Antecedentes | Introducción

Tras surgir como disciplina y asentarse en la primera mitad del s. XX, la ciencia de la computación abrió paso a la ingeniería del *software*, y ambas progresaron vertiginosamente hasta encontrarse con un tejido industrial sólido y consagrado.

Los videojuegos, apéndices de ellas, nacieron de forma casual e inmersos en ese avance irrefrenable. Medraron bajo su amparo como rémoras durante años en que eran simples experimentos, obra de desarrolladores, hasta que iniciaron un paulatino distanciamiento de sus orígenes. Pero el hecho de haberse nutrido de ellos durante tanto tiempo condicionó al videojuego una vez pretendió consolidarse como materia independiente. Así, en las décadas de los setenta y los ochenta las innovaciones fueron meramente tecnológicas, aunque de gran éxito comercial, y para los años noventa ya la industria del videojuego era un calco multidisciplinar de la ingeniería del *software*, desde el punto de vista empresarial.

Poco a poco el videojuego aprendió a despegarse de su matriz como medio creativo, pero no como resultado de un proyecto. Los equipos punteros siguieron dependiendo de las tendencias en la ingeniería del *software* hasta que, durante la década pasada, la eclosión del desarrollo independiente convirtió la demanda en necesidad y la industria del videojuego se propuso por fin establecer sus fundamentos.

Actualmente presenciamos cómo el videojuego coge asiento sobre cimientos que más de tres décadas han mantenido inconexos: cada nuevo año es partícipe de un sinfín de publicaciones sobre diseño, y universidades de todo el mundo ya están ofertando titulaciones específicas de diseñador de juegos —algo impensable hace poco tiempo—; aparecen tesis que giran en torno a las metodologías de esta industria, sus peculiaridades narrativas o cómo seguir alimentándose de otras disciplinas; las tecnologías de desarrollo, hasta ahora prohibitivas, dieron un vuelco al panorama industrial al democratizarse a finales de la década pasada; nuevas formas de expresión en el arte plástico surgen de las propias restricciones del videojuego, como el denominado pixelismo o el bajo poligonaje [nota a pie de página con los términos originales en inglés]; incluso la creación musical ha visto nacer una ramificación suya con características muy particulares y nombres de figuras divinizadas, a la usanza del cine.

El videojuego, sin ninguna duda, se consagra hoy como disciplina y como medio.

Antecedentes | Diseño

En los albores del videojuego la figura del diseñador ni siquiera existía. Tampoco el diseño. El videojuego no era amalgama, sino pangea. Fue cuando los videojuegos ganaron complejidad que se empezó a dar peso a qué ofrecía como juego.

…

MDA.

Clockwork. Elegancia. Contrajuego. Diseño por sustracción.

Antecedentes | Metodologías | Introducción

La falta de formalización afecta también a las metodologías, aunque en menor medida. Parte de culpa reside en la controvertida concepción de que el videojuego es, en esencia, un proyecto de desarrollo de *software* convencional. Es cierto que para según qué asuntos es útil entender un videojuego como un *software* con ciertas peculiaridades, pero ¿qué hay de aquellos otros en que esas peculiaridades se ven involucradas? Hace mucho tiempo que ya nadie se plantea el diseño de juego como diseño convencional, por ejemplo.

En el caso de las metodologías, el videojuego se ha valido de sus orígenes para aprovecharse de las tendencias que sucedían en la ingeniería del *software*. Uno de los grandes potenciadores de las ciencias informáticas ha sido la evolución de las metodologías de desarrollo y la gestión de procesos. El desarrollo ágil dio un impulso incalculable a la industria del videojuego, pero el precio a pagar por haber mantenido tal actitud parasitaria no fue trivial.

Hasta que se produjo ese cambio los estudios sufrieron, al tratar inconscientemente sus proyectos como proyectos de *software*, la obligación de adaptarse a formas de proceder en las que sus desarrollos eran insolubles. Primero se adoptó el modelo en cascada, completamente contrario a la naturaleza del desarrollo de videojuegos. El modelo en espiral no fue muy tenido en cuenta por los estudios de la época —finales de los ochenta—, y de hecho, si bien habría solucionado el problema anterior con su enfoque iterativo, la cantidad de recursos dedicados a cada etapa es casi opuesta a la que demandan. Por último, el Proceso Unificado niega al desarrollo una flexibilidad crucial para los proyectos de videojuegos.

Como se ha dicho, fue con la llegada del desarrollo ágil cuando los desarrolladores vieron que de verdad se encontraban cómodos. Y esta vez el error fue adoptar directamente las metodologías. Ni siquiera dentro de un marco propicio para la esencia del videojuego servía con seguir aquello que al desarrollo convencional le funcionaba.

Con el tiempo los estudios intentaron convertir en estándares sus propias adaptaciones. Pronto se vio que la mayoría de equipos habían llegado a procesos casi idénticos, que no eran sino las transcripciones directas de las metodologías vertidas sobre las peculiaridades del desarrollo de videojuegos, pese a la total falta de formalización. De nuevo el eterno lastre que frena al videojuego por haber estado demasiado tiempo a merced de una disciplina que ya no lo contenía: crecimiento inabarcable para unos fundamentos, entre ellos los metodológicos, aún en pañales.

El mayor impedimento hoy día es que todas las propuestas son académicas o virtuales. Muchos se han atrevido a proponer metodologías desde las que configurar un cuerpo de conocimiento consistente, pero apenas hay casos de estudio reales que hayan seguido tales propuestas más allá del terreno pedagógico. Se aprecia en estas un patrón que las termina llevando al fracaso: todas dedican gran parte de su longitud a enumerar las proposiciones anteriores en el tiempo y a exponer por qué fallaron, haciendo hincapié en los mismos errores una y otra vez. Sin embargo, la especificación de la metodología en sí suele quedar insuficiente y genérica. Dan la sensación, leyéndolas de corrido, de referirse todas a algo que sirve sólo como proposición teórica.

En la realidad del desarrollo de videojuegos hodierno hay también unos patrones muy evidentes, que servirían para dar pie a esa metodología fundamental aún ausente.

Por un lado está el dominio casi unánime de las metodologías ágiles. Impuestas desde hace más de una década, desbancaron por motivos evidentes a las que mejor aceptación tenían entre los equipos de desarrollo de videojuegos, como el modelo en cascada. La naturaleza iterativa y la necesidad de mostrar pronto la experiencia jugable, así como el acercamiento y cooperación que suscitan en todo el equipo, hicieron del desarrollo ágil la base con la que desarrollar videojuegos.

Todos los estudios, desde aquel con mayor presupuesto hasta el más solitario desarrollador que incorpora gestión a su proyecto, coinciden en compartimentar la planificación ágil en tres grandes grupos: preproducción, producción y posproducción.

Por otro lado, los equipos suelen reconocer que son los diseñadores quienes tienen que guiar la metodología. Es el diseño del juego y no su desarrollo el eje sobre el que rotar la planificación.

Además, la reiteración en el error hace que los equipos recomienden adelantar todo lo posible la creación de audio y música, defenestrados casi siempre a un plano de menor relevancia en la asignación de recursos.

Estas y muchas otras convergencias, que se estudiarán en mayor profundidad a continuación, demuestran que hay una formalización tácita y que sólo haría falta que los expertos del sector se congregasen para ponerla de manifiesto.

