Scrum Quest

Este documento trata de explicar la metodología Scrum a través de una historia ficticia. Los personajes y las situaciones no son reales y cualquier parecido con la realidad es pura coincidencia.

El equipo

John Lisp

Fanático de la IA. El único que se supone que sabe algo de Scrum e intenta adoctrinar al resto. Moderado en sus posiciones. Adicto a la cafeína.

Peter Class

Fundamentalista ultraortodoxo fanático de Java, discípulo radical de los Three Amigos y del Gang of Four. Verbalmente inofensivo si se le aleja de los ordenadores y de cualquier conversación que gire en torno a la programación. Adicto a la cafeína.

Float

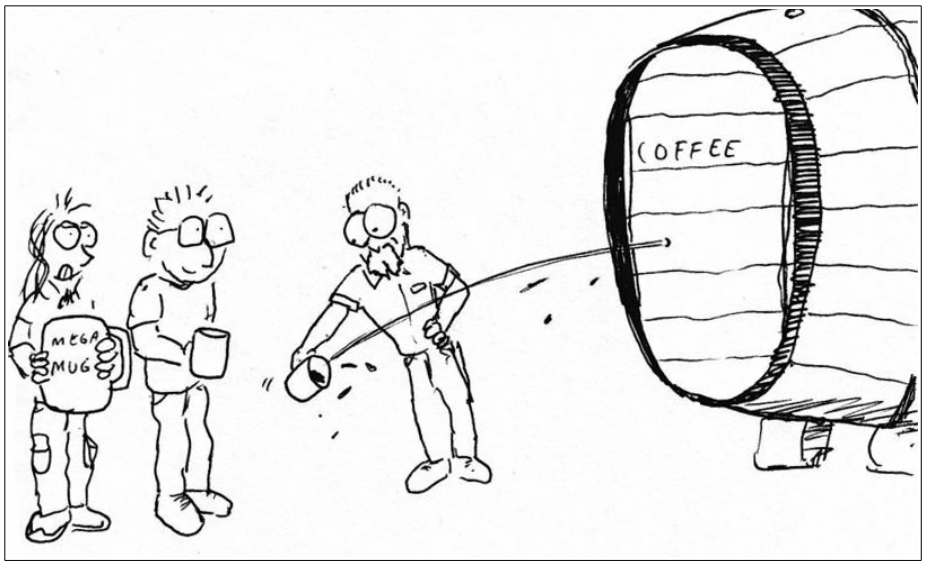
Programador bajeril, fanático de perl y cualquier lenguaje que te permita hacer mucho con cuatro líneas. Su Kung Foo es muy bueno. Adicto.

Simon Dice

El cliente: heavy, soltero, rolero y hetero. En el pasado fue programador de ensamblador. Envuelve todo en fundas y bolsitas protectoras de plástico pero su cuarto está hecho una mierda.

Contenidos

1. El marrón



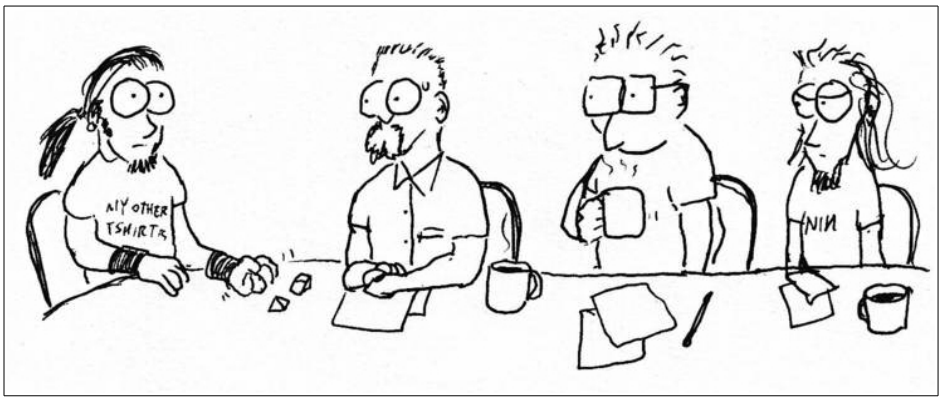
Amanecía un nuevo día en la oficinas de Bubble Telecom, un proveedor de servicios de internet de medio pelo donde trabajaban nuestros héroes: los programadores. Vírgenes por delante, mártires por detrás, los miembros del noble gremio de los picateclas arrastran la maldición de mezclar su hobby con su trabajo. Y por eso mismo con resignación pisaban el suelo técnico enmoquetado en dirección a la máquina de café para chutarse la primera dosis de estimulantes de la mañana y para intercambiar los rumores de lo que iba a deparar el día.

-Temblad chavales, ayer antes de salir el comercial me pilló por banda y me dijo que tenía que hablar con nosotros – comentó Peter con preocupación.
-Pero ¿no tenía vetado entrar en nuestra cueva? - dijo John sorprendido.
-Debe ser que con las últimas ventas ha derribado el escudo protector del departamento – dijo Float.- Abandonad toda esperanza, el fin se acerca. Veo una tormeta de mierda que se cierne sobre nosotros. Corred insensatos.
-Me temo que nos van a meter otro proyecto pero joder, que esto no da más de si- protestó Peter.
-Los comerciales ya se sabe, con saliva y con cariño todo entra – dijo John – Prepárense para ser sodomizados, la resistencia es fútil.
-¡Hombre, si están aquí los campeones de la programación! Tengo un proyecto para vosotros que os va a encantar.

Así fue como el comercial profanó la sagrada rutina del primer café y se llevó a los programadores de su santuario con la bebida a medio terminar camino de la cámara de torturas: la sala de reuniones, ese lugar donde jefes y compañeros de otros departamentos ignotos que no molan nada te piden más trabajo mientras te restan tiempo para hacerlo; vamos que la barra de vitalidad te baja a la mitad como si hubieras recibido tres haddouken seguidos.

En esta ocasión en la sala no había la típica pareja de personas trajeadas con carpetas extrañas y logos exóticos. Solo había un pavo con el pelo largo recogido en una coleta y perilla, y con una camiseta gris oscura en la que ponía: mi otra camiseta está limpia.

Llevaba muñequeras de cuero y estaba distraido lanzando dados de colores de geometrías no-euclidianas.



-Chicos, os presento a Simon. No entiendo nada de lo que necesita así que os lo cuente a vosotros. Yo pasaré a cobrar mi comisión de todas formas. ¿Os he dicho que el trabajo tiene que estar para ayer? Jajaja, es broma. No, ahora en serio. Es una cosa que le urge. Os dejo, que tengo una reunión con los de sistemas para exigirles que me mantengan el Outlook 97 en el windows 8.

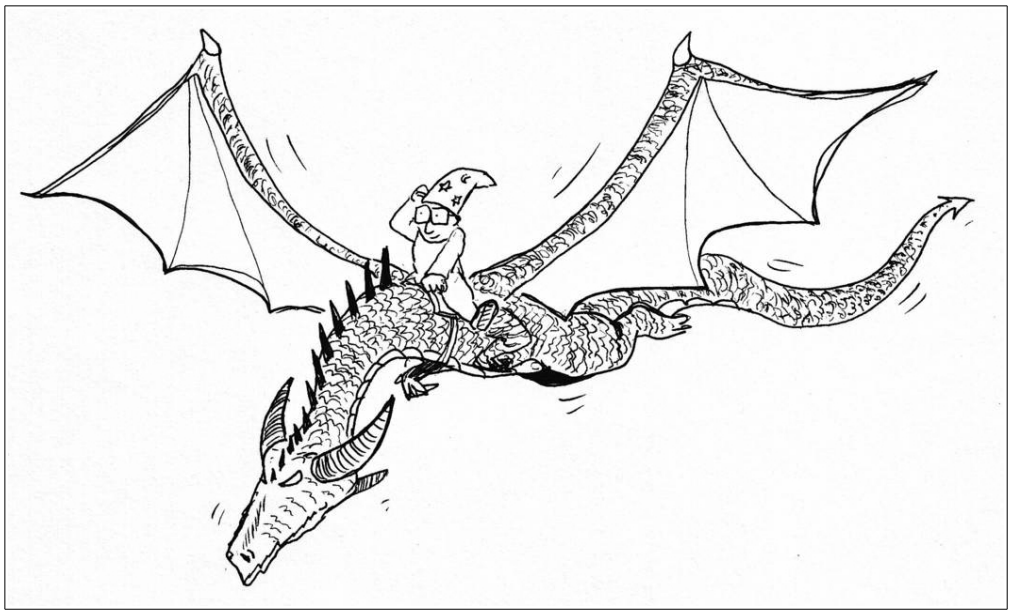
En cuanto el comercial cerró la puerta de la sala, Float susurró:
-A mi señal, ira y fuego.
-Esto... tú diras – dijo John.
Y el cliente comenzó:
-Pues veréis...
¿Vamos a llevarnos comisión? - interrumpió Float . Se hizo un silencio. Se oyó un grillo. - Es broma.
-Esto no sé. Veréis, soy el MetaMaster de la asociación de rol El cuesco del Orco.
-Será el cuerno – corrigió Peter
-No, lo llamamos el cuesco porque sonaba más … grosero.
-Sí es más orcish – apuntó John ajustándose las gafas.
-Y lo de MetaMaster ¿significa que eres un Master de Masters? - preguntó Float.
-Algo así. Me hicieron MetaMaster porque en todas mis partidas la peña muere de forma irremediable en una agonía infinita. Me trago sus almas pero escupo sus camisetas de Children Of Bodom. - Se hizo un silencio. Se oyó una tos lejana. - Es broma.

