Metodologías Ágiles de Producción

Guillermo Jiménez Díaz (gjimenez@ucm.es)

Curso 2019-2020

Prefacio

Estos son los apuntes de la asignatura Metodologías Ágiles de Producción, impartida en la Facultad de Informática de la Universidad Complutense de Madrid por el profesor Guillermo Jiménez Díaz, del Departamento de Ingeniería del Software e Inteligencia Artificial.

Este material ha sido desarrollado a partir de distintas fuertes, destacando como referencia principal el libro *Agile Game Development with Scrum* de Clinton Keith, el curso *Software Processes and Agile Practices* de la Universidad de Alberta y los apuntes de asignaturas similares impartidas por Pedro Pablo Gómez Martín, Virginia Francisco Gilmartín y Gonzalo Méndez Pozo.

Metodologías de desarrollo de software

2.1. Conceptos básicos sobre desarrollo de software

El **ciclo de vida de un desarrollo de software** es el proceso que comprende todo el desarrollo de un producto, desde su concepción inicial hasta su "abandono".

Un **proceso** es la organización de un trabajo de desarrollo en distintas **fases o hitos** que hacen avanzar el proyecto hasta ser completado.

Cada fase de un proceso se compone de una serie de actividades. Una **actividad** es un conjunto de tareas relacionadas.

Una **tarea** es una unidad de trabajo que genera un "producto" tangible. Una tarea suele ser realizada por un **rol**, pertenece a una actividad, consume **recursos** y genera un **producto de trabajo**.

Las **prácticas** son reglas, guías, técnicas o herramientas para definir cómo desarrollar software o gestionar proyectos de manera efectiva. Algunas prácticas puede ser aplicadas solo a una fase concreta de un proceso mientras que otras pueden ser utilizadas a lo largo de todo el proceso.

Una **metodología** define un conjunto de prácticas para el desarrollo de software. Su objetivo es obtener un mejor rendimiento del equipo de trabajo y permite la obtención de mejores resultados.

Figura 2.1: Una definición gráfica de una tarea (Fuente: Notas del curso *Software Processes and Agile Practices*)

2.2. Modelos de procesos de desarrollo

Un modelo de proceso es una abstracción de un proceso que ha sido utilizado anteriormente con éxito. Todo proceso se divide en fases y los modelos definen cómo se secuencian estas fases. De acuerdo a esta secuenciación se pueden definir tres tipos de modelos:

- Modelos de proceso lineales: cada fase ocurre de manera lineal, una detrás de otra.
- Modelos de procesos iterativos: las fases son repetidas en ciclos.
- Modelos de procesos paralelos: las actividades relacionadas a las fases pueden realizarse simultáneamente en paralelo.

Los modelos de desarrollo de software, en general, definen las siguientes fases:

- Especificación de requisitos: es el punto de inicio, en el que se identifican las ideas o necesidades de los usuarios, se especifican y analizan y, generalmente, se priorizan.
- Diseño: en esta fase se especifica la arquitectura del sistema, de las bases de datos y de las interfaces de usuario, si fuesen necesarias.
- Implementación: es la fase en la que se realiza la programación del proyecto, creando código ejecutable, integrando las distintas funcionalidades en un mismo producto y documentando el código desarrollado.
- Verificación: Consiste en la realización de pruebas que aseguren que el producto desarrollado funciona adecuadamente. Implica no solo la creación y ejecución de pruebas de código sino también las evaluaciones con usuarios, las revisiones y auditorías y las demostraciones al cliente.
- Mantenimiento: Consiste en el cierre del producto y su lanzamiento, así como la

realización de los mantenimientos oportunos del software (ampliaciones, parches. . .) También incluye tareas de retrospectiva, que sirven para identificar las áreas que han de ser mejoradas en futuros proyectos y productos.

2.2.1. Modelos lineales

Son aquellos en los que cada fase se ejecuta de manera secuencial, sin capacidad para volver atrás. Se usan en desarrollos en los que no es necesario feedback del cliente y en el que los requisitos están claramente definidos y en los que no se esperan cambios.