Pero, para ser justos, hay que desgranar las metodologías según su criterio de aplicación y juzgar por separado hasta qué punto están normalizadas. No hacerlo sería caer en el mismo error de tratar el videojuego como *software* convencional, y ¿dónde si no en las metodologías puede ejemplificarse mejor?

Antecedentes | Metodologías | Metodologías de desarrollo

Las metodologías más evidentes son las que se aplican al desarrollo, entendiendo aquí el desarrollo como la sinécdoque del proyecto completo. Las metodologías para el desarrollo de videojuegos beben directamente de la gestión de proyectos convencional. De hecho, todas las propuestas bien conocidas se reducen a adaptar a este dominio una o varias metodologías de desarrollo de *software*. A continuación se ordenan cronológicamente tales proposiciones, adjuntando tras ellas algunas publicaciones igual de interesantes pese a no plantear una metodología como tal.

Waterfall Game Development

Pese a que el modelo en cascada data de los años setenta [referencia a Winston Royce], el crecimiento tardío de los videojuegos respecto a la ingeniería del *software* hizo que en esta industria no se adoptase hasta bien entrados los noventa. En los setenta los videojuegos de renombre aún eran proezas tecnológicas sin apenas desarrollo como tal, y el resto se reducía a personas inquietas que desarrollaban como complemento a su afición por las computadoras.

Cuando en apenas una década los videojuegos sentaron su paradigma de desarrollo, para convertirse en proyectos mastodónticos, la necesidad de abrazarse a alguna metodología apareció. Y ahí ocurrió que los desarrolladores eran, valga la redundancia, meros desarrolladores, y por tanto sabían cuáles eran las metodologías del desarrollo convencional, pero no fueron lo suficientemente inventivos como para adelantarse a las trabas. No había una especificación que ofreciese alternativas. La ingeniería del *software* era la única vía para el desarrollo del videojuego, pese a que empezaban a vislumbrarse nuevas exigencias que quedaban descubiertas.

En los años en los que el modelo en cascada dominó el desarrollo de videojuegos, el colectivo lo fue puliendo poco a poco hasta implantarlo aceptablemente. Eso sí, los problemas inherentes al desarrollo secuencial siguieron siendo insalvables, y de ahí que el modelo se abandonase en cuanto fue posible.

Las fases por las que pasaba el modelo adaptado a videojuegos eran:

* Concepción: analistas de negocio, productores, gerentes y jefes de desarrollo discutían el concepto. Audiencia, plataforma, detalles técnicos… La escritura de un documento conceptual [referencia a su apartado en metodologías de diseño] era optativa —lo cual cambió drásticamente cuando la industria se ensanchó—. Llama la atención cómo el equipo de desarrollo, incluso el de diseño, era apartado de estas decisiones. En verdad aún no estaba muy dibujada la franja entre diseñadores y productores de contenido.
* Especificación: toda la documentación quedaba generada. El documento de diseño guiaría el resto de las fases, y era reforzado por biblias narrativas y artísticas, especificación técnica y otros anexos. Aquí hay que poner de manifiesto cómo los equipos traían el diseño de *software* intacto a sus metodologías. Las especificaciones técnicas, sobre todo la arquitectura, eran modeladas usando diagramado en UML, lo cual quizá ha quedado algo relegado actualmente a equipos con mayor poder pecuniario.
* Construcción: los desarrolladores elaboraban aquello que sus documentaciones respectivas especificaban, mientras los gerentes les dirigían. El proceso creativo estaba ya restringido en este punto.
* Verificación: las piezas del juego, ya desarrollada, se enfrentaban a los documentos iniciales.
* Prueba de jugabilidad: con implicación del equipo completo, incluyendo gerentes, productores, editores y hasta distribuidores, se generaban pruebas de experiencia con realimentación. Era en este punto donde empezaban los problemas y las replanificaciones mal calzadas.
* Alfa: cuando el juego estaba preparado se pasaba a evaluadores terceros.
* Beta: los evaluadores aprobaban las alfas y entonces gente ajena al proyecto daba una segunda pasada de verificaciones. Estas versiones eran en resumidas cuentas muy similares al producto completo, generalmente porciones truncadas si el objetivo era el consumidor final. A veces tenían finalidad mercadotécnica. Para entender cómo ha cambiado la industria desde entonces, hay que recordar que la administración de versiones beta en consolas era prácticamente imposible [referencia a Flood], y que los consumidores finales no se consideraban verificadores útiles.
* Fase dorada: aún hoy se habla de que un juego entra en etapa g*olden* cuando su desarrollo oficialmente termina y se envía a las distribuidoras. Huelga decir que en aquella época el despliegue era uno y definitivo. No había noción de mantenimiento en el sector del videojuego, ni siquiera de actualización, porque no existían los medios necesarios.

En la figura [meter figura numerada, figura en *Is Agility Out There?* Página 4, figura 1] se resume el flujo del modelo en cascada habitual adoptado a videojuegos.

Extreme Game Development (XGD)

Extreme Game Development es una de las formalizaciones más ilustres del sector. Fue propuesta en 2003 como aproximación necesaria de Extreme Programming al desarrollo de videojuegos, aunque para entonces ya varios equipos habían hecho adaptaciones ingenuas de XP en desarrollos de cierto éxito. De hecho, los autores comienzan mencionando uno de estos casos: el *Spiderman* de Treyarch [referencia más trabajada al juego]. El reto a la hora de formalizar tal adaptación residía en lo concerniente al personal no técnico, que era donde más se distanciaban las diferentes implementaciones conocidas.

Dado el sempiterno problema de la industria del videojuego con los retrasos, XGD aprovecha para basar sus principios en minimizar el riesgo con las fechas. Basándose en la popular colección de *post mortem* de Gamasutra [referencia, de algún modo], los autores resumen en tres las tesituras que llevan a los equipos a retrasar sus planificaciones:

* Evolución de la tecnología: los equipos vivían bajo constante presión por aventajarse en la publicación en nuevos dispositivos. Actualmente esto se ha mitigado gracias a la homogeneidad del mercado, pero a principios de siglo aún se requería prácticamente un desarrollo distinto para cada plataforma (podían reciclarse las partes independientes, como el diseño de juego). Sólo los proyectos de mayor presupuesto tienen aún en sus preferencias este tipo de cuestiones [probable mención al caso de *Battlefield 4* con PS4].
* Autoexigencia: los miembros del equipo suelen tener problemas para limitar el potencial de sus tareas. Es importante que cualquier creador de contenido —programador o artista— conozca en todo momento o bien la fecha de vencimiento o bien la calidad mínima buscada, que puede usarse como umbral.
* Instrucciones de producción: los intereses, negociaciones y enfoques de los productores pueden provocar cambios súbitos en el proyecto, y las metodologías convencionales no están preparadas para asumirlos.

El objetivo de XGD es proporcionar un marco en que los cambios sean bien afrontados y el equipo se sienta orgulloso de su trabajo. Para ello se deben superar algunas asunciones problemáticas de la industria, siendo la más importante el documento de diseño. Los autores ven el documento de diseño como una traba que dificulta los cambios y vuelve pasiva la cooperación. La alternativa que proponen es una colaboración activa y no contractual entre el equipo de desarrollo y el de producción, edición e incluso distribución.

Por supuesto la figura del contrato debe mantenerse. De hecho, se ve potenciada: los desarrolladores se comprometen a entregar una primera versión a los productores cuanto antes, y estos aprobarán o no más iteraciones sobre el desarrollo según consideren que el producto está listo para su despliegue.

