de uso temporal definirá un valor concreto solo donde se hubiera definido, las propiedades de la clase son características inertes a las cuales se hará referencia desde diversos ámbitos, y los parámetros serán valores adicionales, cuales no pueden ser considerados propiedades del objeto sin embargo estos son requeridos para que una opción modifique las propiedades del objeto.

En caso de expresiones extensas estas se pueden cambiar por varias del mismo tipo más cortas las cuales permitan una fácil lectura, también se pueden encontrar

métodos extensos los cuales son difíciles de interpretar los mejores agrupar las expresiones en la misma línea, y extraer el código llevándolo a diferentes métodos.

Otro ejemplo son los métodos duplicados en una misma clase la cual se soluciona extrayendo el código duplicado de los métodos, y colocando este en un nuevo método de la clase, el código duplicado en varias clases con la misma herencia es un ejemplo más dado en el libro este se soluciona extrayendo un el código duplicado en las clases hijas.

Con este se crea un nuevo método en la clase madre, como último ejemplo se da el código duplicado en varias clases sin la misma herencia en esta la solución es extraer el código duplicado crear una nueva clase con el código duplicado, crear un nuevo método para esta nueva clase que podría ser heredada por las anteriores o instanciada.

Flower y Beck hablan en el libro Refactoring Improving the Design of Existing Code sobre el refactoring a gran escala, siendo uno de los más destacables el siguiente párrafo: “Las grandes refactorizaciones requieren un grado de acuerdo entre todo el equipo de programación que no es necesario con las refactorizaciones más pequeñas. Las grandes refactorizaciones marcan la dirección de muchos, muchos cambios. Todo el equipo tiene que reconocer que una de las grandes refactorizaciones está "en juego" y hacer sus movimientos en consecuencia. No querrás meterte en la situación de los dos tipos cuyo coche se detiene cerca de la cima de una colina. Salen a empujar, uno en cada extremo del coche. Después de una infructuosa media hora, el tipo que va delante dice: "Nunca pensé que empujar un coche cuesta abajo fuera tan difícil". A lo que el otro responde: "¿Qué quieres decir con 'cuesta abajo'?"” (Flower & Beck, Refactoring Improving the Desing of Existing Code, 2002).

Kanban

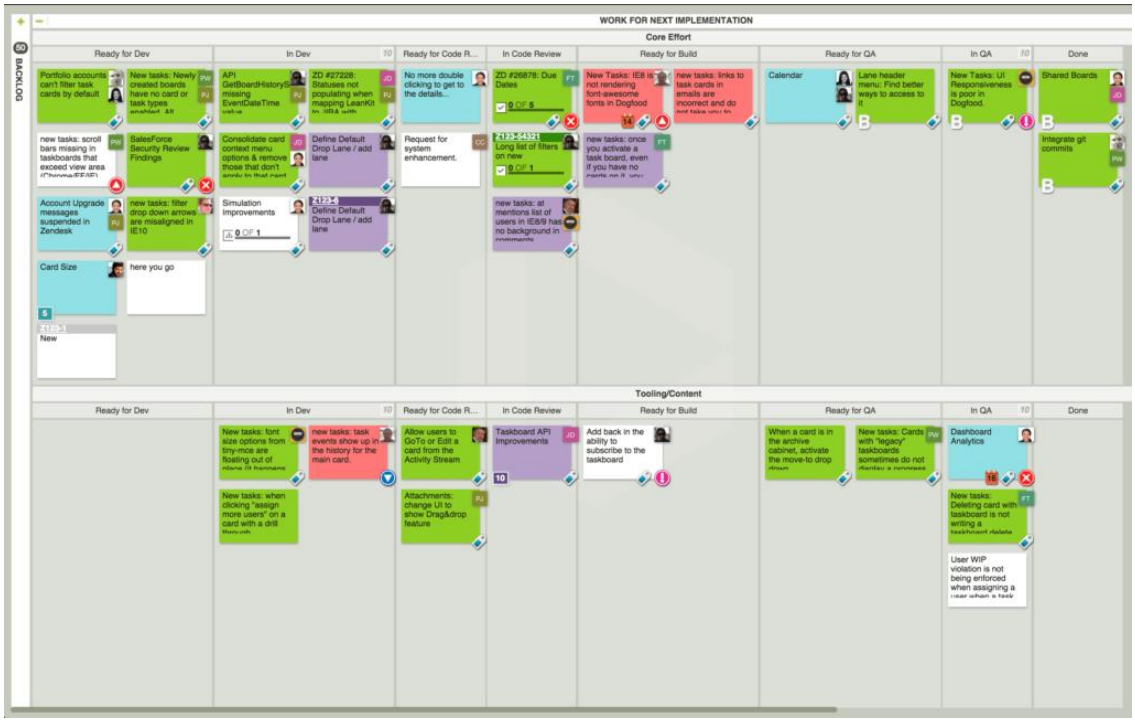

Ilustración No.100: tabla de un ejemplo real mediante Kanban

En la ilistracion No.32 se puede apreciar un ejemplo real mediante Kanban en el cual se puede apreciar como en las columnas de ciclo esta: el listo para desarrollar, en desarrollo, listo para la revisión de código, en revisión del código, listo para compilarse, listo para asegurar calidad, teniendo calidad y hecho.

Eugenia Bahit en su libro Scrum y eXtrem Programming para programadores dice lo que Kanban significa: “Kanban es un término japonés el cual puede traducirse como “insignia visual” (Kan: visual, ban: sello o insignia).” (Bahit, 2011).