2.2.1.1. El modelo en cascada

Es un proceso lineal en el que cada fase genera los productos necesarios para poder realizar la siguiente fase. Aunque inicialmente no prevé una vuelta atrás, existen variaciones que permiten retroceder a una fase anterior.

Figura 2.2: Metodología en cascada clásica

El proceso es sencillo y claro. Sin embargo, no se adapta a cambios ya que se basa en

conocer todos los requisitos del sistema a priori. El cliente no ve el resultado hasta el final del desarrollo, lo que puede implicar que no concuerde con lo que éste esperaba.

2.2.1.2. El modelo en "V"

Hace más hincapié en la validación del producto. Son dos ramas lineales en las que primero se realizan las fases de especificación y diseño (rama izquierda) para posteriormente realizar pruebas de validación a distintos niveles (rama derecha)

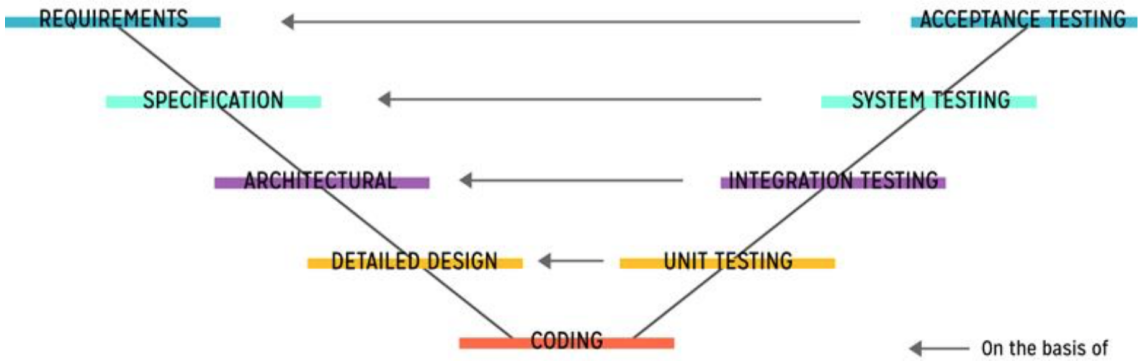

Figura 2.3: Modelo en "V"

Al igual que el anterior, no es flexible a cambios. Igualmente, el cliente no ve el producto hasta el final del desarrollo. Como ventaja, el equipo de desarrollo puede verificar el producto en cada fase.

2.2.1.3. El modelo de dientes de sierra

En este modelo, el cliente está envuelto en el proceso de desarrollo, ya que éste hace revisiones periódicas de prototipos del producto. El proceso sigue siendo lineal, en todo caso.

La principal ventaja de involucrar al cliente es que se espera que el producto final termine satisfaciendo las necesidades del usuario de una manera más exacta que con los otros modelos. Sin embargo, sigue teniendo los problemas de los procesos lineales: no es flexible a cambios en el producto.

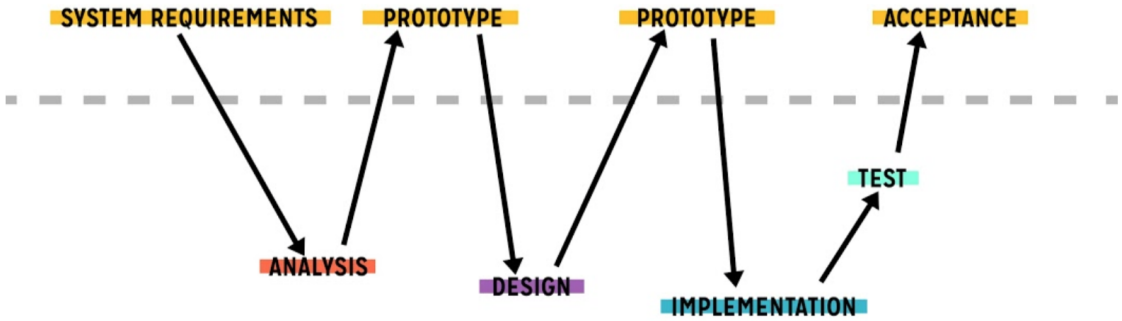

Figura 2.4: Modelo en dientes de sierra

2.2.2. Modelos iterativos

Están diseñados de modo que repiten varias veces cada una de las fases del desarrollo de software. Siguen una estructura iterativa o cíclica, de modo que se revisa cada una de las fases una y otra vez. Estos modelos suelen involucrar al cliente en cada una de las iteraciones.