Esta metodología adapta directamente los requisitos del proceso de desarrollo del Manifiesto Ágil [referencia]. Es, en definitiva, una metodología ágil con dos peculiaridades: debe enfrentarse a la presencia de diseño de juego y creadores de recursos, así como incluir pruebas ausentes en el desarrollo convencional, como el factor de diversión.

Para fundamentar Extreme Game Development, los autores repasan los cuatro valores y trece principios de la original XP.

* Sencillez: es el pilar clave de Extreme Programming, cuya consigna dice: «haz lo más sencillo que vaya a funcionar». En cualquier proyecto donde intervengan desarrolladores de *software* hay que poner límites a las expectativas de escalabilidad, mantenibilidad y buen diseño que estos deben tener. Si no, el desarrollador engrosará su esfuerzo en el proyecto en pos de una sostenibilidad muy por encima de lo requerido.
* Comunicación: crucial en videojuegos. Varias disciplinas diferentes deben mantener completo entendimiento entre ellas. Algunos documentos son males necesarios —el documento de diseño, el concepto, la biblia narrativa…—, pero deben primar la legibilidad a la longitud. Quien tiene una idea no la escribe; la notifica al resto.
* Retroalimentación: muy ligada a lo anterior. Es un error categórico reducirla a un simple aunque necesario *post mortem*, ya que un equipo pequeño puede no tener una segunda oportunidad en la que aplicar las lecciones aprendidas. Los autores hacen una genial analogía con un termostato que se enciende sólo cuando alguien abandona la estancia. Su propuesta consiste en una especie de *post mortem* periódico, independiente de las pruebas y de las fechas del resto del plan. Esta rutina debe comenzar incluso en preproducción, ya que los equipos son más reacios a digerir realimentación cuanto menos lo hacen.
* Valentía: que los hitos no sean inmutables puede mermar la seguridad de varios trabajadores. Sobre todo de editores y productores. No es sino un gran cambio de paradigma en la forma de planificar el proyecto. De hecho, la falta de programación en fase de preproducción puede confundir mucho a trabajadores acostumbrados a metodologías más cerradas [posible nota al pie aclarando que esto ya está más que asumido].

Al trasladar todo lo anterior al desarrollo de videojuegos surgen los principios básicos de Extreme Game Development:

* Incluir a distribuidores, editores y productores en el equipo de desarrollo. Quizá esto es lo más complicado de llevar a cabo, ya que actualmente el modelo no lo soporta demasiado bien (los distribuidores son una empresa externa y rara vez alguna de esas tres figuras convive siquiera en edificio con el equipo).
* La figura del cliente recae sobre el productor. Como se ha comentado antes, en desarrollo de videojuegos no tiene por qué existir un cliente al uso. Salvo subcontratas, proyectos bajo demanda de autónomos y emprendedores y proyectos publicitarios [nota al pie explicando qué son. Los típicos juegos de marcas], ¿quién es en realidad el cliente de un videojuego? ¿El diseñador, como miembro cuyo deseo se lleva a producto? ¿El productor, quien firma y aprueba? ¿El distribuidor, que es quien recibe el producto? ¿O el consumidor, que paga por él? Manteniendo la perspectiva de proyecto el consumidor es evidentemente descartado, si bien habría que cavilar en más profundidad sobre ciertas situaciones como la financiación mediante micromecenazgo. Desde el punto de vista empresarial, el diseñador no tiene la responsabilidad del proyecto, ni tampoco el distribuidor, que puede incluso ser cliente del propio estudio.
* El desarrollo debe girar en torno a hitos de corta duración constantemente programados y refinados. Con independencia de la programación global de hitos, cada ciclo [de seis semanas en el ejemplo que ponen los autores en su empresa] comienza con una planificación mediante historias de usuario.
* Cada ciclo se divide en tres periodos secuenciales de igual duración, a saber: definición, organización y elaboración. Esta es una de las propuestas de XGD más difíciles de seguir actualmente.
* La integración, como conocido cuello de botella, se adelanta lo máximo posible. Pasa a ser otro elemento fragmentado en iteraciones, evitando muchos esfuerzos innecesarios del equipo de arte (es conocimiento popular que en torno a un tercio de los recursos creados no forman parte del producto final). La construcción debe estar automatizada —se realizará en las noches—, así como la exportación de recursos.
* El equipo es visto como un conjunto. Los trabajadores son intercambiables entre sí en la medida de lo posible (evidentemente alguien del equipo de arte no programará lógica ni un diseñador creará efectos gráficos). Se evita por tanto la especialización, y he aquí algo que actualmente es inviable en grandes desarrollos. A diferencia de ellos, los equipos independientes suelen estar compuestos por “miembros orquesta”, por lo que se adecuan mucho mejor a este requisito.
* Importancia radical de las pruebas. Deben estar automatizadas y tolerar proceso por lotes. Se diferencian dos tipos de pruebas a pequeña escala:
* Pruebas funcionales: definidas simultáneamente y añadidas a la cadena global. La cadena de pruebas podría verse como una cola circular que ejecuta repetidamente todas las pruebas añadidas a ella.
* Pruebas unitarias: responsabilidad de los creadores —programación y recursos—.
* Esto no significa que se prescinda de las pruebas de calidad, que formarían un último tipo a gran escala. ¡Probar la jugabilidad no es algo automatizable!
* Se heredan de la metodología original las reuniones diarias cortas. Esta característica ha sido muy cuestionada en el desarrollo de videojuegos [ver tal sección].
* Se estipulan reglas de estilo, codificación y nombres. Estas reglas deben ser concisas y no dar pie a interpretaciones, ya que si el equipo no las sigue a rajatabla se perderán las ventajas de la no especialización y las pruebas automatizadas.
* Por último, los autores proponen el uso de UML para los diseñadores, ya que es una oportunidad estupenda para aprovecharse de un modelo consolidado. Esta es quizá la característica de XGD que peor ha soportado el paso del tiempo. Las necesidades del diseño de *software* distan mucho de cubrir las del diseño de juegos, que en el año de publicación de esta metodología aún era, prácticamente, una materia por descubrir —si es que no lo sigue siendo hoy— [referencia a la sección].

Game Unified Process

De forma coetánea a la formalización de XGD, Kevin Flood publicaba su artículo sobre el agotamiento del modelo en cascada y su consiguiente propuesta, Game Unified Process. Hablando de los *post mortem* de la época, Flood se quejaba en el inicio del texto de ver una y otra vez los mismos problemas: retrasos, exceso de errores en el producto final, tiempos de desarrollo desbalanceados, fechas apretadas… y se pretendía poner por fin de manifiesto todos esos errores con implicaciones de negocio. No hay que olvidar que Kevin Flood ocupaba cargos de ingeniero de *software*, pero defendía que eran los desarrolladores de contenido quienes deben interesarse *motu proprio* por las metodologías, ya que son ellos quienes tienen mayor influencia en el proceso.

Toda la propuesta de Game Unified Process gira en torno a la experiencia de Flood en Wagerworks, donde un proyecto de juegos de casino le hizo ver que el modelo había sido esquilmado. En ese proyecto había muchos módulos diferentes trabajando en partes del juego, con tecnologías tan diversas como J2EE, Weblogic, Flash, Informix, XML y Solaris.