-Bueno, a lo que iba, hemos juntado pasta y queremos pediros que nos hagáis un juego de mazmorras pero para ordenador.

-Oh cielos, que idea tan innovadora! - exclamó Float.
-Y además queremos que sea para consola, de momento.
-Ya me veo cargado collares y anillos de oro y con mi cabeza hundida en una montaña de farla. Primo, de esta nos retiramos.
-Eso sí, hay que sacarlo ya porque tenemos a mucha gente esperando para probarlo.
-Pero a ver, a ver... - dijo Peter pasándose la mano por la frente, en su típico gesto de “este no sabe ni lo que quiere” y “marrónazo” - ¿Qué tiene que ofrecer ese juego?
-Básicamente el jugador tiene que ir de una mazmorra a la otra hasta encontrar una que contiene la salida- respondió Simon.
-Pero a ver, ¿cómo pasa de una mazmorra a la otra? ¿y cuántas mazmorras hay?
-Yo que sé, pues puede ser un laberinto de 10x10 y en cada mazmorra podría haber 4

puertas, norte sur este oeste. ¿Cuánto tardaréis?
-Insuficient data for meaningful answer – espetó Float
-No le hagas caso, ya tenemos por dónde empezar. Unas pocas frases que resumen el proyecto. Y te tenemos a ti, Simon – Dijo John.
-¿A mí?
-Sí, eres el usuario/cliente y debes formar parte del equipo. Bienvenido a un proyecto Scrum.

2. Evangelizando infieles



-Espera espera – dijo Peter - ¿cómo que Scrum? Esta no es la forma de hacer las cosas que tenemos aquí. Primero debemos completar un documento de toma de requerimientos que debe firmar el cliente, el comercial y el jefe de proyectos. Luego debe tener lugar una reunión con el analista para que este elabore un borrador de análisis. ¿De qué me estás hablando?

-Hablo de que si pretendemos afrontar este brown de forma convencional estamos condenados a repetir los errores del pasado – contesto John. - Fijaros en las condiciones: no tenemos mucho tiempo, el cliente solo tiene una vaga idea de lo que quiere y es probable que nos vaya introduciendo cambios sobre la marcha.

-Yo digo que nos pongamos a programar pero ya, de hecho ya estoy elaborando un prototipo – saltó Float mientras tecleaba frenéticamente.

Peter se llevó las manos a la cabeza y se tiraba de los pelos:

-¡Eso nos llevará al caos! ¿En qué te basas para estar programando? ¿Tienes algún diseño de clases dibujado en algún documento compartido? ¿Y dónde está tu código? Un proyecto debe ser predecible y ante todo controlable, ¿o es que quieres volver a la época tenebrosa? Décadas de ingeniería del software, de patrones, de refinamiento de metodologías...

-...con las que me limpio el culo.- contestó Float- Esto de las metodologías no son más que excusas para que existan castas de analistas, jefes de proyecto, gerentes y demás fauna que lo único que quieren es no programar. Eso sí, cuando pasan meses y por fin se presenta algo al cliente, si algo sale mal la culpa la tendremos nosotros. ¿O no John? ¿tú que dices?

-Los dos tenéis parte de razón, por eso con metodologías ágiles como Scrum se trata de conciliar esas dos palabras que ha mencionado Peter: Caos y Control.

-Pero tengo entendido que si seguimos esa metodología las reuniones son mínimas, se reparten tareas y se programa desde el principio – dijo Float.

-Perdónales señor porque no saben lo que dicen. Estos herejes adoran la programación extrema como si fuera un ídolo de oro pagano- Contestó Peter - Si empiezas programando a saco sin planificación, sin preveer nada os va a salir una chapuza. Una chapuza que en cuanto tengáis que cambiar una línea de código va a petar por todas partes.

-Ahí es donde entran en juego las pruebas unitarias. Antes incluso que programar las clases programaremos las pruebas unitarias y desde el principio podremos testear de forma automática cada método de cada clase. Luego conforme vayamos cambiando el código no tendremos más que volver a pasar los test para asegurarnos que todo funciona igual.

-Joder,¡pero eso es programar el doble! - protestó Float.

-¿No te gustaba programar? - rio John – las pruebas unitarias son el control de calidad de tu software. Además te obligan a programar mucho mejor: haciendo que cada clase y cada uno de sus métodos hagan una cosa concreta para facilitar su testeo nos lleva a un código menos acoplado y mucho más cohesionado. Es decir, las clases no se pisan unas a otras sino que colaboran entre ellas.

-Pero ¿tú sabes lo lento que va a ser eso? - se desesperaba Peter - ¿y cada vez que quiera ejecutar algo tendré que compilar más cosas, pasar los test, etc ? Vamos a pasar más tiempo pulsando botones que escribiendo código.

-Y ahí es donde sacamos del arsenal herramientas de integración contínua como Ant. Basta con configurar un script para poder automatizar todas esas tediosas tareas: crear carpetas, compilar, pasar tests, ejecutar, etc... con un solo click o un solo comando podremos hacer todo eso de forma mucho más ágil.

-Pero por mucho que hagamos pruebas, vamos a estar todo el día retocando todo y eso como poco nos obligará a reescribir mucho código. Y además a ver quién entiende un código que está venga cambiar – dijo Peter.

-No vamos a reescribir – contestó John- vamos a refactorizar.

-¡Buah! Ya salió el cursi, que le pone un nombre molón a algo normal que llevamos haciendo desde siempre – saltó Float – Déjame que te diga,vas de cool y das por cul[1].

John Lisp levantó un dedo para enfatizar sus siguientes palabras.

-Refactorizar no es sinónimo de reescribir sin más. Se trata de retocar el código de tal forma que sea más legible para las personas. Se trata de dejarlo tan claro que no sean necesarios los comentarios en el código. Muchas refactorizaciones son well-known y para vuestra tranquilidad, los IDEs modernos tienen herramientas que te permiten refactorizar de forma automática, segura y fiable.

-Mucho sabes tú, apuesto a que aspiras a jefe de proyectos – dijo Float.

John se volvió a ajustar las gafas en un gesto de triunfo.

-No no... el orden jerárquico cambiará bastante. De hecho habrá orden pero no jerarquía. Vamos a repartirnos los roles y veréis que esto es una cosa más bien horizontal.

1 http://www.youtube.com/watch?v=UkY5JGt65cg

3. El reparto de roles

-Como os podéis imaginar, yo seré el ScrumMaster y vosotros el equipo– comenzó John.

-Lo sabía – dijo Peter.

-En el google honest translate tu frase se traduce así: “yo me tocaré los cojones y vosotros trabajaréis para mí” - añadió Float.

-Yo seré el ScrumMaster, porque no tenéis ni puñetera idea de Scrum – explicó John – El ScrumMaster no es un jefe, sino el garante de que el equipo sigue las reglas de Scrum. El ScrumMaster lidera la aplicación de Scrum y a la vez sirve al equipo. El ScrumMaster actúa como facilitador, desatascador y optimizador para el equipo.

-¿Entonces no vas a ser el que manda? - los esquemas de Peter se derrumbaban.

-Para nada. Una de las claves de Scrum es que los equipos se autogestionan y todo se hace por consenso. Yo seré el ScrumMaster más que nada porque ahora mismo soy el que más experiencia tiene con Scrum y soy el que más puede ayudar para que lo asimiléis. Pero eso no me impide formar parte del equipo, así que tranquilos que también voy a picar teclas.

-¿Y qué va a pintar Simon en todo esto? ¿De verdad va a trabajar con nosotros? - preguntó Float.

-El papel de Simon va a ser fundamental: será el PorductOwner, es decir, el propietario del producto. Básicamente es el que va a gestionar el ProductBacklog.

-¿Mandé?

-Luego explicaré mejor eso del ProductBacklog. Dicho rápido y mal es la fuente inicial de donde surgirán todas las tareas concretas. Es un repositorio de las funcionalidades del proyecto y el ProductOwner es el encargado de mantener y sobre todo priorizarlo.