2.2.2.1. Modelo en Espiral (Boehm, 1986)

Usa las mismas fases del modelo en cascada pero de manera cíclica. Además, incluye unas tareas específicas de identificación y resolución de riesgos y de prototipado y evaluación de dicho prototipo. El proceso se realiza una y otra vez hasta que el cliente da por bueno el producto generado tras terminar una iteración.

El proceso se compone de 4 fases:

- Objetivos: Se identifican los objetivos de la iteración, se exploran alternativas y se establecen restricciones.
- Riesgos: se identifican y valoran los riesgos de las alternativas propuestas en la fase anterior.
- Desarrollo y pruebas: se desarrolla un prototipo funcional y se evalúa de acuerdo a los riesgos propuestos en la fase anterior de esta iteración.
- Planificación: se planea el siguiente ciclo teniendo en cuenta los resultados de esta fase. Es una revisión que involucra a desarrolladores, clientes y usuarios

La evolución del proyecto se mide en base a las coordenadas polares del diagrama:

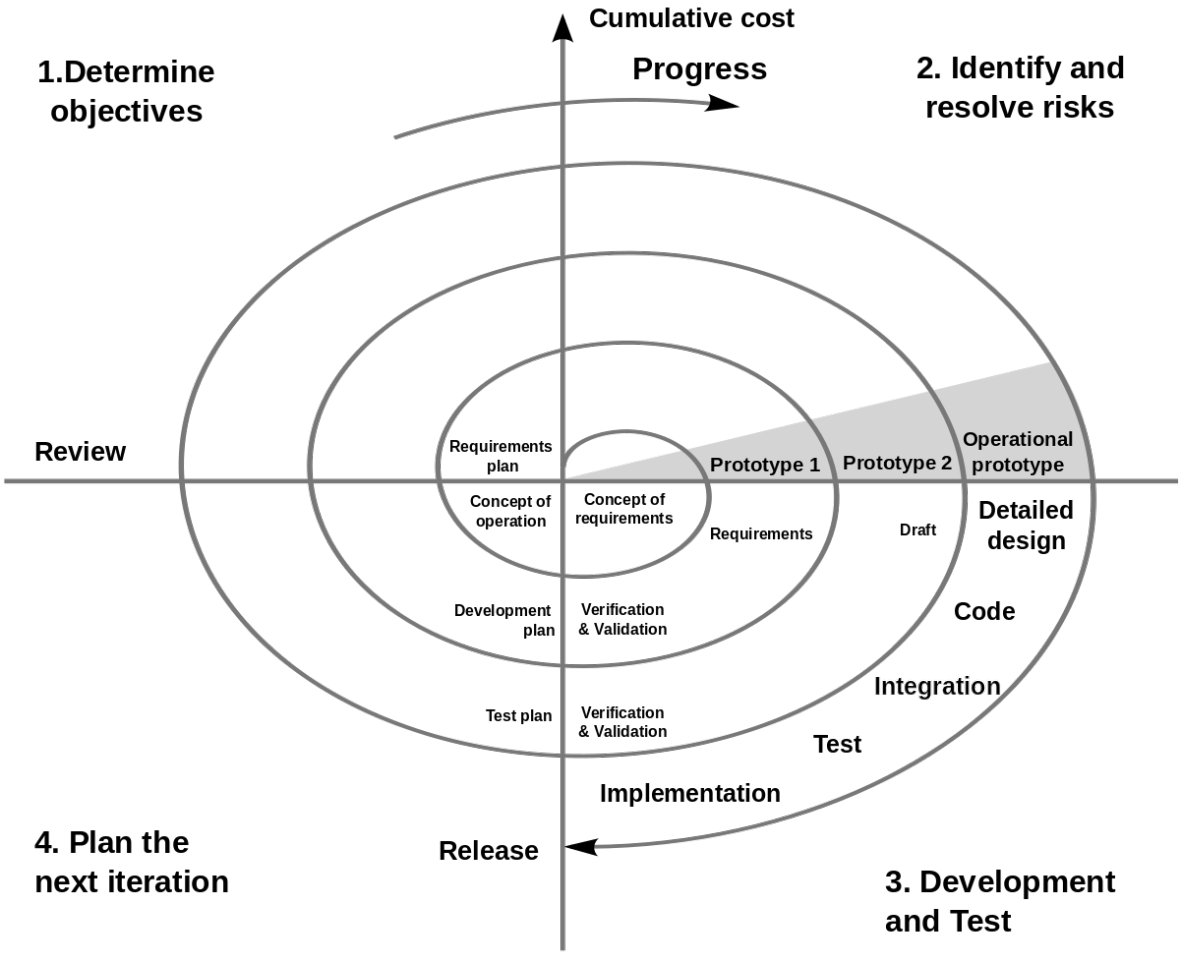

Figura 2.5: Modelo en Espiral (Fuente: Wikipedia)

- La coordenada angular mide el progreso en la iteración actual.
- La coordenada radial (distancia desde el origen) representa el coste acumulado del proyecto.

Presenta muchas mejoras con respecto a los procesos lineales, como la implicación del cliente en el proceso o el continuo refinamiento al que se somete el producto, lo que hace que sea más flexible a cambios. Sin embargo, tiene dos problemas fundamentales:

- Es necesario que el equipo de desarrollo tenga la experiencia suficiente para valorar adecuadamente los riesgos, ya que son fundamentales para el desarrollo de cada iteración.