Debido a su simplicidad y facilidad para camuflarse con las metodologías más tradicionales es estadísticamente la que menos resistencia al cambio ofrece de pasar de una metodología predictiva a una ágil, si significado en japonés es insignia visual, esta es

la más nueva con una década de diferencia, nació en Toyota a manos de Taiichi Ohno, y se implementó en el medio del software en el 2004 por David Anderson en la unidad de negocio de XIT de Microsoft el cual dio resultados alentadores.

Esta metodología se apoya en la producción de demanda, en el cual solo se produce lo necesario haciendo que el ritmo de demanda sea quien controle el ritmo de producción, en el sistema hay un disparador el cual es representado por tarjetas, las cuales se dan de manera limitada; cada ítem de trabajo se acompaña de estas tarjetas, por tanto este nuevo ítem podrá iniciarse si dispone de una tarjeta, dado esto si no hay nuevas tarjetas no se pueden iniciar nuevos trabajos; cabe aclarar que al terminar el trabajo se libera la tarjeta.

En la metodología hay tres reglas tales son mostrar el proceso, limitar el trabajo en curso y optimizar el flujo de trabajo. Mostrar el proceso es una regla que busca hacer visible los ítems del trabajo permitiendo conocer de manera explícita el proceso de trabajo actual, así como los impedimentos que surjan, estas medidas se visualizan a través de un tablero físico, este es fundamental para comprender la capacidad de desarrollo del equipo.

Kingberg y Skarin escriben en Kanban y Scrum Obteniendo lo Mejor de Ambos los siguiente sobre la limitación en Kanban: “En el ejemplo anterior de Kanban, como mucho puede haber 2 elementos en el estado de flujo de trabajo "en curso" en un momento dado, independientemente de las longitudes de cadencia. Tienes que elegir qué límite aplicar a qué estados del flujo de trabajo. Pero la idea general es limitar el WIP de todos los estados del flujo de trabajo, empezando por "lo más pronto posible" y

terminando "lo más tarde posible" a lo largo de la cadena de valor. Así, en el ejemplo anterior deberíamos considerar añadir un límite al WIP también en el estado "pendiente" (o la que quiera que sea tu cola de entrada). Una vez que tenemos los límites de WIP en su lugar, podemos empezar a medir y predecir el tiempo de entrega, es decir, el tiempo medio de un elemento para realizar todo el camino a través de la pizarra. Tener tiempos de entrega predecibles nos permite comprometer los SLA (acuerdos de nivel de servicio, en inglés service-level agreements) y hacer planes de entrega realistas." (Kingberg & Skarin, 2010).

Para cumplir con esta regla será necesario definir con precisión el punto de inicio y finalidad de visibilidad del proceso, los tipos de ítems de trabajo, si son bugs, refactoring o actualizaciones, y el diseño de las tarjetas que acompañaran a cada ítem, ya que la información que se de en cada tarjeta debe ser lo suficientemente precisa, lo cual permite su evaluación y toma decisiones de a cualquier involucrado en proyecto, por lo general esta cantidad mínima suele ser el título, ID, Fecha de entrada, tipo de ítem, persona asignada, prioridad y deadline.

En el Kanban el flujo de proceso de trabajo ser verá reflejado de izquierda a derecha mediante columnas que representen cada etapa y estas a su vez se dividen en dos: curso y terminado, en medio de cada columna representativa se colocan los ítems en espera.

En kanban la limitación de trabajo en curso está dado por el WIP: work in progress este consiste en definir la cantidad de ítems simultáneos que pueden ejecutarse

en el mismo proceso, cada uno de estos procesos puede tener diferentes límites de WIP, estos límites permiten detectar los cuellos de botella rápidamente.

Esto se da solo con observar en las columnas bloqueadas, el proceso anterior y posterior inmediatos, así se podrá decidir donde se encuentran los puntos de conflicto, y trabajar en ellos para remediarlos.

Hay muchos métodos para trabajar los cuellos de botellas, pero la recomendada es colocar en la cola del proceso un buffer previo al lugar en el que genera el cuello de botella. El flujo de trabajo es la progresión visible de los ítems de los sucesivos procesos en un sistema de Kanban.

El objetivo de optimizar dichos flujos es alcanzar un proceso de desarrollo estable, acorde a las necesidades del proyecto y previsible el cual distingue un desarrollo de trabajo parejo, si un ítem se encuentra estancado el equipo debería colaborar con el fin de lograr recuperar el flujo de trabajo y tomar medidas para prevenir futuros estancamientos.

Eugenia cirra su libro con el siguiente párrafo: “Si bien no existen reglas preestablecidas para optimizar el flujo de trabajo en Kanban, las principales actividades sobre las cuales suele enfocarse esta optimización, en la práctica, son:

• El trabajo sobre cuellos de botella

• Análisis sobre las colas de entrada (buffer de ítems de trabajo)

• Mejoras que impliquen modificaciones en el proceso de creación de valor.” (Bahit, 2011).

Frecuencia de uso del UML y metodologías ágiles

Una de las constantes en los diagramas del UML es que el veinte por ciento de los diagramas resuelven el ochenta por ciento de los problemas, incluso en los libros oficiales del Lenguaje de programación unificado se identifican el uso de frecuencia de estos.

Los casos de uso son los más usados dando frases como “no me imaginaria una situación en la que usara un caso de uso”, estos son definidos como una herramienta esencial para los requerimientos, la planificación o el control de proyectos interactivo, esta es una tarea principal durante las fases de elaboración, sin embargo, se irán descubriendo otros a medida que avance.