Peter se rascaba la perilla y preguntó:

-¿Son como los casos de uso?

-Pueden verse así se te gusta. Hay quién las llama User Stories, historias del usuario. Son frases que resumen propiedades del producto, del proyecto de software en nuestro caso.

-umm ya veo. Al final, va a ser el cliente va a ser el que manda. - dijo Float en tono suspicaz.

-No no, el ProductOwner (que no tiene porqué ser un cliente, puede ser un usuario o un gerente de producto) fijará las prioridades y eso sí, debemos respetar sus decisiones. Es quién se comunica con el exterior para recabar requerimientos, y en cierto modo hace de muro de contención frente a los “StakeHolders”. También tiene otros poderes como veremos más adelante; por ejemplo una vez estemos en pleno fregao el ProductOwner podría cancelar un trabajo, y eso no lo puede impedir ni el ScrumMaster.

-Hablando en plata, el ProductOwner es un tocapelotas – añadió Float.

-Tampoco, de hecho también puede formar parte del equipo de desarrollo. En las metodologías ágiles todo el mundo se involucra o está metido en el ajo. Y su papel es fundamental para que haya transparencia manteniendo a la vista de todos el ProductBacklog. Eso sí, el ProductOwner no puede ser ScrumMaster.

-¿Tu sabes programar Simon? - preguntó Peter.

Simon, con la mirada perdida suspiró:

-Bueno, en su día programé un Arkanoid en ensamblador para intel 286. He visto cosas que no creeriais. Alterar el vector de interrupciones para colocar mis propias subrutinas.Dejar un programa residente en memoria de 4 maneras distintas. Invocar todas las funciones de la lista de Ralph Brown. Crear música con el Pc-Speaker. Todos esos momentos se perderán en el tiempo como lagrimas en la lluvia.

-Si has programado en ensamblador, tú puedes con todo. - dijo John.

-No sé si podrá con con el rigor y la ortodoxia de Peter; si hacen PairProgramming va a haber muertes. - susurró Float al oido de John. - ¿A todo esto, cuando empezamos a trabajar?

-Sí, supongo que al menos en toda esta vaina ¿habrá una serie de pasos a seguir no? - dijo Peter – necesito algún asidero más o menos metódico, un ciclo de desarrollo concreto.

-No será un ciclo, sino muchos pequeños ciclos de entregas – contestó John - Y todos ellos se irán revisando para adaptarlos, afinarlos y mejorarlos. Os expongo los pasos y los ponemos en práctica.

Further reading:

- Roman Pilcher: Agile Product Management with Scrum. Cap 6: Transitioning into the product, the Owner role

4. Scrum en un folio

-En un ciclo Scrum hay unos bloques de tiempo en los que se mantienen reuniones y se producen una serie de artefactos, todos ellos siguiendo unas reglas – Comenzó John.

-Sí John, ha quedado claro que eres el que sabe de Scrum, pero ¿nos lo podrías traducir a nuestro idioma? - dijo Float.

-Lo siento pero mi Klingon es bastante básico.

-Yo tengo un B2 en Klingon y un A2 en Romuliano, lo digo por si puedo ayudar en algo para que nos entendamos – intervino Simon.

-A veces me preguntó de dónde sacarán los clientes estos comerciales – comentó Peter para si mismo.

-Mmmh, pues este no parece salido de un bar de carretera – se extrañó John.

-Nos encontramos en una sauna- dijo Simon encogiéndose de hombros.

-Eso te pasa por preguntar- cortó Float - Yo no quiero saber más pero dinos qué pasos tiene un ciclo Scrum anda.

-Bien, pues todo parte de las historias del usuario con las que el Product Owner genera el Product Backlog ¿qué es eso? Es una lista de todo lo que es necesario para conseguir el producto, siguiendo un orden prioritario. Una vez tenemos esa lista de historias se prepará el primer ciclo de desarrollo donde se generará una primera entrega.

4.1. Release Plan

En esta fase se planifica qué contendrá la entrega. Esto se puede formalizar opcionalmente llevando a cabo una primera reunión,

1. **La reunión de planificación de la entrega**

En ella se trata de establecer un objetivo para la entrega seleccionando los elementos del Product Backlog de mayor prioridad. Es decir, se establece hasta dónde se pretende llegar. Tras esto nos metemos de lleno en:

4.2. El Sprint

Un sprint es una iteración de desarrollo que se divide en los siguientes bloques de tiempo:

1. **Sprint Planning Meeting**, o reunión de planificación del Sprint: partiendo del Product Backlog presentado por el Product Owner el equipo decide qué elementos del mismo, qué funciones va a cumplir en el próximo Sprint. Para tomar esas decisiones se basa en su propia capacidad, su experiencia, el resultado del Sprint anterior, etc... Por otro lado, el equipo debe diseñar una serie de tareas encaminadas a conseguir esas funciones. En cierto modo el objetivo se descompone el pequeñas tareas que alimentan la lista del Sprint Backlog. El equipo se auto-organiza y se asigna esas tareas.
2. Desarrollo + **Daily Scrum Meeting** o Scrum diario: una vez repartidas las tareas, cada desarrollador se pone a programar a su bola. Eso sí, para que haya transparencia, a diario se debe celebrar una breve reunión de 15m en la que cada miembro del equipo debe contar:
    - Los progresos desde la última reunión
    - Los objetivos marcados hasta la próxima reunión
    - Los problemas que se está encontrando
3. **Sprint Review** o revisión del Sprint: se presentan las funcionalidades conseguidas y el equipo analiza qué se ha hecho, qué problemas se encontraron, cómo se resolvieron, etc.. y el Product Owner identifica qué se ha hecho y qué no se ha hecho, y analiza el estado en el que queda el Product Backlog. Es una revisión fundamental de cara a preparar el siguiente ciclo.
4. **Sprint Retrospective**, o retrospectiva del Sprint: el ScrumMaster debe conseguir que entre todos se revise el proceso de aplicación de Scrum, identificar lo que se ha hecho bien y las posibles áreas de mejora de cara al próximo Sprint.

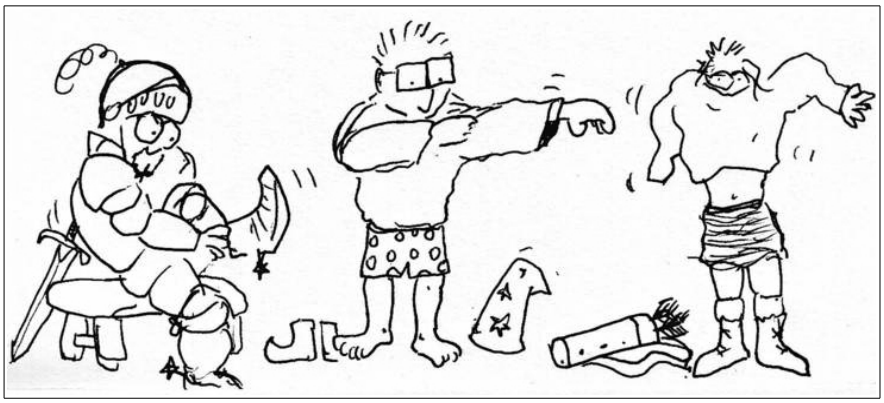
-Me abuuurro... - se quejó Float - podemos programar ya?

-Venga vamos a por el primer ciclo – dijo John.

Further reading:

- Jorge Serrano: Explicando Scrum a mi abuela
- Ken Schwaber & Jeff Sutherland: Scrum guide.
- Kim Pries, Jon Quigley: Scrum Project Management. Cap2: Scrum Basics
- http://eugeniaperez.es/wordpress/?p=246

5. La primera vuelta



Peter se puso nervioso:

-Espera espera, para el carro. ¿Cómo vamos a empezar si apenas tenemos una vaga idea de lo que quiere el amigo Simon?

-Cierto, vamos a alimentar un poco el Product Backlog – dijo John – Simon, es hora de que hagas la carta a los reyes magos. Pídenos esas cosas alucinantes que tendrá el producto que tienes en mente.

-¿Puedo pedir lo que quiera?

-Tomaremos nota y crearemos una lista. En proyectos grandes tu serías una especie de portavoz, filtro, de todos los implicados o StakeHolders con el producto: usuarios, gerentes, etc... En este caso tú lo englobas todo, así que es más simple. Ahora tú debes asignar las prioridades

-Pues, básicamente quiero un Dungeon Crawler, ya sabéis, un juego en el que un personaje se mueve de una mazmorra a otra.

-¿Qué tipo de personaje? ¿Un ninja? ¿Y los enemigos qué son? ¿Zombies? Dime que sí, siempre he querido mezclar los dos géneros – suplicaba Float.