- Al igual que los modelos lineales, presupone que conocemos a priori las especificaciones completas.
- Se desconoce a priori el número de iteraciones y cada ciclo no tiene una duración fija, por lo que puede ser difícil de estimar el coste de desarrollo.

2.2.3. Proceso unificado de desarrollo

Este modelo se caracteriza por ser un modelo iterativo e incremental, en el que cada iteración tiene un tamaño fijo que genera un *incremento* (una versión del producto mejorada con respecto a la fase anterior). Se considera un proceso paralelo ya que en cada iteración se incluyen actividades de cada una de las fases del proceso de desarrollo genérico (análisis, diseño, implementación...), aunque con distinta importancia a medida que se avanza en las fases del proyecto. Se centra en la arquitectura del sistema, de modo que no hay una única arquitectura sino múltiples vistas y modelos del sistema. Al igual que el modelo en espiral, se centra en la evaluación de riesgos, de modo que los mayores riesgos se afrontan cuanto antes.

Tiene 4 fases:

- Inicio (*inception*): Se define el ámbito general del proyecto, sus riesgos potenciales, se estiman los costes y la planificación y se establecen los escenarios de usos más básicos.
- Elaboración (*elaboration*): Se define la arquitectura del proyecto, creando los distintos modelos y vistas. También valida la arquitectura y establece el plan de implementación.
- Construcción (*construction*): Se implementan versiones funcionales del software que son probadas y evaluadas.

- Transición (*transition*): El software se pone en manos de clientes y usuarios con el fin de conseguir feedback o detectar errores.