Bajo tal contexto no era una locura que cada módulo usase su propia metodología predilecta. En el caso que ocupaba al autor se trataba de modelo en cascada adaptado a videojuegos. Era el modelo más usado pese a todos los problemas que ofrecía su carácter secuencial, de fases sucesivas sin vuelta atrás. Y es que esa naturaleza introducía un problema insalvable para el desarrollo de videojuegos: la pobre gestión de riesgos. Los inconvenientes se localizaban muy tarde y la metodología ya no estaba preparada para hacerles frente de manera natural. Se tenía que recurrir a pequeñas planificaciones modulares sobre la marcha y los errores escalaban.

Esos problemas, recurrentes en ingeniería del *software*, dieron con el nacimiento de los enfoques ágiles y el Proceso Unificado de Desarrollo. Todas las nuevas vertientes reconocieron que el desarrollo de *software* es iterativo por naturaleza. Se adelantaron las interacciones y los involucramientos, y todos los grupos del equipo comenzaron a tomar parte desde el principio en cada una de las fases.

De forma pionera y no muy acertada, el equipo de Kevin Flood adoptó Proceso Unificado de Desarrollo y lo aproximó ligeramente a las necesidades del videojuego, llamándolo Game Unified Process. Los artefactos —esto es, la documentación— y casos de uso se convirtieron en el eje del desarrollo, y la documentación pasó de ser algo estático a reflejar el estado actual del proyecto. Pero no todo el equipo de aquel proyecto migró a GUP: los desarrolladores optaron por usar Extreme Programming trabajando en pequeñas divisiones, y el propio equipo de ingeniería sentía la necesidad de volver al hábito del modelo en cascada cuando las cosas se torcían. El proyecto no formalizó por completo la adopción del Proceso Unificado de Desarrollo a los videojuegos, y esta no fructificó.

SUM

En 2009, varios alumnos de la Universidad de la República de Uruguay presentaron una metodología ágil para desarrollo de videojuegos llamada SUM. Según los autores esta metodología fue creada para adaptarse a la realidad de la industria uruguaya, compuesta por estudios independientes, lo cual descarta su uso en proyectos de cierta envergadura. SUM es por desgracia una propuesta superficial que hace aguas a la hora de definir los procesos que la conforman.

Se plantea como una unión de SCRUM y XP en la que algunas modificaciones promueven su adaptación al desarrollo de videojuegos:

* El equipo de desarrollo pasa a estar formado por cuatro subequipos, a saber: programadores, artistas gráficos, artistas sonoros y diseñadores.
* Aparece la figura del verificador beta, en analogía con el *tester*.
* El productor interno del videojuego suple al *SCRUM Master*.
* El cliente, si lo hay, hace el papel de *Product Owner*. No se especifica quién recoge el rol de cliente en caso de ser necesario.

La división de iteraciones que SUM propone está sintetizada en la siguiente figura. [meter figura numerada] Las dos primeras fases, concepto y planificación, y la última, cierre, se realizan en una sola iteración. La elaboración y la publicación de versiones, que los autores restringen a la denominación «beta», siguen tantas iteraciones secuenciales como la planificación mande. Una fase continua, la gestión de riesgos, envuelve a las demás.

SUM incurre en no profundizar en su propia taxonomía. Divide sus fases en etapas casi siempre secuenciales, pero no detalla correctamente el papel de esas etapas: entregables, salidas, roles implicados, etc. Por el contrario, menciona someramente a qué acción hace referencia cada una, lo cual se desprendía ya de la figura anterior.

Como el lector podrá comprobar, SUM se reduce a una versión descafeinada de lo que Game-SCRUM documentaría en mayor exhaustividad poco tiempo después.

Game-SCRUM

Desde su publicación en 2010, además de llevar varios años divulgándose, Game-SCRUM ha sido mencionada asiduamente en publicaciones y conferencias sobre metodologías de videojuegos. Para entonces el uso de SCRUM ya era mayoritario en la industria del videojuego, pero su trabajo sirvió como punto de partida desde el que formalizar los procesos que la mayoría de equipos estaban siguiendo a nivel conceptual. De hecho comienzan dejando claro que, si bien las metodologías ágiles se han vuelto obligatorias en desarrollo de videojuegos, cada equipo ha tenido que adaptarlas a su realidad para sacarles partido. Hablan sobre lo idóneo del desarrollo ágil gracias a la rapidez de resultados palpables, su importancia en el crucial descubrimiento precoz del factor de diversión [referenciar a aclaración sobre el concepto de diversión] y cómo mantiene implicados, comunicados y cooperantes a todos los miembros.

Achacan a la industria poca cantidad de trabajos sobre ello en la bibliografía actual y, a los existentes, su carencia de uso práctico datado como efectivo. Y no sólo denuncian falta de formalización en las metodologías, sino incluso de conocimiento sobre problemas recurrentes. Mencionan a Petrillo [mejorar esta referencia y añadir la concreta] respecto a estos errores comunes, quien por un lado los relaciona con la ambición, el alcance o las estimaciones y por otro con la naturaleza convergente del videojuego (insinúa que el problema es la comunicación entre desarrolladores y artistas, aunque pasa por alto el mismo inconveniente entre diseñadores y desarrolladores, aún más grave).

El gran contraste entre esta situación y la que vive la ingeniería del *software* lleva a los autores a aludir al Manifiesto Ágil y a dos pilares del modelo: SCRUM y XP.

La metodología que ellos proponen a partir de las dos anteriores, Game-SCRUM, toma de la primera lo referente a la planificación y de la segunda el enfoque centrado en la ingeniería. Sin embargo, ninguna de esas dos bases se adecua por sí sola al desarrollo de videojuegos, en que la distribución no es uniforme a lo largo de las iteraciones [Keith 2010]. La forma de paliar esto es, como se adelantó, encajonar el desarrollo iterativo en tres grandes fases: preproducción, producción y posproducción.

Durante la preproducción, casi toda la carga de trabajo recae sobre diseñadores. Es en esta fase donde se discuten las mecánicas principales, se prototipan para validarlas o revisarlas y se genera la primera gran dosis de documentación. El trabajo de programadores y artistas se reduce a posibilitar los prototipos que los diseñadores requieren. Estos prototipos, y he aquí uno de los errores que los equipos independientes solían cometer a menudo, deben ser independientes, volátiles y suponer muy poco trabajo. No importa que la codificación sea sucia, porque los prototipos bajo ningún concepto se conservarán tras la preproducción. En muchas ocasiones los artistas ni siquiera son solicitados, ya que los programadores usan recursos básicos para prototipar.

Una buena preproducción adelanta el encuentro de los diseñadores con el elusivo factor de diversión, y exonera a futuras iteraciones de tareas de experimentación, de manera que la producción se centra en el desarrollo en sí [Kanode y Haddad 2009]. La dirección que tomará la fase siguiente es zanjada aquí, sin sofocar con ello la creatividad que pudiera surgir después. El trabajo de la preproducción es encontrar el concepto del juego, y eso es algo inalcanzable sin ensayo y error. Los autores recomiendan a los diseñadores —junto con el resto del equipo para evitar la rotura de la cooperación— métodos colaborativos como las tormentas de ideas.

Una vez el equipo de diseño confía en haber dado con el concepto que el estudio busca, la planificación se sella mediante el documento de diseño, y después se trasiega a la reserva de tareas de la metodología. Traducir correctamente el documento de diseño en tareas, que no tienen por qué ser atómicas, es tan importante como la propia preproducción.