-No exactamente. Hombre, sería importante que el jugador puediera ser de distinto tipo, ya sabéis lo típico: bárbaro, elfo, enano y mago, cada uno con características distintas.

-¿Pero va a haber tortas o no? - preguntó John.

-Tortas y tesoros también, esa es la idea, matar y robar. - respondió Simon.

-Vaya qué educativo ¿y por qué no metes bien de drogas, ya que estamos? - Peter estaba escandalizado.

-Claro, bueno, yo las llamo pociones. Lo típico es que el jugador se pueda encontrar cosas. Tesoros, armas y por supuesto cosas mágicas que alteran sus capacidades.

A Float solo le importaba una cosa

-¿Pero va a haber zombies o no?

-Sin duda. Zombies everywhere. En la mazmorra pueden salir toda clase de enemigos: zombies, esqueletos, infectados, no-muertos, espectros, fantasmas,...

-¿Pero todos esos no son la misma cosa? - preguntó Peter con cierta malicia.

John intervino rápido:

-Como Scrum Master os prohibo debatir el tema de Zombies e infectados. La última vez hubo más hostias que cuando se discutió si usar los setter y getter en la propia clase. You shall not pass.

Vamos al lío, ya tenemos bastantes cosillas. ¿Se te ocurre alguna más Simon?

-Pues... no. De momento.

-No importa. En las metodologías ágiles se preveen ese tipo de escenarios. De hecho por eso este tipo de metodologías se ajusta tan bien a proyectos vivos: páginas web, aplicaciones para móviles, todas ellas tienen usuarios que te demandan cambios y mejoras casi en tiempo real. Si surgen más cosas las añadiremos más adelante, el Product Backlog no es una lista escrita a fuego sobre piedra, es algo vivo (no quiero usar la palábra dinámico). Con lo que nos has dicho tendriamos este Product Backlog, a ver qué te parece. Simon dice:

- Crear un Dungeon Crawler bidimensional en el que el jugador debe buscar una salida.
- El jugador puede ser de distinta clase, cada una con sus características
- En cada mazmorra el jugador puede encontrarse enemigos contra los que luchar
- Los enemigos pueden ser de distinta clase, cada uno con sus características.
- En cada mazmorra el jugador puede encontrarse objetos
- Las mazmorras tienen puertas, algunas de ellas pueden estar cerradas y la llave puede estar perdida en otra mazmorra (esta la he deducido yo Simon)

-Pero esto es un mundo ideal – se quejaba Float – qué pasa con la preparación del entorno, qué pasa con la tarea de investigación y sondeo de librerías y técnicas, etc.. ¿toda esa labor sorda no se refleja?

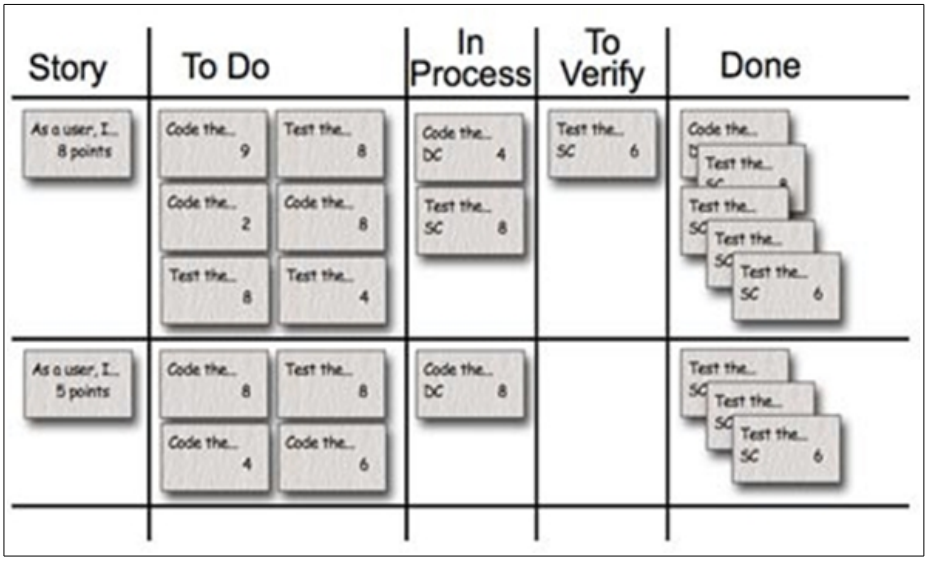
--Se puede hacer, efectivamente se pueden incluir esas tareas. todo eso depende del nivel de complejidad. Incluso si queremos distinguir las historias podemos asociarles una especie de etiqueta para agruparlas que puede ser un tema concreto -Theme-, Problems, Features. Bien- dijo John- Ahora si queréis podemos usar una pizarra convencional y poner cada una de estas historias a modo de postit en la sección ProductBacklog. Si las usamos en un Sprint desglosaremos las historias en Tasks o tareas y las podremos en el SprintBacklog y luego ahí podremos ir jugando con las tareas.

-Ah esto de los postits que se ve tanto... no śe – desconfiaba Peter – ¿nos vamos a fiar de que nadie haga el bobo con los papelitos?

-En las metodologías ágiles se insiste mucho en simplificar, en no complicarse la vida con herramientas ni perder el tiempo usando software farragoso para gestión de proyectos. De todas formas, si os da más confianza podemos y usar Kunagi. Es una aplicación web para tomcat que se instalá y se pone en marcha en cero coma.

Pizarras Scrum

Se puede montar con pizarras blancas y con postits. Sin complicarse la vida. Las secciones que pueden tener son estas:



Otra opción es adquirir Scrum Boards más profesionales, que pueden tener este aspecto:

El product backlog en Kunagi

Este sería el aspecto que tendría el product backlog en Kunagi. La aplicación es suficientemente sencilla y fácil de instalar y puede ser una alternativa en caso de no quere usar papeles o tener el equipo disperso. Vemos la lista de historias con una de ellas

desplegada donde se puede apreciar el detalle.

5.1. El primer Release Plan

-Bueno, vamos a tirar a la piscina y hacemos nuestro primer Release Plan ¿vale?- preguntó John – ¿Qué te parece lo más prioritario Simon?

- Hay varias cosas pero de una de ellas depende el resto en este caso. Sin duda crear la base del juego, un conjunto de mazmorras que un jugador recorre hastas encontrar una salida.

-Está claro.- afirmó John- En nuestra primera entrega trataremos de conseguir un objetivo básico. Si os parece vamos a tomar esa historia como base para nuestro primer Sprint. Calculo que necesitaremos de 2 a 4 semanas para llevarlo a cabo, y está bien porque al menos eso es lo que debiera llevarnos cada entrega.

-¿Pero si te pasas de tiempo? - preguntó Float - ¿El Product Owner se tragará mi alma atormentada?

-Tranquilo tú ya vendiste tu alma hace tiempo a cambio de un mazo de infrecuentes de Magic. Si acaso me quedaré con tu camiseta de Blind Guardian – replicó Simon.

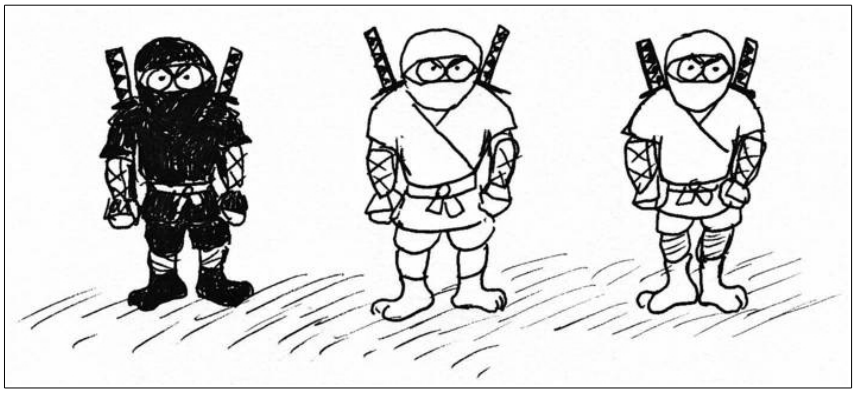
John templó los ánimos:

-No pasa nada, para eso son las reuniones de Scrum, para evaluar esos errores de estimación, problemas, etc... y ajustarlos para la próxima vez. Por otro lado hay que ir haciendo una estimación de todas las historias entre todos.

-¿Entre todos?-preguntó Float -pensaba que esto lo impondrías tú.

-No, siempre por consenso; se puede hacer una especie de juego de estimación llamado Planning Poker. En la siguiente vuelta lo podemos aplicar pero en esta empezaremos por la historia que es claramente más prioritaria y de la que dependen el resto.