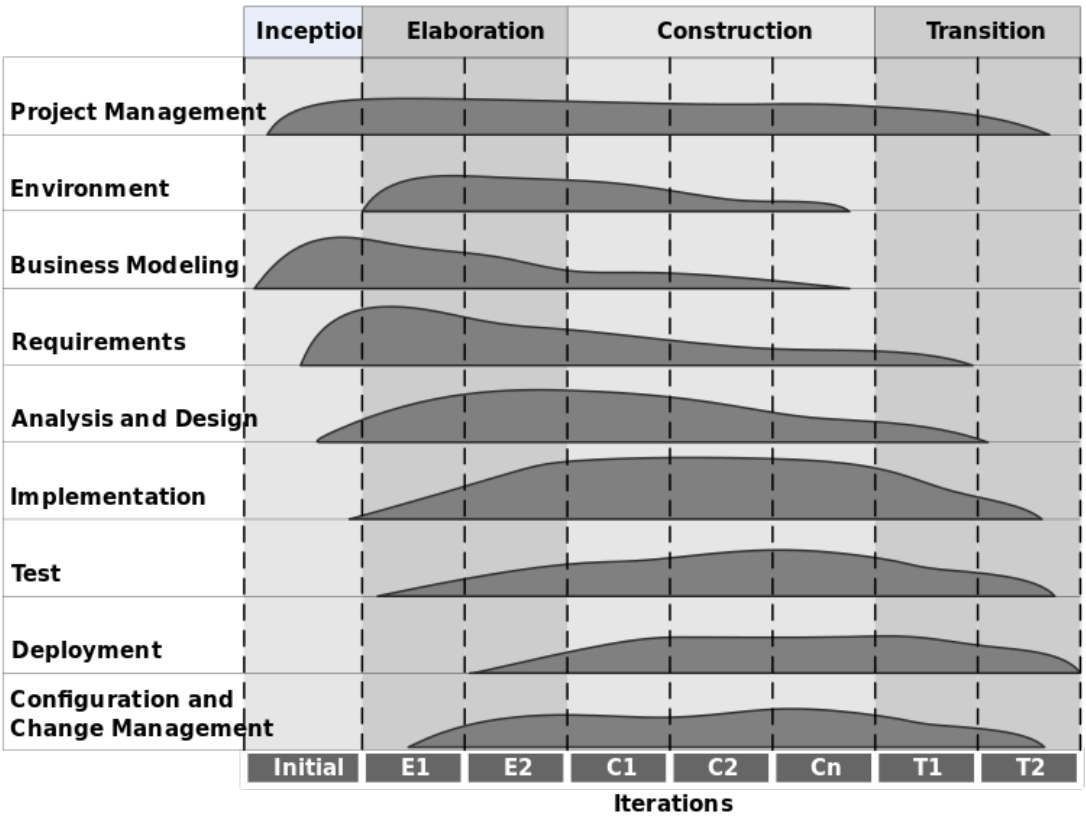

Figura 2.6: Proceso unificado (Fuente: Wikipedia)

Este modelo se adapta muy bien a grandes proyectos y tiene todas las ventajas de los procesos iterativos.

2.3. Metodologías tradicionales y desarrollo de videojuegos

Las metodologías tradicionales son metodologías que siguen un enfoque predictivo, que buscan imponer disciplina al proceso de desarrollo con el fin de volverlo predecible y

eficiente. En general, pueden seguir procesos lineales y sin marcha atrás, aunque hay metodologías tradicionales, como el Proceso Unificado Racional (*Rational Unified Process* o RUP, de IBM) que siguen modelos iterativos. En todo caso, la metodología define cuál es el resultado de cada una de las fases y/o iteraciones. Además, las metodologías tradicionales hacen hincapié en el desarrollo de una extensa documentación del proyecto.

Ejemplos: PMI, ITIL, Métrica 3, RUP.

Todas estas metodologías, en general, tienen una serie de problemas que hacen que no se adapten bien al proceso de desarrollo de videojuegos:

- Los clientes no saben lo que quieren hasta que no ven el software funcionando. En general, todas estas metodologías suelen mostrar el software funcionando en etapas muy tardías.
- Los diseñadores puede que no anticipen todas las dificultades a priori o pueden plantear arquitecturas que posteriormente no son válidas para nuevos requisitos, restricciones o problemas.
- Los errores encontrados en requisitos o diseño suelen ser difíciles de resolver.

Además, el desarrollo de videojuegos tiene sus propios problemas, los cuales hacen que las metodologías tradicionales no sean demasiado útiles para su desarrollo:

- *Feature creep*: En videojuegos, es muy común que aparezcan características adicionales sobre la marcha (nuevos requisitos impuestos por los publishers, cambios en características ya implementadas porque no generan jugabilidad entretenida). Estos cambios han de ser afrontados con el menor coste en tiempo y dinero posibles y hemos visto que, en general, las metodologías tradicionales no se adaptan bien a cambios. Necesitamos metodologías que nos ayuden no a alcanzar un objetivo concreto, sino a alcanzar el objetivo que surja de probar el juego hasta encontrar algo divertido (Figura 2.7).