La fase de producción se basa en iterar sobre las tareas de la reserva. Si el diseño de mecánicas o de niveles se ha delegado también a una creación iterativa, en esta fase los diseñadores seguirán el mismo bucle que los desarrolladores y los artistas. Aquí surge un problema, según Keith [Keith 2010], cuando los equipos en que la cantidad de artistas empieza a ser elevada. El equipo artístico trabaja de forma muy secuencial entre puestos distintos, ya que uno depende directamente de las creaciones de otro (esto es fácilmente visible por ejemplo al pensar en modeladores, diseñadores de esqueleto y animadores, que forman una cadena en serie). Ello implica la necesidad de una especificación metodológica para los artistas, y aquí Keith propone el uso de Kanban. A diferencia de SCRUM, Kanban permitiría al equipo promocionar de iteración con trabajo por terminar. Es decir, mientras ciertos artistas —y sólo artistas— se mantienen en la iteración anterior, el resto del equipo pasa a la siguiente. La meta es lograr un flujo continuo de creación de contenido, aunque los autores no detallan cómo implementar el uso de Kanban. Una buena idea podría ser aplicar encauzamiento al equipo de artistas, que sería dividido en etapas y por orden de dependencias.

Por último, Game-SCRUM relaciona directamente la posproducción con la escritura de un documento post mortem, para recoger la propia experiencia que deja el proyecto y su realimentación una vez desplegado.

Cabe ser mencionado que los autores rehúyen hablar sobre verificación y pruebas. Las ubican al comienzo de la posproducción sin detenerse en ellas, cuando en realidad aplicar a esta metodología pruebas en posproducción sería nefasto. Por supuesto, en la posproducción se mantiene la panoplia de pruebas que trata de sacar errores a la luz, pero lo más adecuado parece incluir la verificación como parte de las propias iteraciones.

Errores al aplicar SCRUM

Más allá de proponer metodologías, algunos autores han disertado sobre temas muy ligados a ellas. Es el caso del reputado desarrollador Paul Miller, quien aportó en 2008 un decálogo de fallos recurrentes en el uso de SCRUM para videojuegos. Según introduce, la mayoría de estas faltas son consecuencia de no seguir buenas prácticas bien establecidas o de intentar reemplazarlas innecesariamente. Son las siguientes [resumidas en tabla]:

* La reserva del tablero sustituye al documento de diseño: se extendió la creencia de que ambos son suplementarios y redundantes. Ello lleva a descuidar el tiempo dedicado al documento de diseño y como resultado la especificación de diseño se torna más pobre. Se cede responsabilidad e interpretación al desarrollador y al artista; se pierde normalización. Una reserva de tareas no sustituye al documento impreso, que permite anotación propia y materializa el trabajo. Ya sea impreso o en la nube, incluso simbolizado como wiki en el control de versiones, el documento de diseño aporta comunicación.
* Interrupciones para la reunión diaria: programar la reunión en un momento intermedio puede cortar el flujo de trabajo. Ya que son reuniones de carácter no técnico, pueden desplazarse a conveniencia. A fin de cuentas, el objetivo es evitar el gasto de jornadas completas, para lo cual la mayoría de veces no será necesario este esfuerzo. Una buena alternativa es la monitorización de tareas abierta a todos los miembros. En este punto el autor propuso varios hábitos que hoy día ya son imprescindibles en los gestores de proyectos: listas, etiquetas, porcentaje de terminación, fechas de vencimiento…
* Separar habitaciones por departamento: la comunicación es clave en el desarrollo de un videojuego y también en una metodología ágil, ergo es estúpido ponerle obstáculos. De hecho, debería eliminarse cualquier traba en la comunicación que el equipo pueda padecer. Se propone el uso de escritorios móviles, uso de equipos portátiles, comunicación inalámbrica y cualquier elemento que permita desplazarse rápidamente. Aligerará todos los momentos en que varios miembros del equipo deben trabajar en conjunción, así como las etapas finales en que algunos recursos son usados sólo como apoyo de otros.
* Miembros expectantes a asignación de tareas: aquí aparece la clásica diferenciación entre la actitud proactiva y reactiva. Limitar la visión de conjunto del equipo a base de encajonarlos en la longitud de las iteraciones genera reactividad. Añadir a la reserva tareas que uno debe cumplir a corto plazo es análogo a pedir permiso, a menudo para trivialidades. Por el contrario, hacer que el desarrollador conozca las necesidades próximas [el autor se refiere al próximo mes] ayuda a su proactividad.
* Comienzo de administración de tareas previo a tener el equipo completo: planificar historias de usuarios a completar antes de que haya verificador de calidad es un fallo de difícil corrección. Iniciar el proyecto sin control de calidad durante el primer mes desemboca en fechas comprimidas y tiempos finales de crisis [crunch]. Está bien asumido que los desarrolladores deben arreglar errores según estos aparezcan, y la lista de errores debe ser mínima. No obstante, a los equipos les cuesta más aceptar que sin control de calidad no es posible tener constancia de la mayoría de errores. El control de calidad reduce los tiempos de crisis al final del desarrollo. Una buena práctica es limitar la cantidad de errores abiertos, de manera que nunca superen al número de tareas asignadas. Aparte de eso, los trabajadores deben tener vía libre para preguntar a cualquier otro miembro si creen que puede tener errores dependientes o derivados de los suyos propios.
* La reunión diaria se convierte en una mesa redonda: sin acción, sin valor remarcable, la reunión diaria es inútil. La reunión diaria no es algo global, sino que debe aportar valor al propio empleado que ha dejado su trabajo para asistir a ella. El coste del resto no supera el beneficio de ahorrar el tiempo de la reunión. Es preferible dedicar los recursos de las reuniones a maximizar el plan de los miembros, algo mucho más importante. Las reuniones que deban programarse, se harán con agenda, descripción, interesados y vencimiento, de manera que el empleado pueda estar al corriente de cuándo hacer franjas en su plan propio.
* Forzar el SCRUM: por ejemplo, si los gerentes llevan a cabo los procesos de estimación cuantitativa sin realimentar a los participantes. La comunicación debe ser bilateral siempre, desde la cadena de mando hacia abajo. Lejos de cortarla, si hay que forzar algo eso es la comunicación; en la consecución de hitos y objetivos, el refuerzo positivo nunca es un error.
* Planificar o diseñar sólo la próxima iteración y desarrollar en función del conocimiento actual de diseño: el desarrollador ignora cambios potenciales en el diseño y, volviendo a lo anterior, no trabaja en tareas no asignadas. El código ascendente, propuesto por el enfoque ágil, no es mejor que el descendente. Por la incertidumbre del diseño, suele descuidarse su sostenibilidad, que no tiene que ser enfermiza pero sí cumplir unos mínimos. Por tanto, es un acierto trabajar de baja a alta fidelidad.
* La preproducción se corresponde con el *sprint zero*: este aforismo es un error habitual al querer obligar la metodología por encima de las buenas prácticas. La preproducción termina consistiendo en colocar tareas en el tablero durante dos semanas y producir un gran flujo artístico que, salvo lo conceptual, será descartado. Nadie puede solicitar la creatividad en un momento concreto; la creatividad no puede ser planificada. Tener mucha práctica en la elaboración de prototipos puede ayudar a acomodarse a lo anterior, y también las herramientas de prototipado, pero ayuda más al proyecto modificar convenientemente la planificación.
* La estimación de tareas da comienzo antes de escribir el documento de diseño: en cada planificación de iteraciones, repetir el comportamiento de priorizar tareas antes de haber diseño es tan dañino como darle al licenciante la firma antes de que vea el concepto, o incluso otorgarle potestad para imponer cualquier cambio de diseño durante la producción. Como ya se ha reiterado, el diseño de juegos es el eje sobre el que el proyecto debe girar. Esto ha sido bien asimilado con el paso del tiempo, llevando a los estudios a adelantar lo máximo posible un borrador del documento de diseño, que se asemeja a lo que actualmente es el documento conceptual [referencia a su apartado]. Su creación requiere realimentación de todos los interesados, estar bien entendido por parte de equipo de desarrollo e ingenieros y por supuesto ser previo a la creación y estimación de tareas. Manteniendo la fuerza de las buenas prácticas sobre la metodología en sí, es conveniente hacer caso a los malos presentimientos sobre el desarrollo por encima de los consejos del asesor metodológico.