El caso de uso es un requerimiento potencial y hasta que no se capture no se podrá palear como manejar el proyecto, algunos prefieren listar y analizar los casos de uso primero, sin embargo, los usuarios pueden ayudar a identificar los casos de uso del sistema, se pueden probar las dos maneras y escoger la que mejor sirva.

Los diagramas de clase son la columna vertebral de casi todos los métodos orientados a objetos por consiguiente es ampliamente utilizado en diversas disciplinas de la tecnología para la abstracción del sistema que se planea, se recomienda ajustar las perspectivas desde las cuales se dibuja los modelos a la etapa del proyecto, si está en la etapa de análisis dibuje modelos conceptuales.

Si se trabaja con software céntrese en los modelos de especificación, dibuje los modelos de implementación solo cuando se ilustre una técnica de implementación en

particular, centrase en áreas claves, es mejor mantener pocos modelos activos y actualizados.

Lo diagramas de interacción se utilizan cuando se desee ver el comportamiento de los objetos detrás de un caso de uso, son buenos mostrando esta colaboración entre objetos, pero no para definir exactamente su comportamiento, sin embargo, si lo que se quiere es ver el comportamiento de un objeto mediante varios casos de uso se usan los diagramas de estado, pero para el comportamiento de un objeto en muchos casos de uso u procesos se usa el diagrama de actividades.

Los diagramas de paquetes se usan en caso de sistemas grandes, este se usa cuando el diagrama de clases es más grande que ya no cabe en un tamaño carta, se deberá querer mantener las dependencias al mínimo ya que esto reduce el acoplamiento, estos son especialmente útiles para pruebas, sin embargo, se prefiere hacer las pruebas clase por clase, aunque se pueden hacer pruebas unitarias paquete por paquete. Los diagramas de emplazamientos se usan en su mayoría para el trato de bocetos informales, es uno de los diagramas con mayor nivel de alcance del sistema.

El diagrama de objetos es una técnica que está entre los casos de uso y los diagramas de clase su frecuencia de usa debe no ser tan extensa ya que muchos otros diagramas permiten la interacción de los objetos dentro del sistema.

Los diagramas de colaboración pueden ser definidos como el segundo paso en la planificación de sistema informático estando entre el caso de uso y el diagrama de actividades, o secuencia.

Los diagramas de secuencia son esenciales en la planificación ya que permiten ver a detalle muchas actividades entre los objetos en un mismo diagrama, además las actividades entre objetos e interfaces, interfaces y personas, personas y personas, personas y datos.

El diagrama global de interacciones permite el uso de diagrama de actividades, así como de secuencia, en este se permite modularidad si el sistema es muy grande, esta metodología no ha sido tan utilizada teniendo en cuenta la poca información sobre este diagrama en la internet siendo pues más utilizado el diagrama de actividades y el diagrama de secuencia.

El diagrama de interacción es un diagrama que permite ver a detalle las funciones y detalles dentro de una clase, mostrando además entradas y salidas de conexiones entre clases por medio de las funciones, se determina que su uso no es tan alto ya que los programas más complejos requieren de varias variables y funcionalidades que pueden hacer una proyección de clases compleja, y muy difícil de realizar.

Los diagramas de componentes dan una vista física del sistema, la cual es a alto nivel y muestra las relaciones entre el sistema, este se compone de una parte modular, esta parte sirve para ver el sistema, el componente es autónomo y encapsulado, se puede ver como si fuera un sub sistema, este sistema puede ser visto como un diagrama de clases poco avanzado haciendo que, aunque sea útil para ver partes iniciales de un sistema.

Es menos útil que el diseño de clases ya que se gasta más energía en el diseño de componentes que en el de clases, dando el de clases una mejor forma de evaluar lo que sucede a nivel de programación, siendo entonces los diagramas de componentes menos prácticos que los de clases a la hora de ser implementados en la planeación de un sistema.

En el apartado de diagramas se mostraron otros modelos, pero son de muy altos requisitos haciendo que para la búsqueda de frecuencia de uso se vean en la práctica muy pocos casos en los que se hacen uso siendo así descartados como posibles candidatos para la solución de la monografía; como lo puede ser el diagrama de despliegue.

En un hecho que la programación orientada a objetos es una de las metodologías de programación más utilizadas hoy en día tanto en su versatilidad, como en permitir el uso de variables y funciones dentro de sus parámetros, haciendo que más que simplificar el trabajo de anteriores metodologías las amplifique y potencie.

Como metodología más utilizada se encuentra el XP; aunque otros documentos sugieren que la más usada es Scrum; discutiendo siempre el primer puesto y dejando a kanban en el último puesto, para el primer caso significa que en las metodologías agiles de desarrollo de software se busca antes que alguna otra cosa llegar al código fuente lo más rápidamente posible, por encima de una filosofía de desarrollo.

Scrum además de buscar llegar rápidamente al código fuente busca saber si los requisitos implementados fueron los requeridos por el cliente. Al estudiarse los casos de éxito en la implementación del UML se encontró con que los diagramas más utilizados fueron los diagramas de clases.

Los diagramas de estado y los diagramas de actividades lo cual no está lejos del título, o la búsqueda de información sobre los diagramas esenciales a la hora de desarrollar la mayoría del software orientada a objetos.