- *Big Designs Up Front (BDUF)*: En general, las metodologías tradicionales proponen especificar completamente el software a desarrollar desde el primer momento. Esto implicaría construir un GDD que resuelva todas las preguntas de diseño desde el inicio, lo que no es posible ya que no podemos saber cómo irá el juego hasta que no lo probemos. De hecho, en general, comenzamos con una gran incertidumbre que poco a poco vamos reduciendo a medida que tenemos versiones jugables más completas del juego (Figura 2.8).

- Planificaciones optimistas: Planificar no es una ciencia exacta y depende de muchos factores, como las diferencias en experiencia y productividad de los miembros del equipo, el número de tareas en la que está involucrado cada miembro o la

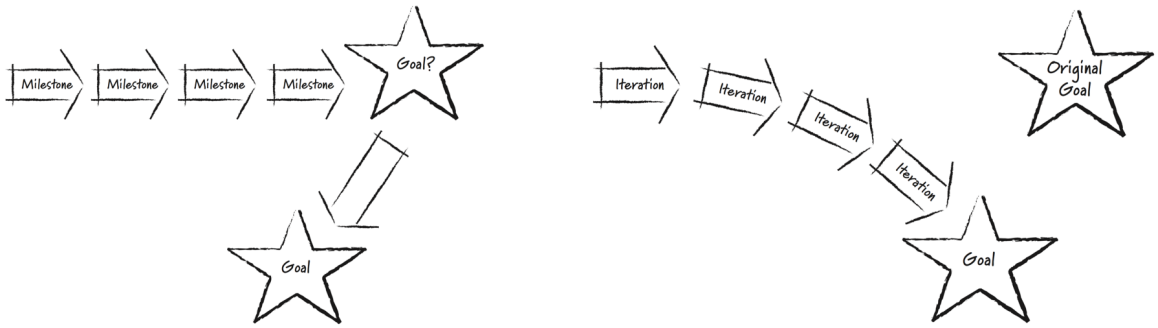

Figura 2.7: Alcance de objetivos: lo que conseguimos con las metodologías tradicionales frente a lo que esperamos conseguir (Fuente: *Agile Game Development with Scrum*)

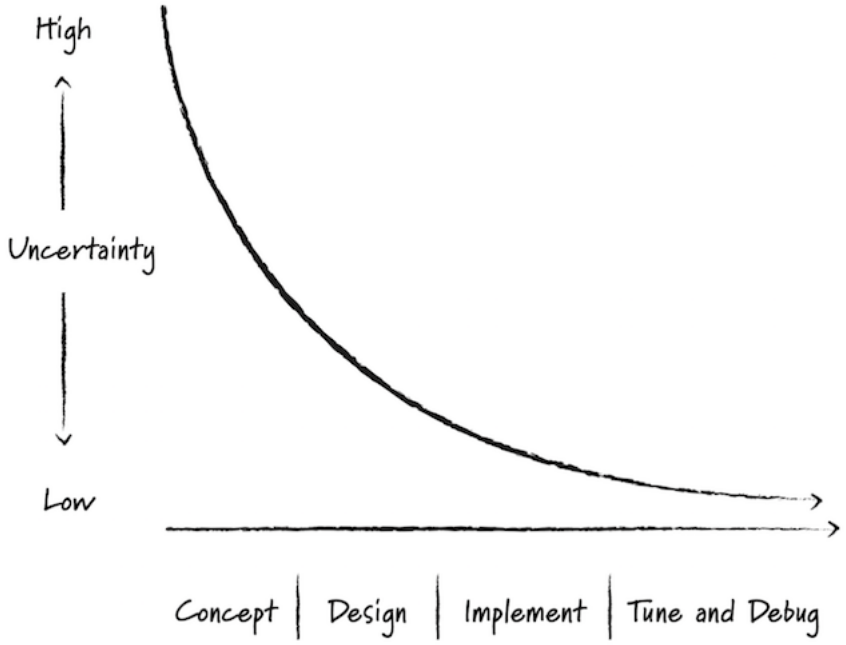

Figura 2.8: Reducción de la incertidumbre (Fuente: *Agile Game Development with Scrum*)

estabilidad del hardware y del software (¿cuántas versiones de Unity habéis visto desde que conocéis Unity?). En el desarrollo de videojuegos existe un handicap adicional y es que no sabemos cuánto tiempo nos llevará encontrar algo "que sea divertido".