A modo de conclusión, el autor se refiere a SCRUM en particular y la propuesta ágil en general como una herramienta de trabajo más, y no como una panacea. Si se usan mal pierden su valor, llegando a ser contraproducentes. El buen uso de cualquier metodología de gestión de proyectos se gana con la experiencia; sobre todo, con las malas experiencias. Por tanto, usar de forma cruda una metodología es involucionar. Es desechar prácticas bien establecidas que el mal uso de esa metodología sacó a la luz.

Inspección de errores de la industria

Uno de los documentos más citados en el desarrollo de videojuegos es el que publicaron Fabio Petrillo, Marcelo Pimento y otros [referencia al largo]. En él, resumido además en un artículo de publicación previa [referencia al corto], los autores comparaban los problemas de la industria del videojuego con errores bien conocidos de la industria del *software* tradicional. Evidenciaban, de nuevo, falta de formalización incluso en algo tan inmediato como reunir y categorizar los errores comunes.

Para cubrir ese hueco, seguían la división tradicional en cuatro categorías de Yourden [referencia] y de Charette [referencia]: planificación, presupuesto, calidad y negocio.

En este artículo se hace referencia a Flood, dado que él se quejaba ya a principios de siglo de que todos los documentos *post mortem* son el mismo, haciendo de esta práctica algo genérico y sin valor. [referencia] Y también a Berthke, en su acusación de que la industria del videojuego suele adoptar metodologías muy pobres [referencia] o muy mal implementadas.

Así, los problemas principales se compendian en:

* Problemas de alcance: según Flynt y Salem [referencia], la mayor causa de imperfección es no delimitar un alcance sin fisuras. No guiarse por un objetivo férreo hace que cada cambio sea un potencial inoculador de caos. Aquí aparece el tan extendido término de “hiedra de características”, o “características enredadera” [pie de página con la original *feature creep*]. Las características enredadera engrosan el proyecto subrepticiamente si sinergias complejas o nuevas cualidades seducen al equipo de desarrollo, si se introducen componentes externos de difícil integración o si se opta por usar algoritmos propias en lugar de librerías. No hay que descartar, sin embargo, cualquier cambio que llame a la puerta del proyecto. Todo lo contrario: un cambio propuesto formalmente por el equipo debe ser abrazado desde la gestión de riesgo y, si resulta viable, añadirse a la programación del proyecto. Varios juegos exitosos ganaron enteros gracias a características añadidas a última hora, lo cual demuestra, una vez más, que el videojuego no es un proceso lineal [Flood también].
* Problemas de planificación: son habituales las estimaciones irreales, la incompetencia estableciendo fechas de vencimiento y los problemas derivados de la naturaleza multidisciplinar del equipo. Los dos primeros suelen llegar, según Flynt y Salem [referencia], por la falta de datos históricos y la consecuente percepción pobre. También por caer en estimaciones crédulas, olvidando añadir tiempo para corrección de errores sobre la marcha, reuniones, otras imposiciones de la metodología, etc.
* Compresión de calendario: también presente en la ingeniería del *software* tradicional, pero no con tanta frecuencia como en los videojuegos. Los problemas de fechas son una de las mayores fuentes de complicaciones en cualquier proyecto, incluso en aquellos de mayor presupuesto y nombradía [referencia a Presura]. Uno de los mayores errores, ya al parecer irresolubles, de la industria del videojuego fue aceptar como propia la denominada “hora de la verdad” [pie de página con el original *crunch*]. Dice Hamann [referencia] que los productores toman por decreto los tiempos de crisis como ley de vida del desarrollo, y lo compara acertadamente con el devoto de fe ciega para con un dogma.
* Problemas tecnológicos: las tecnologías usadas en videojuegos están tan avanzadas y continúan progresando tan velozmente que son a la vez punteros en la computación gráfica (aquí el autor parece olvidar que, históricamente, la evolución tecnológica en gráficos de videojuegos ha sido hermana pequeña de la animación por computador). Cuando una plataforma no está consolidada, usarla para un proyecto es condicionarse con un gran riesgo de inicio; la plataforma neonata tendrá poca documentación de terceros, y quizá también propia, y es probable que la tecnología contenga problemas que sólo sean desvelados tras el lanzamiento del producto o, en cualquier caso, en una fase demasiado tardía.

Los autores hicieron para el artículo un estudio a una muestra de proyectos completados. Entre ellos aparecen nombres populares como *Unreal Tournament, Black & White, Resident Evil, Diablo II* o *Rainbow Six*. Los problemas que reconocieron los desarrolladores de la muestra se sintetizan en la tabla [referencia a tabla numerada]: tres cuartas partes de ellos manifestaron un alcance demasiado ambicioso y características enredadera. Casi tantos hablaron de características cortadas durante el desarrollo. En menor medida, los encuestados coincidieron en problemas de diseño, planificación optimista y tecnologías problemáticas.

Hay varias conclusiones inmediatas de la encuesta. La mayor parte de los problemas tradicionales se traspasan al desarrollo de videojuegos. La tabla lo refleja en su prominente primer lugar. En esencia, puede simplificarse a que los problemas no son tecnológicos sino de gestión. Y además de los problemas trasegados del desarrollo tradicional, aparece la división del equipo por heterogeneidad y la aparición de nuevos requisitos, como la elaboración del diseño o el factor de diversión.

Tras haber sufrido para sonsacar la información que requerían, los autores sellan el artículo con una extraordinaria lección: el carácter competitivo daña la formalización de la industria del videojuego.

Prácticas ágiles escondidas

Los autores anteriores aprovecharon el estudio de sus dos trabajos [referencia a corto y largo] para una nueva investigación. Se pone un grito más en el cielo respecto la fundamentación del videojuego: aunque en lo referente a lo tecnológico la industria goza de una vasta documentación, esta salud se desbalancea en el resto de ámbitos que la conforman, incluyendo a la ingeniería. De hecho, son en aquel momento los trabajos de Bethke y de Flynt y Salem los dos únicos de cierto peso con trascendencia en el sector.

Dado que los documentos *post mortem* suelen dividirse en aquello que fue bien y aquello que fue mal, los autores toman su anterior trabajo como lo segundo y tratan de revisar los datos desde el punto de vista contrario [tabla figura 2 de ese artículo]. Como resultado, se visualiza que las buenas prácticas en videojuegos también vienen de las tradicionales. Todas las referidas al factor humano se anteponen a las demás, algo que ya defendía Bach en 1995 [referencia a Enough About Process…].