5.2. El primer Sprint



John trató de centrar el balón:

Bueno, tenemos un primer objetivo, y aunque parezca poco si lo desglosamos en pequeñas tareas veréis que tiene su miga. Con esas pequeñas tareas formaremos el Sprint Backlog. Cada uno se ocupará de una tarea y... a programar. Por si no os habéis dado cuenta, estamos en pleno Sprint y esto sería el Sprint Planning Meeting.

-La cabra tira al monte ya sabéis – intervino Peter - esta primera historia es una cosa concreta pero supongo que vamos a desarrollarla con clases. A mí se me ocurren al menos tres clases y podriamos empezar por ahí: cada clase una tarea del Sprint Backlog y cada una de esas tareas para cada uno de los miembros del equipo.

-Déjame adivinar: clase Main, clase Maze y clase Dungeon – dijo Float – pero antes que nada ¿cómo asignamos los marrones?

-Para esta primera vez vamos a coger las tareas un poco a boleo. Ya iremos ajustando en siguientes vueltas, o en la reunión diaria de Scrum se puede detectar rápidamente si hay que corregir algo.

-De todas formas tampoco se me hace mucho trabajo – se sorprendió Float – esto en un día lo tengo hecho.

John se puso en pie:

-**Hecho**. Esta es una palabra que hay que discutir o al menos dejar clara para todo el mundo. ¿qué es lo que consideraremos hecho?

- Pues... compilado correctamente. - respondió Float.

-¡No!

-Su diagrama de clases correspondiente, compilado, ejecutado de alguna forma y documentado por dentro y por fuera con javadoc – respondió Peter.

-¡Nopes! No me seais cutres. A todo lo que ha dicho Peter hay que añadirle como poco las pruebas unitarias. Podriamos meter más incluso: pruebas de integración, pruebas de rendimiento, pruebas de usuario, etc... Como poco meteremos las unitarias, así que vale, cada clase creada debe tener su correspondiente clase de testeo jUnit. Y por supuesto, todo debe compilar, funcionar, pasar los test y entonces se considerará algo como HECHO.

-Ah ¿vamos a hacerlo en Java? Es broma Peter, no me mires así – dijo Float – lo malo que para el tema de pasar los test, etc... vaya paliza.

-¿Si verdad? - dijo John - Venga Float, aparte de programar la principal, te vas a currar un script de Ant. Dos tareas para ti. Yo me ocupo de la clase Maze y tú Peter de la clase Dungeon. Cread los postit y ponedlos en la sección Sprint. O bueno, tiramos de Kunagi.

-Para para, antes que nada – interrumpió Float – Simon: ¿cómo se va a llamar este producto? Nos vendría bien para cuando hagamos el repositorio, creemos el paquete, etc

-Pues yo había pensado llamarlo TOOOMB.

-Suena a hostión ominoso – dijo Peter – me das miedo.

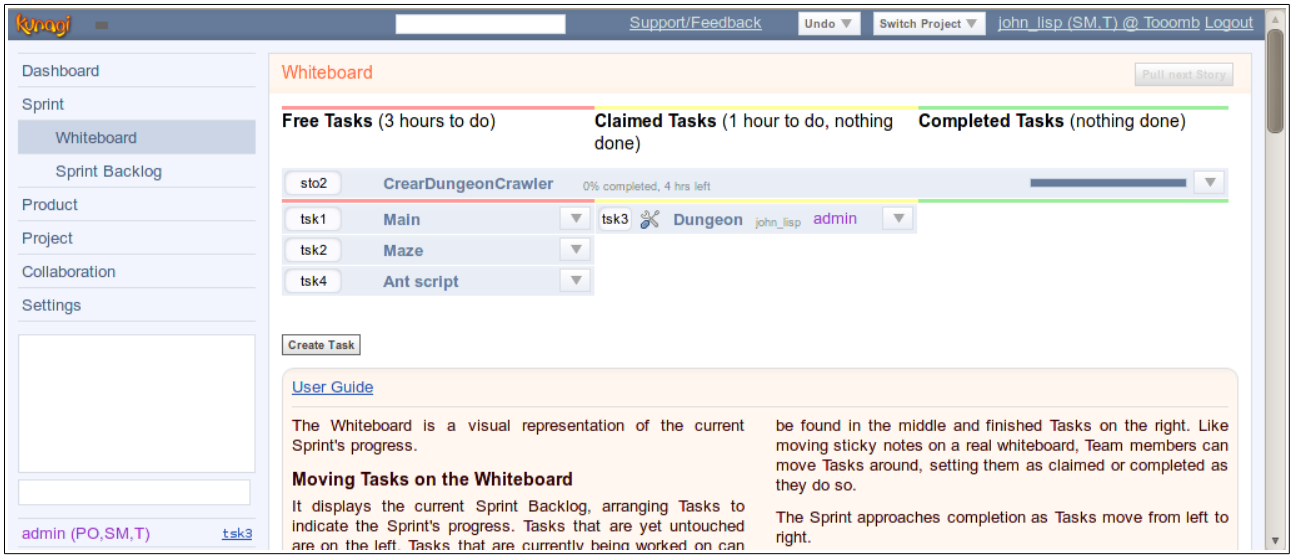
-Significaría algo así como Tooomb is Object Oriented Obscure Maze Builder.

-¡Ok! - dijo John – Float, crea un repositorio de software, nos lo clonamos y nos ponemos a trabajar ya. ¡A Sprintar!

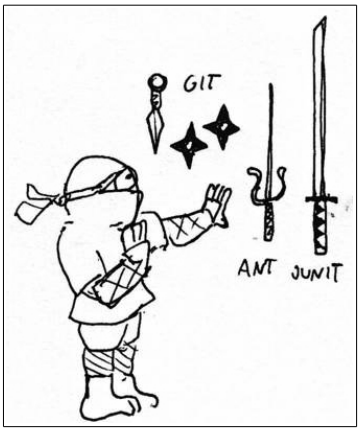
Las tareas en Kunagi

En el Kunagi, una vez que se selecciona una historia hay que darle una estimación de tiempo directamente o con una pequeña aplicación de planning poker y solo entonces podemos hacer un *pull to Sprint*, es decir que la usamos en sl siguiente Sprint donde la desglosaremos en Tasks o tareas.

En la siguiente pantalla vemos cómo se crean las Task del Sprint:



5.3. Show me the code!!



Float crea el fichero build.xml de ant y una clase main que instancia la clase Maze. De momento nada más:

El script build,xml de ant que utilizarán todos:

```
<?xml version="1.0"?>
<!-- author: float -->
<!-- NOTE: make sure that you edit using utf-8 -->
<!-- To generate: ant [target] o ant -buildfile [file.xml] [tarea] -->
<project name="Tooomb" default="run" basedir=".">

	<property name="dir.src" value="src"/>
	<property name="dir.build" value="bin"/>
	<property name="dir.dist" value="dist"/>

	<!-- Generates output dirs: ant prepare -->
	<target name="prepare" description="Crea los directorios">
		<mkdir dir="${dir.build}"/>
		<mkdir dir="${dir.dist}"/>
	</target>

	<!-- Cleans generated dirs: ant clean -->
	<target name="clean" description="Elimina todos los ficheros generados">
		<delete dir="${dir.build}"/>
		<delete dir="${dir.dist}"/>
	</target>

	<!-- Compiles, first prepares: ant compile -->
	<target name="compile" depends="prepare" description="Compilar todo.">
		<javac includeantruntime="none" srcdir="${dir.src}" destdir="$
{dir.build}"/>
	</target>

	<!-- Generates jar, first compiles: ant jar -->
	<target name="jar" depends="compile" description="Genera un fichero jar en
el directorio 'dist'.">
		<jar jarfile="${dir.dist}/project.jar" basedir="${dir.build}"/>
	</target>



	<target name="run" depends="jar" description="Run">
		<echo message="Ok, compiled &amp;amp; packed, let's run: " />
		<!-- ejecuta el método main dentro de la clase java -->
```

```
            <java classname="info.pello.tooomb.Tooomb">
            <classpath path="${dir.dist}/project.jar" />
      </java>
</target>

</project>
```

La clase principal del proyecto.

```
package info.pello.tooomb;

/**
 * the class that takes you deep inside nightmare
 * no one will hear you screaming. rest in pieces.
 * @author float
 */
public class Tooomb {
      public static void main (String args[]) {
            Maze maze = new Maze("Test Maze");
            System.out.println("Tooomb (C) Bubble Telecom - 2013");
      }
}
```

Vamos a ver algunas refactorizaciones que emplean los muchachos desde el primer momento:

Refactorización: Rename y Encapsulate Field

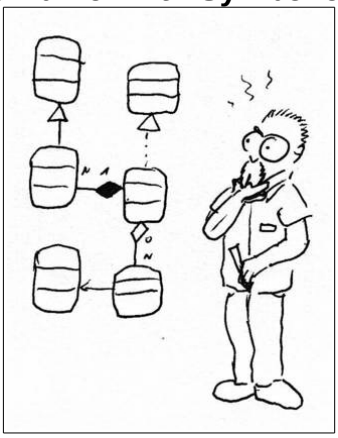
John Lisp comienza escribiendo la clase Maze y mete un atributo público. Eso está muy mal:

```
package info.pello.tooomb;

/**
 * The Maze, where the action takes place
 * @author John Lisp
 *
 */
public class Maze {

      String mazeName;

      /**
       * @param mazeName
       */
      public Maze(String mazeName) {
            this.mazeName = mazeName;
      }


}
```

John Lisp refactoriza y hace dos cosas: renombra el campo mazeName a name y encapsula ese mismo campo name. De hecho solo incluye el getter, el setter no se contempla para este caso.

```java
package info.pello.tooomb;
/**
 * The Maze, where the action takes place
 * @author John Lisp
 *
 */
public class Maze {

	private String name;

	/**
	 * @param name for the maze
	 */
	public Maze(String name) {
		this.name = name;
	}

	/**
	 * @return the name of the maze
	 */
	String getName() {
		return name;
	}

}
```

<u>Refactorización:</u> Replace Magic Numer with Symbolic Constant y Extract Method



Peter comienza escribiendo la clase Dungeon y de primeras le sale lo siguiente:

```java
package info.pello.tooomb;

import java.util.Random;

/**
 * Each of the individual rooms of the maze.
 * We can set the Dungeon with a scary name
 * @author Peter Class
 *
 */
public class Dungeon {
	private String name;
	private int damage;
	private Random random = new Random();

	/**
	 * default constructor, dungeon is baptized
	 * with a generic Dark Dungeon name.
	 */
	public Dungeon () {
		name = "Dark Dungeon";
		damage = random.nextInt(6);
	}

	/**
	 * constructor with a name for the dungeon
	 * @param name
	 */
	public Dungeon(String name) {
		this.name = name;
		damage = random.nextInt(6);
	}
}
```

Hay dos cosas que le sugieren mal olor (code smells). Por un lado está ese número 6 que no se sabe de dónde sale o por qué es ese y no otro: vamos, es un Magic Number..Hay que explicarlo o al menos agruparlo como una constante, así que haremos un Extract Constant. Y luego, la inicialización que se produce para el daño (damage), se repite en ambos constructores. Esto se puede extraer a otro método en el que seguro podremos ir añadiendo más inicializaciones.

```java
package info.pello.tooomb;

import java.util.Random;

/**
 * Each of the individual rooms of the maze.
 * We can set the Dungeon with a scary name
 * @author Peter Class
 *
 */
public class Dungeon {
	private static final int MAX_DAMAGE = 6;
	private String name;
	private int damage;
	private Random random = new Random();
```

```
	/**
	 * default constructor, dungeon is baptized
	 * with a generic Dark Dungeon name.
	 */
	public Dungeon () {
		name = "Dark Dungeon";
		init();
	}


	/**
	 * constructor with a name for the dungeon
	 * @param name
	 */
	public Dungeon(String name) {
		this.name = name;
		init();
	}

	/**
	 * inits Dungeon parameters
	 */
	private void init() {
		damage = random.nextInt(MAX_DAMAGE);
	}
}
```

Uno de los atributos esenciales de una mazmorra debe ser si tiene salida o no. Peter añade ese atributo que se iniciará a false en el método init y añade un método para establecerlo a verdadero.

```
package info.pello.tooomb;

import java.util.Random;

/**
 * Each of the individual rooms of the maze.
 * We can set the Dungeon with a scary name
 * @author Peter Class
 *
 */
public class Dungeon {
	private static final int MAX_DAMAGE = 6;
	private String name;
	private int damage;
	private Random random = new Random();
	private boolean hasExit;

	/**
	 * default constructor, dungeon is baptized
	 * with a generic Dark Dungeon name.
	 */
	public Dungeon () {
		name = "Dark Dungeon";
		init();
	}
```

```
	/**
	 * constructor with a name for the dungeon
	 * @param name
	 */
	public Dungeon(String name) {
		this.name = name;
		init();
	}

	/**
	 * inits Dungeon parameters
	 */
	private void init() {
		damage = random.nextInt(MAX_DAMAGE);
		hasExit = false;
	}


	/**
	 * @param hasExit is set to true
	 */
	public void setHasExit() {
		this.hasExit = true;
	}

}
```

John Lisp sigue escribiendo su clase Maze y crea un método para iniciar el laberinto: buildMaze. Este rellena el array de 10x10 que contiene el escenario y en cada coordenada mete una instancia de Dungeon. Luego de forma aleatoria hace que en uno de los Dungeon haya una salida.

```
	...
	private Dungeon[][] dungeons = new Dungeon[10][10];
	private Random random = new Random();

	...

	/**
	 * generates all the Dungeons
	 */
	private void buildDungeons () {
		for (int i= 0;i<10;i++)
			for (int j=0;j<10;j++)
				dungeons[i][j] = new Dungeon("Dark Dungeon");

		int exitI = random.nextInt(10);
		int exitJ = random.nextInt(10);
		dungeons[exitI][exitJ].setHasExit();
	}
		...
```

Esto último queda poco claro. John hace un **Extract Method** de la parte en la que se genera el Dungeon de salida. Luego se puede mejorar ese método, añadir más salidas, etc...

...

```
	/**
```

```
	 * generates all the Dungeons
	 */
	private void buildDungeons () {
		for (int i= 0;i<10;i++)
			for (int j=0;j<10;j++)
				dungeons[i][j] = new Dungeon("Dark Dungeon");

		setExitDungeon();
	}

	/**
	 * sets one of the Dungeons as exit point
	 */
	private void setExitDungeon() {
		dungeons[random.nextInt(10)][random.nextInt(10)].setHasExit();
	}
...
```

Refactorización: Hide delegate

Con el código completo de las clases Maze y Dungeon Float ya puede completar la clase Tooomb que no es más que la principal. Le esta saliendo un poco larga y lo que es peor, aparecen llamadas a varias clases de dominio, aunque sea a través de clases delegadas. Por ejemplo, para comprobar si una mazmorra es la mazmorra de salida hace:

```
...
				if (maze.currentDungeon().isExit()) {
					System.out.println("You are out!!");
					System.out.println("Total moves: " + moves);
					System.exit(1);
				} else {
					System.out.println("Not exit");
				}

...
```

En lugar de eso lo que se debe hacer es ocultar, encapsular la existencia de otras clases. Eso es fundamental en la programación orientada a objetos. Para eso en la clase Maze creamos un método que lo hace todo:

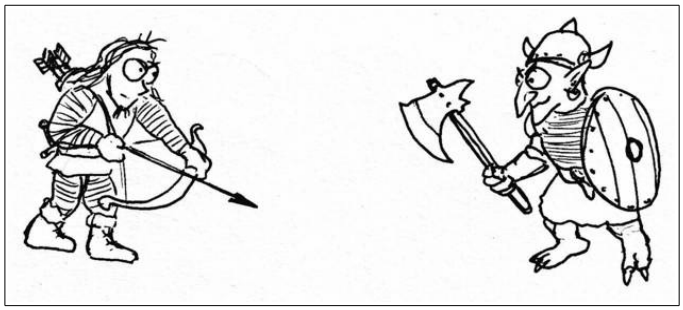
```
...
	/**
	 * tells is current Dungeon has an exit or not
	 * @return
	 */
	public boolean isExit() {
		// TODO Auto-generated method stub
		return currentDungeon().isExit();
	}
...
```

Y ahora en la principal Tooomb ya no tenemos porqué saber que existe la Dungeon:

```
...
				if (maze.isExit()) {
					System.out.println("You are out!!");
					System.out.println("Total moves: " + moves);
					System.exit(1);
				} else {
					System.out.println("Not exit");
```

}

…



Los integrantes del equipo van guardando el código y lo van subiendo al repositorio. El resto se descarga lo de los demás. Así termina un día cualquiera. Al siguiente, habrá que comenzar con

5.4. La Daily Scrum Meeting

Nada más comenzar la mañana siguiente, John convocó el Scrum diario junto a la máquina de café.
-Bueno chavales, esto tiene que ir rápido. Os recuerdo que en Scrum diario cada miembro del equipo tiene que contar qué ha hecho, qué piensa hacer y qué problemas se ha encontrado. De esa forma podemos corregir y ajustar la marcha del desarrollo.

-Yo lo tengo hecho. Hoy me voy a dedicar a trollear en foros de ingeniería del software, y a bajarme gifs animados de 4chan. - comenzó Float.
-Yo también – dijo Peter – me voy a dedicar a dibujar diagramas UML para planificar mis vacaciones.
-Yo no he hecho nada – dijo Simon.
-¿De verdad habéis terminado? - se sorprendió John - Yo he programado la clase Maze, pero no la he probado, no he programado las pruebas unitarias, no lo he integrado con el resto. ¿Vosotros habéis hecho todo eso de verdad?