Diagramas esenciales para el desarrollo de sistemas informáticos

El primer diagrama que termino siendo elegido son los casos de uso, debido a diversos factores comenzando por ser el más apartado de la programación orientada, dando a entender la comunicación que hará el sistema “con bolas y palos”, haciéndolo sumamente fácil de entender, sin embargo su importancia no se ve reflejada del todo en los libros anteriores al UML Lenguaje de modelado Unificado o el UM Modelo Unificado.

Ya que su existencia se relata en el libro de Ivar Jacobson de mil novecientos noventa y cuatro, y mil novecientos noventa y cinco, siendo antecesores en el mismo UML el diseño guiado por responsabilidades en mil novecientos ochenta y nueve, los libros de Sally Shalaer y Steve Mellor de mil novecientos ochenta y nueve a mil novecientos noventa y uno, los métodos orientados a prototipos de Coad y Yourdon.

Otra de las razones por las cuales se escogió es que es capaz de adaptarse a los requerimientos del usuario con el sistema prácticamente en cualquier ámbito, en esta monografía se demuestra su inclusión en Integración de UML con refinamiento de servicio para modelado y análisis de requerimientos de Yilong Yang, Wei Ke, King Yang y Xiaoshan Li el cual es usado para definir en qué momentos se está definiendo de un servicio público y uno privado.

En el libro de UML gota a gota los autores especifican que no se imaginan un escenario en el que no se use un caso de uso para acoplar el requisito de un sistema con el usuario, es tan esencial que hasta cierto punto es imposible desarrollar el sistema si no se

pasa por el caso de uso, los requisitos pueden ser detectados por el o los desarrollares, sin embargo también pueden ser desarrollados por el cliente mediante interacciones para la captura de requisitos.

El segundo diagrama consensuado para estar en este apartado son los diagramas de actividad los cuales permiten definir el comportamiento de un objeto a través de muchas actividades o procesos, sin embargo, este diagrama no se usa para ver la relación de objetos entre ellos. Este diagrama existe desde la versión 1 del UML, se inventó para este lenguaje, por tanto, no existían con anterioridad.

Este diagrama se aplica en Generación automática de código a partir de diagramas de gráficos de estado UML de Sunitha E. V. y Philip Samuel así como en Generación de especificaciones de Mude a partir de diagramas de descripción general de interacción UML: un enfoque basado en la transformación de gráficos de Chafika Djaoui, Khaled Khalfaoui, Elhillali Kerkouche y Allaoua Chaoui, en estos casos de éxito se puede apreciar que el diagrama de actividades se usa de manera muy conjunta con otro diagramas como el de estado y de interacciones globales.

El tercer diagrama del título es el de estado este sirve para ver el estado de un objeto a través de varios casos de uso. Este diagrama se presenta en Generación automática de código a partir de diagramas de gráficos de estado UML de Sunitha E. V. y Philip Samuel así como en una Propuesta de Algoritmo Spin / Promela para el Análisis y Diagnóstico de Errores en Diagramas de Secuencia UML de Cristian L. Vidal-Silva, Rodolfo H. Villarroel, Xaviera A. López-Cortés y José M. Rubio.

En este se usan los diagramas para dar con los estados fundamentales del sistema, en la generación de combinación para cerradura, para la representación del despertador, el horno micro hondas; en el otro artículo descrito se menciona su uso, en este no se muestran los diagramas impresos pero se menciona que es de ayuda para la realización de diagramas de secuencia, se deduce que usar los diagramas de secuencia para sacar los diagramas de estado es viable pero sacar los diagramas secuencia a partir de los de estado es poco viable.

Al igual que el diagrama de actividades el diagrama de estado se hizo exclusivamente para el UML en la versión 1.0 de lo modelo en el año mil novecientos noventa y siete. El ultimo diagrama que se agrega es el de clases, este es el diagrama más competido antes de que pareciera el UML pues todos los integrantes tendiendo definido un método para este diagrama un ejemplo está en el mismo nombre de los libros guías para el desarrollo de sistemas en donde se le daban nombres como clase y objeto en coad, objeto por Shaler/mellor, y Jacobson, tipo de objeto de Odell aunque un sinónimo podrá ser “clase de objeto”.

Mientras Rumbaugh usa el termino generalización, Jacobson y booch usan heredar, Shlaer/mellor y odell el nombre sub tipo, mientras que coad es el único de usar el método Espec-hen; sin embargo para agregación todo tienen un nombre distinto como “contiene” por booch, “parte todo” de coad, “consiste en” Jacobson, composición “odell”, siendo únicamente compartido el nombre de composición por Agregación.

De igual forma la composición uno a uno, uno a varios o ninguno, o uno a ninguno aparece de igual forma en todos los métodos anteriores al UML. El diagrama de

clases es esencial a la hora de representar sistemas informáticos ya que no solo muestra una relación posible entere los objetos que contiene una clase.

Sino que además es la fase más avanzada en la planeación de un sistema de información, además la mayoría de programas hoy en día usan la programación orientada a objetos la cual permite un alto acoplamiento de este diagrama en el desarrollo de la solución.

Este diagrama de implemento en la Generación automática de código a partir de diagramas de gráficos de estado UML de Sunitha E. V. y Philip Samuel, Cristian L. Vidal-Silva, Rodolfo H. Villarroel, Xaviera A. López-Cortés y José M. Rubio lo utilizaron para la realización del proyecto y articulo Una Propuesta de Algoritmo Spin / Promela para el Análisis y Diagnóstico de Errores en Diagramas de Secuencia UML.