Tras probar la correlación entre la cantidad de buenas prácticas adoptadas y la ausencia de problemas durante el desarrollo, el documento muestra la que es sin duda la conclusión más interesante del estudio de la hipótesis. En ella se enumeran las buenas prácticas coincidentes en la muestra y su clara relación con las metodologías ágiles. Dicho de otro modo, los desarrolladores de la década pasada ya usaban el modelo ágil incluso cuando no lo adoptaban voluntariamente. Las prácticas ágiles estaban tan estandarizadas en el desarrollo convencional, que la industria del videojuego recurría instintivamente a ellas.

Para terminar, los autores rompen una lanza en favor de investigar materias del videojuego a través de modelos ágiles. Por ejemplo, en la incesante búsqueda del factor de diversión.

Antecedentes | Metodologías | Metodologías de diseño

Mientras que el desarrollo de videojuegos, por suerte o por desgracia, se ha valido de su simbiosis con la ingeniería del *software*, el diseño de juegos nació casi huérfano.

…

A diferencia de las metodologías de desarrollo, una metodología de diseño puede tener sólo uno entre diferentes puntos de aplicación posibles. Las metodologías de desarrollo son un cauce sobre el que corre el caudal que es el proyecto, pero no existe esa analogía en cuanto al diseño. Una metodología de diseño puede servir al diseño de mecánicas, al diseño de niveles, al diseño narrativo o incluso sólo al diseño de un género determinado. De hecho, como postulan varios estudiosos del diseño [referencias a Dan Cook, Chris Crawford y Keith Burgun], uno de los grandes errores en esta disciplina es intentar abarcar los videojuegos como un conjunto, algo que sí es evidentemente correcto en el caso del desarrollo.

Existen ciertas prácticas que, sin llegar a ser metodologías, han acabado asentándose entre los hábitos de todos los equipos, sirviendo de semilla a la fundamentación metodología de esta disciplina. Las más inmediatas son el documento de diseño y el análisis *post mortem*, que han sido comentadas a menudo [apartado metodologías de desarrollo] pero que se discutirán en profundidad a continuación. No obstante no son las únicas. En *Is agility out there? Agile practices in game development* [referencia] yerran los autores al mencionar que el documento de diseño es casi siempre la única documentación del desarrollo de videojuegos.

Documento conceptual

El documento conceptual [pie de página a término original *concept*] es en realidad otro artefacto básico. Lo comenzó a ser cuando se instauró el modelo de editoras y distribuidoras, donde este documento es el escaparate que decanta la financiación del proyecto. Acabó por solidificarse como una buena práctica, ya que como proceso exprime al equipo en una fase muy temprana de preproducción, y como hito sirve de riel sobre el que continuar la preproducción hasta desembocar en el documento de diseño. Actualmente la mayoría de desarrollos parten de un documento conceptual, independientemente de su uso para presentar la idea a terceros.

Buscar documentación formal del concept.

Páginas promedio, secciones.

Hablar de su reducción a panfleto.

Documento de diseño

Contrato. Relación contractual del creador de contenido y el diseñador, y del desarrollador consigo mismo.

Por completar.

Biblia narrativa y biblia artística

Por completar.

Diseño de niveles iterativo

Joel Burgess fue partícipe de los proyectos más preciados de Bethesda [referenciar los juegos], bajo las franquicias *Fallout* y *The Elder Scrolls*. Durante la GDC de 2014 realizó una ponencia sobre la metodología que habían seguido en ambos [referencia], mostrando con su ejemplo y referencias a terceros que en procesos de diseño también está asentado el modelo iterativo.

A fin de cuentas, el modelo iterativo funciona: en 1993 Jacob Nielsen [referencia] definió cómo la iteración permite al peso gráfico una más pronta usabilidad (a partir de involucrar a usuarios probeta en fases tempranas), y desde entonces nadie en esta industria le ha contrarrestado.

En diseño de juegos, el enfoque en iteraciones puede verse desde varias perspectivas:

* Carácter iterativo de franquicias continuistas: históricamente, esta ha sido la actitud adoptada por juegos deportivos, en cuyo modelo de negocio casi se requiere una iteración anual. Las metodologías de estos casos ponen el método de diseño por encima del plan de gestión del proyecto, y no al contrario como es habitual. En los últimos años algunas compañías punteras han calcado este modelo en juegos donde en principio no cabría hacerlo, como es el caso de Ubisoft con su franquicia *Assasin’s Creed*. A medio plazo, ha podido verse que este modelo acaba agotando el material del que se basa cuando su naturaleza no se acomoda al carácter iterativo.
* Revisión completa en gran iteración: más cercana al desarrollo convencional, esta perspectiva se da en proyectos —habitualmente independientes— que no tienen muchas expectativas pero que el público aúpa, hasta el punto de reiniciar el desarrollo con nuevos medios. Fue el caso, por ejemplo, del célebre *Spelunky* de Derek Yu [referencias al original y al revisado]. Generalmente este enfoque iterativo lo es *a posteriori*; es decir, sin que los desarrolladores lo planificasen desde el principio.
* Evolución progresiva de recursos individuales secundarios: debido a la finitud económica, es necesario que los estudios prioricen entre sus recursos. En ocasiones esta prioridad se da de forma que un recurso secundario tiene muy poco peso en el reparto al principio, pero va ganando recursos con el tiempo. El autor muestra el ejemplo de los personajes secundarios de *Team Fortress 2* [referencia], claramente mejorados al cabo de las actualizaciones.

Pero estas no son las iteraciones de las que se habla en el diseño iterativo, sino meras capas de gestión de proyectos que al final tienen más que ver con el desarrollo que con el diseño en sí.

En diseño iterativo de niveles, es inmediato asociar una iteración con un nivel. No es una vista a evitar, sino bastante adecuada. El problema es creer que dentro de esas iteraciones se dará siempre un avance secuencial. Es decir, que las iteraciones serán estructurales, contando con un plan, una construcción y un final. Por el contrario, la sempiterna peculiaridad del videojuego obliga a añadir un tipo de iteración distinto: la iteración cualitativa. En la iteración cualitativa, la construcción se divide en sí misma y en la verificación, dos fases que se alimentan cíclicamente durante un periodo de subiteraciones. La figura [meter figura numerada que es el conjunto de las dos figuras que tiene este señor en el artículo] representa esta diferencia.

El proceso con el cual llevar el diseño requiere una evolución natural y no forzada. El autor marca la creación del proceso de su equipo en *Fallout 3* [referencia], ya que fue a partir de tal proyecto cuando Bethesda dispuso de un equipo de diseño desde el principio. Además, no debe verse como un cambio brusco, sino como una forma de modelar y nivelar toda la habilidad acumulada y el conocimiento institucional. En su caso, el equipo estaba creándose tácitamente durante *Morrowind* y *Oblivion*, con procesos en pruebas.

Pese a ciertas corrientes en contra de la planificación del diseño, en el mercado hay innumerables evidencias de que evitar guiarse por fechas de vencimiento es como nadar a la deriva. Por ejemplo, el proceso iterativo de niveles evita errores de diseño de caja negra, que son no intuitivos y se detectan en fases muy tardías. La base para un proceso de diseño iterativo es la división del diseño en rebanadas [figura].