Se hizo el silencio. La máquina de café gorjeó como si se riera de la situación.
-Pero eso de los test unitarios – comenzó Float – ¿no es un trabajo de bajo nivel y para pringados?
-Nooo, nooo, - se desesperó John – son tan fundamentales como los propios programas. Son su sello de calidad. No habéis terminado nada. Vais a programar las pruebas unitarias, y ya que estamos generaremos el javadoc. Luego subid todo al repositorio para que lo bajemos y cada uno podamos probar todo.

-¿Y yo qué hago? - se quejó Simon – me siento como el hobbit chistoso del grupo. Me gustaría aportar más.

-De momento ponte en contacto con tus amiguetes y asociados para ir sacando más historias para el Product Backlog – contestó Jon.

-¡Ah vale! ¿Os los presentó? Se duchan una vez cada cinco días.

-¡De ninguna manera! Tanto tú, el Product Owner, como yo el Scrum Master debemos comportarnos como barreras de protección que aislen al equipo Scrum de influencias del exterior. Si alqó exógeno quiere algo que hable con nosotros y nadie más. Y ahora venga, ya podéis ir programando las pruebas unitarias. Esto será como asuntos internos: yo haré el test de la clase de Peter y él testeará mi clase.

-¿Pero cómo testeo una clase main? - preguntó Float.

-En tú caso, lo que harás será sentar a Simon frente a la pantalla y que haga un test de usabilidad, como usuario.

5.5. Otro Scrum Diario

El equipo se dirigió a la máquina de café para el Scrum Diario.
-¿Qué tal va?- comenzó John – Yo he mejorado la clase Maze, le he incluido alguna función como la de mostrar mapa y ya he programado la clase de Test y el test se pasa correctamente. Lo único que el build.xml de ant debiera incluir el testeo automátizado.

-Yo también he programado el Test y he completado un poco el código – siguió Peter- Mi única duda es con los puntos de daño de cada Mazmorra, de 0 a 6 me parece excesivo. Ese valor habrá que retocar ¿no Simon?

-Quizá sí, he estado probando el juego básico con Float, y no hay bicho que sobreviva. Como experiencia de usuario es una mierda, pero como Master me chifla. Solo veo dolor.

-¿Y el tema de los valores aleatorios? - intervino Float- ¿Estáis usando la clase Random? ¿Qué tal si nos curramos una clase aparte, mejorada?

-¡Yo, yo! -Levantó la mano Peter - Me añado una tarea nueva al Backlog, voy a crear una clase Dice que represente un dado, que nos sirva tanto para generar esos números como para simular tiradas en el juego.

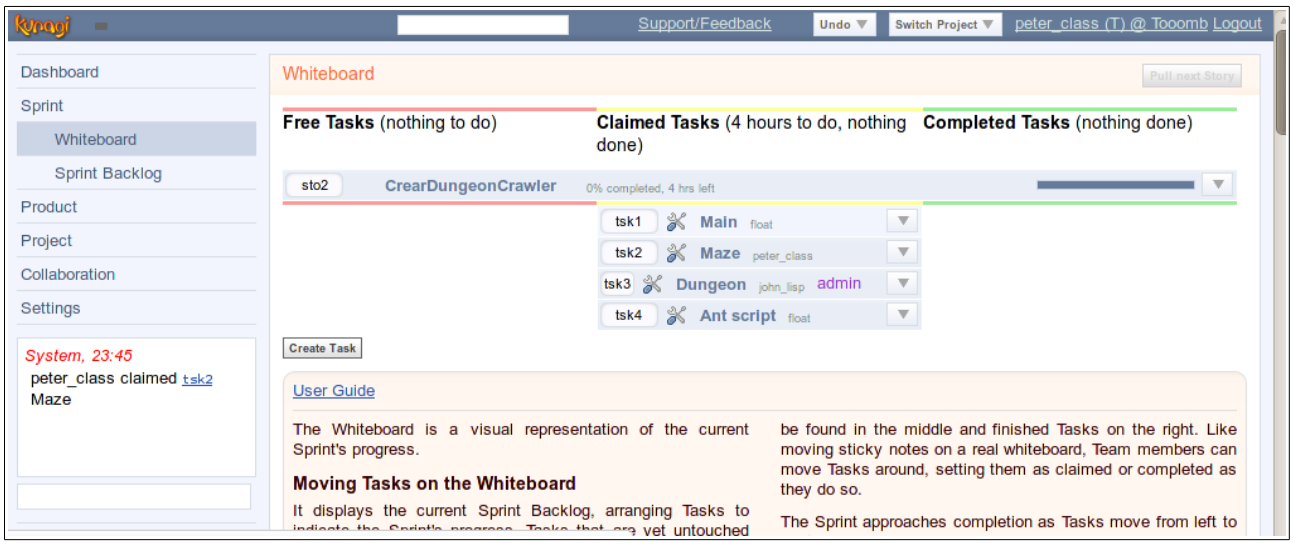
-Buf, tampoco es para tanto – dijo Float - Además acabaremos teniendo un montón de instancias para una tontería.

-La programaré aplicando el patrón Singleton, con lo que garantizo que solo habrá una instancia. Bueno Simon, no sé si tú tienes algo que decir que igual yo no puedo alterar las tareas ni el Backlog.

-Los dados son fundamentales. Supongo que solo es una pequeña tarea. Pero para no alterar mucho el Sprint, basta con que hagas un D6.

El Sprint en Kunagi

En la siguiente pantalla de Kunagi vemos el estado de las tareas dentro del Sprint. Como se ve en este caso todas están en progreso. Es los mismo que se podría hacer con papelitos.



5.6. Llega el Sprint Review

El equipo se reunió para su primer Sprint Review
-Es hora de revisar los resultados de este Sprint. – Comenzó John - ¿Hemos cumplido con las tareas?

-Diría que sí – dijo Peter – hombre, hemos empezado haciendo unos mínimos.

-Unos mínimos, pero ya tenemos una cosa más o menos jugable- dijo Float – mirad:

```
Life: 10 Moves: 0
   0  1  2  3  4  5  6  7  8  9
0 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
1 [ ][ ][ ][ ][ ][ ][ ][@][ ][ ]
2 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
3 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
4 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
5 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
6 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
7 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
8 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
9 [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
   0  1  2  3  4  5  6  7  8  9
Choose direction:
N,S,W,E
```

-Me gusta, y esa arroba me recuerda al Moria y a todos aquellos roguelike de mi adoslescencia. – Dijo Simon.

-Aparte de eso Simon – intervino John- como Product Owner debes decir cómo ves el resultado, si hemos cumplido y cómo ves el Product Backlog de cara al próximo Sprint.

-Yo por mí lo planificado se ha conseguido en esta vuelta. Yo mismo lo he probado y he podido sugerir pequeños cambios como usuario. Quizá en la próxima vuelta se pueda ir a por más, y de momento queda mucho por hacer. Yo esto lo veo muy lanzado y creo que podriamos incluso asignar una historia a cada uno para la siguiente vez.

-Buf, éste se nos entusiasma – se quejó Float- Esto nos pasa por dar mucho en la primera vuelta.

-La velocidad se podrá ir ajustando – dijo John- de todas formas, para la siguiente vuelta no está tan claro que historia escoger así que haremos el Planning Poker para las estimaciones.

-¿Habrá pasta en juego? - preguntó Simon.

-No, habrá curro en juego. Bien, y ahora como Scrum Master debo supervisar la aplicación de Scrum así que la siguiente reunión será la de:

5.7. Sprint Retrospective

-Bueno, ¿qué os está pareciendo esto de Scrum?

-Está bien -comenzó Float- la verdad es que al principio parecía que iba a ser el festival de la reunión pero lo cierto es que tengo la sensación de programar a saco. Y en las reuniones se habla de problemas de programción y no de las cabeceras que debe tener un documento de análisis. Me gusta, pero a Peter el metodológico y el discípulo de Rational no sé yo...

-Pues mira, dentro de lo que cabe, se programa pero se hace bien. Se testea, se comparte el código, todo eso obliga a todo el mundo a hacer las cosas correctamente y además nos enteramos de otras partes del código, no solo lo nuestro. Y además, lo de refactorizar es algo que engancha.

-Fijo- dijo Float - Es mucho mejor dejar el código claro que llenarlo de comentarios, por muy estándarizados que sean son un rollo.

-Aparte que te lleva a hacer un código más pro – terminó John – Bueno, por mi parte en este primer ciclo quizá hemos empezado un poco a lo loco. Y es normal que en la primera vuelta fallemos en las estimaciones e igual nos hallamos quedado cortos. La próxima planificaremos mejor, veremos los Burndown y todo eso para ver el progreso de las tareas, etc...