Conclusiones

A partir de todos los diagramas descritos y cualquier sea la metodología ágil que se pueda usar en el desarrollo de un sistema informático, se ha encontrado que los diagramas críticos que se usan a la hora de desarrollarlo en casi la totalidad de cualquier desarrollo, son el diagrama de actividades, casos de uso, clases y en menor medida de estado.

Se pudo describir el lenguaje de modelado unificado como un conjunto de diagramas que sirven para múltiples propósitos a lo largo de todo el desarrollo del sistema, hasta llegar incluso a usarse como parte de la implementación en casos de actualización automática mediante diagramas.

El UML tiene una historia relativamente corta, así como reciente, comenzando con el UM, donde se puede evidenciar los diagramas de clase, su versión 1.3 donde ya se encuentra el diagrama de casos de uso, y para esta misma versión se encuentran también los diagramas de actividades y estado, los cuales no son añadidos de otros métodos anteriores sino hechos para el mismo UML.

De los diagramas de componentes el cual trata todos los componentes del sistema, objetos en el cual se muestra la interacción de los objetos en el sistema, estructura compuesta en el cual se tratan los algoritmos dentro del sistema, despliegue el cual es muy similar al diagrama de componentes, paquetes en el cual se muestra las carpetas dentro del sistema y que contienen.

Casos de uso los cuales se muestra las interacciones del usuario con el sistema, estado en el cual se muestran los cambios de estado de un objeto, secuencia en el cual se muestra la interacción mediante un caso de uso, tiempo que ayuda a plantear y resolver problemas financieros debido a su utilidad para medir los desplazamientos simbólicos de la capital, global de interacción da a mostrar el modelado de los aspectos dinámicos, y clases los que permiten tener una mejor noción de cuáles serán las clase programadas, sus atributos y operaciones.

De todos estos los cuales al numerarse dan como resultado trece se esclarece que solo tres son totalmente necesarios y uno sirve de soporte en las interacciones del sistema.

Las ventajas del UML son numerosas como lo pueden ser una mejor planificación, diseño y arquitectura de un sistema y algunas desventajas como el conocido “veinte ochenta”. En el caso de éxito de Generación automática de código a partir de diagramas de gráficos de estado UML se evidencia que su diagrama central es el de estado, Una Propuesta de Algoritmo Spin / Promela para el Análisis y Diagnóstico de Errores en Diagramas de Secuencia UML.

Se muestra que su diagrama central es el de secuencia, Generación de especificaciones de Mude a partir de diagramas de descripción general de interacción UML: un enfoque basado en la transformación de gráficos se evidencia que los diagramas principales son los dirigidos a interacción.

En Integración de UML con refinamiento de servicio para modelado y análisis de requerimientos se evidencia un principal desarrollo en los casos de uso, La solución

propuesta en Depurador: un depurador de nivel de modelo para UML-RT muestra el uso de todo tipo de diagrama para la perduración de código a través de los diagramas de UML.

Sobre las metodologías agiles se puede concluir que el Scrum cuenta con tres papeles fundamentales: el Scrum master, el equipo y el dueño del producto; el dueño del producto es quien realiza el BackLog, después se realiza el spirng planning meeting, en siguiente se realiza el Spring BackLog.

Hay varias reuniones que se consideran necesarias en este método de trabajo como lo son el Daily Scrum y el Sprint Review. En XP se concluye que hay tres fases de desarrollo: la retroalimentación a escala fina, el proceso continuo en lugar de por lotes, y el entendimiento compartido, en la primera fase se establece de cuánto tiempo serán las pruebas, se planifica el sistema, en todo momento debe estar el cliente en el sitio, y la programación debe de ser en parejas dando un des atasco en la producción del código y el análisis del mismo.

En la segunda fase al darse una integración continuada dando un incremento de funciones del sistema en menor tiempo bajo las mismas pautas de la metodología. En la refactorización se arregla el código para que sea más legible y “muestra buenas prácticas de programación”, al hacerse pequeñas entregas es fácil medir el tiempo de trabajo que se utilizó para desarrollar determinada funcionalidad.

Estas no pueden pasar las tres semanas; en la última fase se usa el diseño simple lo que permite una rápida ejecución y mantener la estructura del diagrama actualizada, la

metáfora define como funciona el sistema por completo, dando a entender su código y alcance, la propiedad colectiva del código permite que todos los miembros del equipo conozcan gran medida del código del proyecto, que se hizo y como se hizo.

El desarrollo guiado por pruebas se enfoca en minimizar el número de bugs, implementar las necesidades justas que se necesitan evitando la agregación de más pruebas y hacer el software de maneras más modular, y re utilizable. La integración continua hace énfasis en las pruebas unitarias las cuales se llevan a cabo cada tiempo establecido, esto hace que los errores se detecten en etapas tempranas de desarrollo, en esta forma se realizan controles de versiones efectivas haciendo que todos los desarrolladores trabajen con la misma versión del sistema.

El refactoring permite mediante la lectura de código detectar posibles errores en la programación que pueden ocurrir sin que haya un error sintáctico dando arreglo en etapas tempranas, así dando un programa mejor desarrollado y libre de errores, la falta de encapsulación y, la repetición de atributos y operaciones en diversas clases son errores que se pueden arreglar utilizando esta metodología, además esta técnica puede ser usada tanto en toda practica de programación.

La última metodología descrita es el Kanban el cual es un sistema de tarjetas en el que es fundamental que solo una persona se encargue de un trabajo, al igual que otras metodologías tiene una tabla en la que se pueden ver el flujo de los proyectos, y así solucionar fácilmente los cuellos de botella.