Una vez el diseño está fraccionado en lonchas, ya no hay por qué seguir con iteraciones continuas, en que cada nivel se crea de principio a fin hasta terminarlo y comenzar con el siguiente nivel. Se abre la opción a una “iteración por pasadas”. Apilando los niveles de manera horizontal en la figura anterior, la iteración por pasadas consistiría en desarrollar primero toda una columna y no una fila.

Una de las grandes cuestiones antes del modelo iterativo era: «¿Qué hace el diseñador cuando apenas hay código ni recursos gráficos?» Aunque aún suele darse que el diseñador sea un miembro orquesta, este nuevo enfoque solventa la pregunta y descongestiona los atascos clásicos:

* Mantiene frasco al diseñador, que se encuentra en un plan mucho menos fragmentado.
* Evita solapamientos y aumenta la coherencia y cohesión.
* Huye de cuellos de botella en la retroalimentación.
* Permite hacer frente al crecimiento paralelo del desarrollo.

El reparto de esfuerzos en la metodología es piramidal, de manera que un paso sirve de fuerte cimiento al siguiente. Cobra mucha importancia el ritmo [original timing] y el contexto de trabajo. Para ello, cada paso describe cuál será el estado del proyecto en él.

* Paso cero:
  + Planificación conceptual del nivel, a falta aún de recursos gráficos y de lógica.
  + Lista con espacios y eventos que el diseño del nivel pretende incluir.
  + Muy poca información; ni siquiera relaciones o dependencia, ni propósitos.
  + El diseñador esboza en un par de párrafos su intención con ese nivel.
  + Se mantiene rigurosamente el límite espacial para sintetizar la idea y no enturbiarla con demasiados focos distintos.
  + Ideas ingenuas sobre la historia relacionada con el nivel como espacio. Aunque no se transmitan directamente, ayudan a la narrativa ambiental y a tomar pequeñas decisiones aparentemente inconsecuentes.
  + Comienzan las peticiones de recursos gráficos únicos del nivel.
  + Se evitan los diseños con editores o recursos. Los artefactos de este paso son abstractos.
  + Se genera documentación de muy poca granularidad.
* Paso uno:
  + Comunicación con los diseñadores gráficos para recibir y entender equipos y recursos.
  + Creación de disposición del nivel, vacía y sin luz. Los recursos gráficos cambiarán —serán pulidos—, pero sus detalles funcionales como puntos de pivotado se mantendrán hasta el final.
  + Escritura incidental. Está asegurado que no será trabajo desaprovechado, porque en este punto el paso cero ya recibió retroalimentación y aprobación. Permite aliviar trabajo trivial para más adelante.
  + Gestión interna de sincronización con nombres, coordenadas mundiales, etc.
  + Se evita la optimización. La realimentación del paso uno será lo que asegura que los volúmenes son confiables, por lo que optimizar podría ser tiempo completamente desperdiciado.
  + Mismo caso con las mallas de navegación, así como con iluminación o desordenado, trucos gráficos [mesas que son capitales de estanterías], etc.
* Paso dos:
  + Centrado en jugabilidad.
  + El juego está muy avanzado ya en desarrollo.
  + Encuentros con enemigos, ritmo del nivel, botines, bienes ocultos, posición de ciertos objetos, jefes, etc.
  + Esbozo de escenas.
  + Mallas de navegación, cuya supervivencia a lo largo del proyecto ya está asegurada.
  + Aproximación a eventos como puzles, a nivel de posicionamiento o poco más profundo.
  + Optimización obligatoria, ya que no puede relegarse a los pasos siguientes.
  + Es primordial no dar pasos atrás y empezar de nuevo. Aunque parezca una obviedad, es un reflejo instintivo que deriva en pérdida de tiempo, no mejora y reitera lo ya iterado con el único fin, en realidad, de evitar el aburrimiento.
  + No programar comportamientos, porque la jugabilidad aún no está validada —lo estará en el siguiente paso—.
  + La retroalimentación comienza a ser mucho más frecuente. Aunque se componga de buenas sugerencias, se debe catalogar y no reaccionar a ella aún.
* Paso tres:
  + La meta principal es asegurarse de que no hay nada roto o ausente y, si lo hay, comprobar que está pendiente en la planificación.
  + Refinar la jugabilidad del paso anterior y añadir aquello que no fuera posible.
  + Enfrentar en este punto expectativas y realidad.
  + Realimentación por parejas.
  + Evitar optimización —no olvidar que en el paso anterior es preventiva—.
  + Tras este paso, el nivel debe ser jugable. Si los plazos se acortan, en una situación crítica podría tomarse como final.
  + Por ello, este tercer paso es el momento de admitir que algo no funciona, si procede. Se puede arreglar con reajuste, programar una reiteración o descartar.
  + En palabras del autor, «mejor elevar un buen nivel a la excelencia que uno malo a la mediocridad».
  + Uso de equipos en manada para levantar una característica que aún no haya recibido mucha atención, aprovechar al máximo y pulir el contenido de gran visibilidad, apoyar a un diseñador que afronta un gran problema, etc.
  + Compresión de calendario [crunch] como último recurso. Incluso para evitar cortes de contenido, el poco tiempo restante para la hora de la verdad trae riesgo de errores y baja productividad y creatividad. Es una buena herramienta si la jornada actual es reducida.
* Paso embellecedor:
  + Paso extra; ocurre cuando el equipo gráfico deja de producir nuevo contenido.
  + Permite al equipo de desarrollo asegurar que los problemas son de código.
  + El diseñador debe evitar estar libre a toda costa.
* Paso final:
  + El contenido se acerca a su liberación.
  + La única meta es pulir. Resaltar lo positivo de cada nivel y maquillar lo negativo.
  + Casi siempre hay asimilaciones tardías que en este punto aún es posible solventar, como combinaciones nuevas de mecánicas a muy pequeña escala (sinergias naturales, de creación espontánea).
  + Oleadas de realimentación desde el equipo, en pos de mejoras.
  + Terminar cualquier gestión interna restante, como las métricas.
  + No tiene por qué ser una única pasada. El refinamiento o el nuevo potencial pueden requerir más de una pasada de pulimento.

Como se ha visto, el objetivo de la metodología es mantener la visión de conjunto sobre el esfuerzo creativo. La iteración cualitativa, por tanto, llega en el momento oportuno, cuando se maximiza su uso. Incluirla en el principio de la metodología sería un error tan inocente como no contar con ella.

[figura con el resumen de la pirámide y si puede ser con una especie de diagrama cronológico, tipo agrupaciones musicales en Wikipedia]

Documento de diseño y demás.

*Post mortem.*

Open Letter Template [13],

which is composed by three sections. The \_rst section sum-

marizes the project and presents some important aspects of

the development. The next two sections, however, discuss

the most interesting aspects to the game developers

M. Myllyaho, O. Salo, J. Kriinen, J. Hyysalo, and

J. Koskela. A review of small and large post-mortem

analysis methods. In IEEE France, Paris, November

2004. 17th International Conference Software &

Systems Engineering and their Applications.

J. P. Flynt and O. Salem. Software Engineering for

Game Developers. Software Engineering Series. Course

Technology PTR, 1 ed. edition, November 2004.

Antecedentes | Narrativa

Disonancia ludonarrativa, cómo no.

Mitopoeia, diégesis, etc.

Nomenclátor, como analogía al *dramatis personae*.

Narrativa adaptada a flujo de eventos, no secuencial [referencias al de King Lucas, por ejemplo]