- Preproducción y producción: la preproducción explora qué es el juego para definir los objetivos que se implementarán en producción, donde se crearán los contenidos intentando maximizar la eficiencia de los recursos y minimizando el coste a través de la *predictibilidad*. Si la preproducción es demasiado corta y se entra en producción demasiado pronto nos podemos encontrar con que hay demasiada incertidumbre, por lo que puede ser desastroso. La fase de producción debería comenzar cuando sabemos que:
  - Sabemos las mecánicas que vamos a implementar.
  - Sabemos que lo podemos desarrollar con las tecnología y las herramientas de las que disponemos.

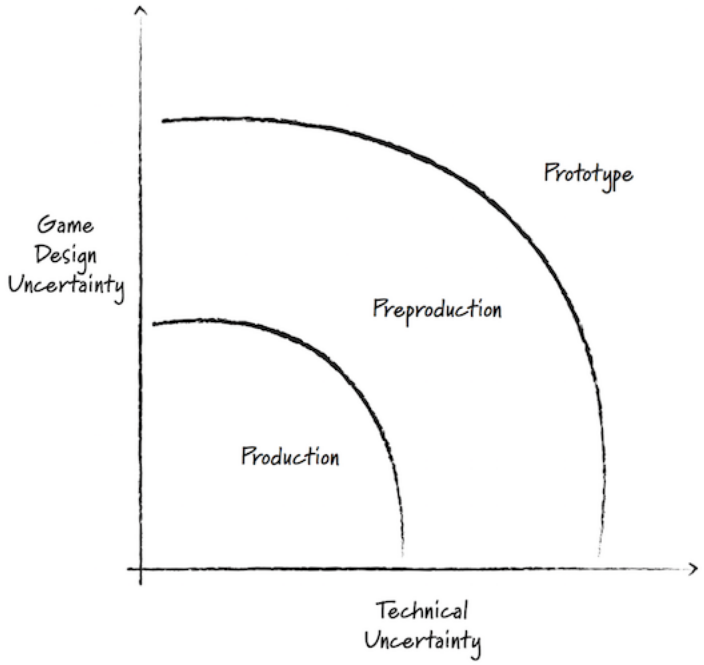

Figura 2.9: Relación entre la incertidumbre y las fases de desarrollo de videojuegos (Fuente: *Agile Game Development with Scrum*)

Si dedicamos demasiado tiempo a la preproducción entonces forzamos el proceso de

producción y desarrollo. Y si le dejamos poco tiempo puede ocurrir que durante la producción malgastemos el tiempo en el desarrollo de mecánicas y assets que posteriormente se tirarán. Por este motivo, necesitamos metodologías que permitan una convivencia temporal de ambas fases y susceptibles a cambios.

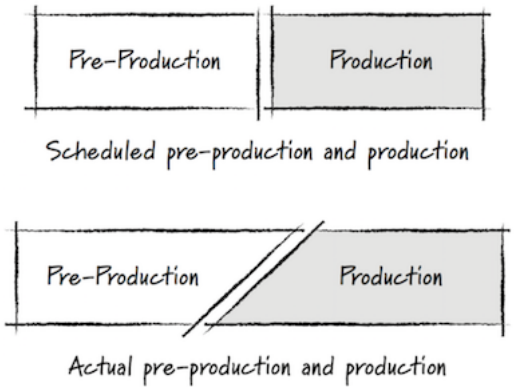

Figura 2.10: Organización de las fases de preproducción y producción (Fuente: *Agile Game Development with Scrum*)

Nota final