En reflexión se da a conocer una de los grandes defectos de las metodologías agiles, es el hecho de que parecen no ser tan útiles cuando se propone a desarrollar un equipo multidisciplinar, ya que la proposición de tiempos no siempre se acomoda a todos los tipos de tarea dentro de estos proyectos. Entre las comparaciones que se pueden hacer entre metodologías se da que Kanban y Scrum son la propuesta de cumplimiento efectivo de objetivos en cortos periodos de tiempo y mejora continua dentro del proyecto, y la empresa, los dos son visuales permitiendo la trasparencia del trabajo de todo el equipo, permiten la detección de errores, fomentan el trabajo en equipo y la organización.

Entre las diferencias se da que en Kanban no se necesitan roles concretos, Scrum requiere de reuniones de tiempo fijo, pero en camban estas reuniones no son necesarias, Scrum limpia el tablero en cada interacción sin embargo Kanban no lo hace y mantiene ese tablero fijo.

Ambos ayudan a eliminar cuellos de botella, de igual forma se requiere un compromiso por parte de sus participantes para que las metodologías funcionen. Ha de considerarse que el uso de Scrum es una metodología de un solo uso en los proyectos mientras que Kanban son de un uso continuado o de bucle, XP por su parte viene genial para la depuración de código o los proyectos hechos sin personal especialmente capacitado como lo pueden ser un equipo compuesto solo de programadores sin ninguna otra rama de conocimiento dentro del tema de la informática.

Al aprender a usar el diagrama de actividades a fondo el diagrama de estado queda relegado a un segundo plano ya que se pueden identificar actores, así como diferentes estados de una clase. Entre la clase de proyectos que usara la totalidad de estos diagramas

estaría el diagrama de clases ya que la mayoría de lenguajes para el desarrollo de páginas web funciona mediante funciones, haciendo entonces que se estructure una clase.

El diagrama de estado y el de actividades van de la mano puesto que con la solución de un diagrama de actividades se puede encontrar solución al diagrama de estado.

El dilema del veinte ochenta es mucho más real de lo que se puede esperar debido a que si se saca el porcentaje de trece sobre tres entonces el porcentaje seria del veintitrés, punto cero ocho. Entre las metodologías en las que mejor se podrá implementar el UML estarían de escala mayor a menor: Scrum, Kanban y por ultimo XP. El diagrama de secuencias permite una lectura más rápida del UML sin embargo debería de hacerse el diagrama de estado ya que no hay manera de permitir que en una secuencia se den estados de un objeto.

Para el diseño de interfaces se puede usar el diagrama de clases con un “poco de trampa” haciendo que se use solo las operaciones de dicha clase “frame”. De todos los diagramas conocer los más críticos pueden ayudar a mejorar las metodologías agiles dando un producto rápidamente, lo cual hace parte de los objetos tanto del UML como de las metodologías agiles.

Notas

- Desarrollar una página web a partir de una metodología ágil implementando los diagramas de caso de uso, de actividades y de clase.
- Desarrollar un sistema de transporte a partir de una metodología ágil implementando el caso de uso, el diagrama de actividades y de clase.
- Desarrollar un videojuego a partir de una metodología ágil implementando solo el caso de uso, el diagrama de actividades y de clase.
- Desarrollar una página web implementando los diagramas de caso de uso, secuencia, estado y clases.
- Desarrollar un sistema de transporte implementando los diagramas de caso de uso, secuencia, estado, y clases.
- Desarrollar un videojuego con los diagramas de caso de uso, secuencia, estado y clases.

Bibliografía

Álvarez, I. (2015). Desarrollo Ágil con SCRUM. Bogotá, Colombia. Pontificia Universidad Javeriana.

Amavizca, L. García, C. Jiménez, E. Duarte, G. Vázquez, J. (2014). Aplicación de la Metodología Semi-Ágil ICONIX para el Desarrollo de Software: Implementación y Publicación de un Sitio WEB para una Empresa SPIN - OFF en el Sur de Sonora. Guayaquil, Ecuador. 12th Latin American and Caribbean Conference for Engineering and Technology.

Arango, F. Gómez, M. Zapata, C. (2005). Transformación del modelo de clases UML a Oracle bajo la directiva MDA: un caso de estudio. Medellín, Colombia. Universidad Nacional de Colombia.

Arquitectura de Software: Puntos de vista. En *Uniandes*. Recuperado el 17 de Julio de 2020 de: https://profesores.virtual.uniandes.edu.co/~isis3702/dokuwiki/lib/exe/fetch.php?media=principal:isis3702-puntosdevista.pdf.

Avilés, E. (2009). Curso UML. Valladolid, España: Universidad de Valladolid.

Bagherzadeh, M. Seekatz, D. Hili, N. Dingel, J. (2018). MDebugger: A Model-Level Debugger for UML-RT. Gotemburgo, Suiza. IEEE.

Bahit, E. (2011). Scrum y eXtreme Programming para Programadores. Buenos aires, Argentina. Creative Commons Atribución-NoComercialSinDerivadas 3.0 Unported.

Bahit, E. (2019). Introducción al desarrollo dirigido por pruebas. Londres, Reino Unido. Hackers & Developers Magazine.

Barraza, F. (2019). Arquitectura de Software, perfiles UML. Bogotá, Colombia. Pontificia Universidad Javeriana.

Beck, K. (2002). Test-Driven Development. Hersham, Inglaterra: Three Rivers Institute.

Bench, K. Andres, C. (2015). Extreme Programing Explained, Embrance Change. Estados Unidos de América. Pearson CMG.

Berzal, F. (2005). El lenguaje unificado de modelado. Granada, España. Universidad de Granada.

Bonaparte, U. (2012). Proyectos UML Diagramas de Clases y Aplicaciones JAVA en NetBeans 6.9.1. Buenos Aires, Argentina. Universidad Tecnológica Nacional.

Bustos, G. (2019). Integración de modelos en UML 2. Valparaíso, Chile. Pontificia Universidad Católica de Valparaíso.

Cabot, J. (18 de 02 de 2019). Las Mejores Herramientas Uml – Edición 2019. Obtenido de Ingenieriadesoftware: https://ingenieriadesoftware.es/herramientas-uml/

Casallas, R. (2019). Integración Continua. Bogotá, Colombia. Universidad de los Andes.

Conde, J. (2019). Diagramas UML. Puebla, México. Benemérita Universidad Autónoma de Puebla.

Cruz, V. Gutiérrez, E. Mendivil, L. (2019). Diagrama de componentes. La Paz, Bolivia. Universidad Salesiana de Bolivia.

Diagrama de Estructura Compuesta UML 2 - DS-UFT. En *NanoPDF*. Recuperado el 17 de Julio de 2020 de: https://nanopdf.com/download/diagrama-de-estructura-compuesta-uml-2-ds-uft_pdf.

Diagramas de secuencia: Interacciones Básicas. En *Uniandes*. Recuperado el 17 de julio de 2020 de: https://profesores.virtual.uniandes.edu.co/~isis2603/dokuwiki/lib/exe/fetch.php?media=principal:1._dsec-conceptosbasicosconejemplos.pdf.

Díaz, M. Collazo, A. (2013). La Habana, Cuba. Universidad de las Ciencias Informática.

Djaoui, Ch. Khalfaoui, K. Kerkouche, E. Chaoui, A. (2018). Generating Maude Specifications from UML Interaction Overview Diagrams: A Graph Transformation Based Approach. Jijel, Argelia. University Mohamed Seddik Benyahia.

Drake, J. (2009). Diagramas de actividad y diagramas de estados. Cantabria, España. Universidad de Cantabria.

Electrónica, Automática e Informática Industrial. (2019). UML: El modelo dinámico y de implementación. España: Universidad Politécnica de Madrid.

Escalona, M. Gutiérrez, J. (2007). Diagramas UML de Actividades para la Definición de Reglas de Negocio y Comportamientos de RFs. Sevilla, España. Universidad de Sevilla.

Fontela, C. (2019). Diseño MVC. Buenos Aires, Argentina. Universidad de Buenos Aires.

Fowler, M. Scott, K. (l999). UML gota a gota. México D.F., México. Pearson Educación.

Fowler, M. y Beck, K. (2002). Refactoring: Improving the Design of Existing Code. Taipei, Taiwán. National Taiwan University.

Galicia, Y. (2019). Paradigma Orientado a Objetos UML: Lenguaje de Modelo Unificado. Puebla, México. Benemérita Universidad Autónoma de Puebla.

García, F. Conde, M. Bravo, S. (2019). Tema 2: Modelo objeto Una descripción de UML. Salamanca, España. Universidad de Salamanca.

García, F. Moreno, M. García, A. (2018). Ingeniería de Software I. Tema 8 UML-Unified Modeling Languaje. Salamanca, España. Universidad de Salamanca.

García, F. Moreno, M. García, A. (2018). UML. Unified Modeling Language. Salamanca, España: Universidad de Salamanca, Departamento de Informática y Automática.

García, F. Pardo, C. (2019). Diagramas de Clase en UML 1.1. Burgos, España. Universidad de Burgos.

Gloria, G. Acevedo, J. Moreno, D. (2011). Una ontología para la representación de conceptos de diseño de software. Medellín, Colombia. Revista Avances en Sistemas e Informática.

Grau, X. Sánchez, M. (2013). Desarrollo Orientado Objetos con UML. Madrid, España: Universidad Politécnica de Madrid.

Gutiérrez, D. (2009). UML Diagramas de Paquetes. Mérida, Venezuela. Universidad de los Andes.

Gutiérrez, D. (2011). UML Diagramas de Clases. Mérida, Venezuela. Universidad de los Andes.

Gutiérrez, D. (2011). UML: Diagramas de Estados, Diagrama de Actividades. Mérida, Venezuela. Universidad de los Andes.

Gutiérrez, J. (2009). Diagramas UML de Casos de Uso y de Requisitos. Sevilla, España. Universidad de Sevilla.

Hernández, P. (2017). Modelo Tecnológico para la Implementación de un Sistema Micro Empresarial para la Gestión de Facturación, Inventario, Compras y Contabilidad. Pereira, Colombia. Universidad Tecnológica de Pereira.

Ionos. (23 de 08 de 2019). 6 herramientas UML para cualquier ocasión. Obtenido de Ionos: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/las-mejores-herramientas-uml/

Kecher, C. (2006). UML 2.0 - Das umfassende Handbuch. Boon, Alemania. Galileo Computing.

Kniberg, H. Skarin, M. (2010). Kanban y Scrum – Obteniendo lo mejor de ambos. Monte Video, Uruguay. Universidad de la Republica.

Krall, C. (14 de Julio de 2019). ¿Qué es y para qué sirve UML? Versiones de UML (Lenguaje Unificado de Modelado). Tipos de diagramas UML. Obtenido de Aprendeaprogramar:

https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=688:ique-es-y-para-que-sirve-uml-versiones-de-uml-lenguaje-unificado-de-modelado-tipos-de-diagramas-uml&catid=46&Itemid=163

Lizcano, L. (2019). UML: Un Lenguaje de Modelo de Objetos. Cúcuta, Colombia. Universidad Francisco de Paula Santander.

López, P. Ruiz, F. (2019). Interacciones del Sistema. Cantabria, España. Universidad de Cantabria.

López, P. Ruiz, F. (2019). Lenguaje Unificado de Modelado – UML. Cantabria, España. Universidad de Cantabria.

Martínez, C. (2017). Cartagena, España. Universidad Politécnica de Cartagena.

Mediavilla, E. (2009). Programación orientada a objeto. Santander, España. Universidad de Cantabria.

Medina, J. (2005). Metodología y Herramientas UML para el Modelado y Análisis de Sistemas de tiempo Real Orientados a Objetos. Cantabria, España. Universidad de Cantabria.

Meléndez, S. Gaitan, M. Pérez, N. (2016). Metodología Ágil de Desarrollo de Software Programación Extrema. Mangua, Nicaragua. Universidad Nacional Autónoma de Nicaragua.

Méndez, M. (2010). Refactoring de Código Estructurado. La Plata, Argentina. Universidad de La Plata.

Mercedes, A. (2006). Diagrama de Actividad. San Salvador, El salvador. Universidad Don Bosco.

Orallo, E. (2002). El Lenguaje Unificado de Modelado (UML). Valencia, España. Universidad Politécnica de Valencia.

Pinelo, D. (2009). Introducción a UML. Mira Flores, México. Universidad Interamericana para el Desarrollo.

programasparapc. (2019). 5 programas para hacer UML y dominar el Lenguaje Unificado de Modelado. Obtenido de Programasparapc: https://programasparapc.net/programas-para-hacer-uml/#StarUML

Reyes, J. (2017). Diseño y Desarrollo de la Solución de Software de Gestión de Espacios Físicos en la Universidad Distrital. Bogotá, Colombia. Universidad Distrital Francisco José de Caldas.

Cortez, A. (2012). Diagramas de Tiempo y Fecha Local. Guanajuato, México. Universidad Virtual del Estado de Guanajuato.

Rivadeneira, S. Vilanova, G. Miranda, M. Cruz, D. (2013). El modelado de requerimientos en las metodologías ágiles. Rio Gallegos, Argentina. Universidad Nacional de la Patagonia Austral.

Ruiz, F. (2019). Ingeniería del software I, Arquitectura Física del Sistema. Cantabria, España. Universidad de Cantabria.

Rumbaugh, J. Jacobson, I. Booch, G. (2000). Lenguaje unificado de modelado manual de referencia UML 1.3. Madrid, España. Pearson Education.

Rumbaugh, J. Jacobson, I. Booch, G. (2007). El Lenguaje Unificado de Modelado Manual de Referencia 2.0. Madrid, España. Pearson Education.

Saavedra, G. Soto, G. Sierra, P. Yapuchura, G. (2019). Diagrama de componentes. La Paz, Bolivia: Universidad Salesiana de Bolivia.

Sánchez, M. Mora, A. (2015). Tema 6: Diagramas de Secuencia. Madrid, España. Universidad Carlos III de Madrid.

Sánchez, M. Mora, A. (2016). Tema 7: Diagramas de Colaboración. Madrid, España. Universidad Carlos III de Madrid.

Schwaber, K. Jeff Sutherland, J. (2013). La Guía de Scrum La Guía Definitiva de Scrum: Las Reglas del Juego. California, Estados Unidos. Scrum. Inc.

Sparks, G. (2018). Una Introducción al UML El Modelo de Componentes. Creswick, Australia. Sparx System.

Störrle, H. Knapp, A. (2006). Unified Modeling Language 2. Múnich, Alemania: Universidad de Múnich.

Sunitha, V. Samuel, P. (2018). Automatic Code Generation From UML State Chart Diagrams. Cochín, India. Cochin University of Science and Technology, Department of Computer Science.

UML: El modelo dinámico y de implementación. En *Upm*. Recuperado el 17 de julio de 2020 de: http://www.ieef.upm.es/moodle/pluginfile.php/2498/mod_resource/content/2/cap5_UMLDinamicov1.6b.pdf.

Vaca, P. Maldonado, C. Inchaurrondo, C. Peretti, J. Romero, M. Bueno, M. (2014). Estudio de Test-Driven Development en el Proceso de Desarrollo de Software. Córdoba, Argentina. Workshop de Investigadores en Ciencias de la Computación.

Vallecillo, A. Burgueño, L. Moreno, N. (2019). Módulo 4: Ingeniería Web Modelado. Málaga, España. Universidad de Málaga.

Vega, M. (2010). Casos de uso UML. Granada, España. Universidad de granada.

Vidal, C. Villarroel, R. López, X. Rubio, J. (2018). Una Propuesta de Algoritmo Spin / Promela para el Análisis y Diagnóstico de Errores en Diagramas de Secuencia UML. Santiago, Chile. SciELO - Scientific Electronic Library Online.

Yilong, Y. Wei, K. King, Y. Xiaoshan, L. (2018). Integrating UML With Service Refinement for Requirements Modeling and Analysis. Macau, China. Faculty of Science and Technology, University of Macau.

Zapata, A. (2006). Diseño del comportamiento: Diagrama de actividades. Zaragoza, España. Universidad de Zaragoza.

Zapata, A. (2009). Diseño estructural: Diagrama de clases. Zaragoza, España. Universidad de Zaragoza.