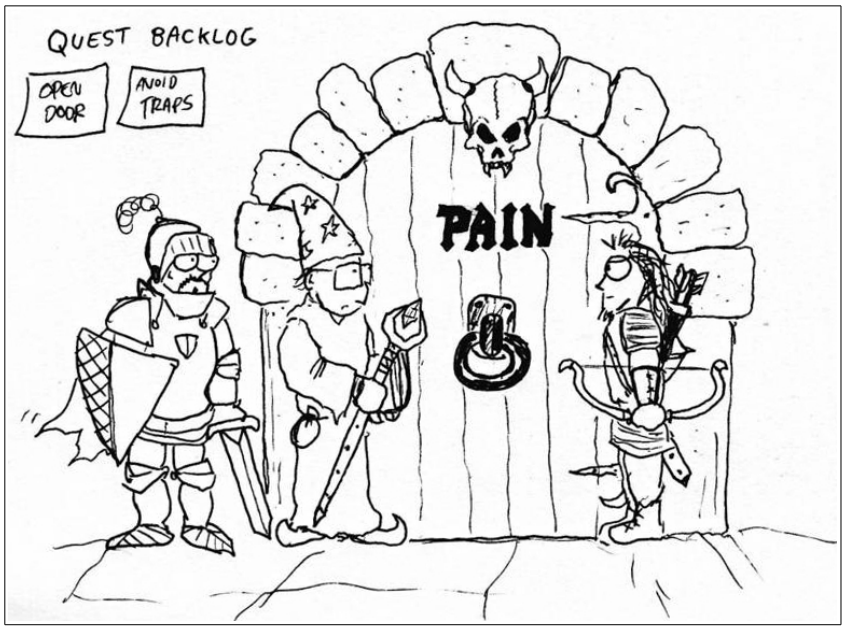
- buf, pero eso es meterse en jardines en lugar de simplificar – se quejó Float.

-No tanto, con que metamos lo mínimo en la herramienta Kunagi nos calculará todo.

Further reading:

- Elizabeth Woodward, Steffan Surdek, Mathew Ganis: A Practical Guide to Distributed Scrum. Cap 3: Starting a project.

6. Otro incremento



John Lisp convocó a la reunión a todo el mundo trayendo consigo una baraja de poker.

-Antes de lanzarnos sin más a por otra historia a lo loco, vamos a tratar aplicar más elementos de Scrum. Lo primero que debieramos haber hecho es una estimación de las historias que tenemos en el Product Backlog. Debemos asignar entre todos Story Points a cada historia.

-¿Qúe son esos puntos?

-Son el número de días ideales que creemos que vamos a necesitar para cumplir una historia, es como digo una forma de asignar un valor a las historias,

-Vale – dijo Float- haz una estimación más o menos de cada historia y que Simon diga cuál es la siguiente para hacer.

No, no, insensatos – respondió John – tomad estas cartas, a cada uno de la pasaré un palo completo de la baraja, para tí Float las picas, para Peter los tréboles, para Simon los corazones y para mí los diamantes. Hay quien da una baraja para cada uno, existen otro tipo de barajas especiales incluso con cartas para indicar “necesito un café”, pero lo haremos así, en plan simple.

-¿No podemos hacerlo con alguna herramienta informática? - preguntó Peter.

-Sí claro, incluso tenemos online el http://www.planningpoker.com. Esto no es un juego para pasar el rato; es una oportunidad de que todos los que estamos en el equipo participemos en la estimación. Es justo, es honesto , es transparente. En muy simple, vamos historia por historia, y por cada una sacad boca a abajo una carta con la estimación que le dáis a la historia cada uno. Tenemos que conseguir un consenso en los puntos. Aquellos que den puntuaciones extremas para arriba o para abajo deberán exponer sus motivos. También se supone que la opinión de los programadores que se vayan a ocupar de la historia tendrá más peso.

Vamos a estimar los puntos de la primera historia: El jugador puede ser de distinta clase,

el rollo de que pueda ser bárbaro, elfo, enano, mago,...

-¿Y no puede ser las cuatro cosas a la vez? - preguntó Float.

Simon comenzó:

-En uno de los apéndices de AD&D edición 2.5 se menciona la posibilidad...

-¡Vale! – gritó John - ¡sacad una carta cada uno!!.

Todos sacaron una carta boca abajo con su estimación para esa historia.

-A ver, Peter le da un 8, Float le da un 5, Simon le da un 10 y yo le doy un 8. Float lo único que quiere es que le pregunte sobre ese número para hacerme la rima y a Simon no lo puedo tomar en serio. Pues bien, según nuestra estimación hacer esa historia tendría un coste de 8 puntos.

-¿Y cuánto tiempo nos debiera llevar? ¿En qué se traducen los 8 puntos? - preguntó Peter.

-Bien preguntado. Según la Velocity del equipo, es decir nuestra capacidad para completar historias, son los puntos que somos capaces de cumplir en cada Sprint. Por lo tanto, si nuestra Velocity fuera 8, esta historia nos la ventilamos un 1 Sprint, y necesitamos 2 semanas (si ese el tamaño de nuestro Sprint). Si nuestra velocity es de 4, pues necesitariamos 2 Sprints para completar la historia.

Al poco el equipo ya tenía asignados los History Points para cada historia. Peter no parecía muy convencido por lo poco ortodoxo del método:

-Pero John ¿para que nos sirve esto de las estimaciones y la velocidad? No me lo digas, ¿es para que Scrum parezca que va en serio?

-A ver, es una forma simple, honesta y adaptable de conocer el desarrollo del proyecto y de paso de medir al equipo y conocer sus límites. Con estos datos podemos crear un Release Burndown Chart, es decir, un gráfico que muestra la relación entre los History Points que quedan por hacer (eje Y) y el tiempo en forma de Sprints (eje X).

-Si queremos reflejar mejor eventos como que en algunos Sprints se añaden nuevas historias al Backlog y estás se van consumiendo podemos usar un gráfico de Release Burndown Bar.

Further Reading

- Mike Cohn, Agile Estimating and Plannin. Cap 6, estimation techniques.
- http://eugeniaperez.es/wordpress/?p=148 : intro a la estimación
- http://www.planningpoker.com/ : herramienta online para Planning Poker

6.1. El Release Plan

Ahora había que decidir qué historias elegir para el siguiente Sprint.

-A mí todas me molan la verdad- dijo Simon – Me da igual cuál de ellas. ¿Qué criterio debo seguir?

-Hay muchas cosas a tener en cuenta, pero sobre todo debes dar importancia a aquellas historias que de verdad dan valor al producto. El resto deben ir abajo en la lista de prioridades.

-El tema de los objetos que se pueden encontrar en las mazmorras mola mucho: Armas, pociones, tesoros,... - dijo Simon- yo tiraría por ahí. Que el jugador, aparte de salir, pueda hacerlo con las manos llenas, o que incluso pueda curarse. Pero también me gustaría darle protagonismo al jugador.

-Bien podemos tomar la historia de los objetos y además de del Jugador. Esta segunda historia la podemos iniciar y continuarla en Sprints posteriores. Esa es otra, nada dice que el Sprint termine al acabar las tareas. Siempre es preferible mantener tiempos fijos pero las funcionalidades flexibles.

-Otra cosa, en el momento de crear las tareas debéis hacer una pequeña estimación del tiempo que os va a llevar en horas esa tarea. Eso es más fácil para hacerlo individualmente. De esas horas iréis “quemando” (burned) algunas y lo podéis indicar fácilmente usando el Kunagi.

Further reading:

- Roman Pilcher: Agile Product Management with Scrum. Cap 3: prioritizing the Product Backlog.

6.2. Preparando otro Sprint

John señaló las dos historias escogidas:

-Bien, pues ya estamos en las puertas del siguiente Sprint. Simon nos ha escogido otras dos historias, el tema de los tesoros que se pueden encontrar en la mazmorra y el jugador. Es hora de sacar las tareas de cada una.

-Partiendo de algo similar a lo anterior, yo crearía tareas asociadas a las clases – comenzó Peter. - Está claro que habrá que incluir una clase Personaje y una clase Tesoro.

-Quiźa no sea tan simple como eso, quiero decir que igual es como una clase pero luego esas clases generan otras cuando empecemos a refactorizar. Aparte que tienes pendiente lo del Dado; esa puede ser una tarea clara – dijo John.

-Eh espera un poco – intervino Float – Vosotros os lo pasáis pipa creando clases pero yo tengo una clase principal que tiene que manejar todo. Conforme vayamos incluyendo más elementos en el juego esa clase principal crecerá como La Cosa y nos engullirá.

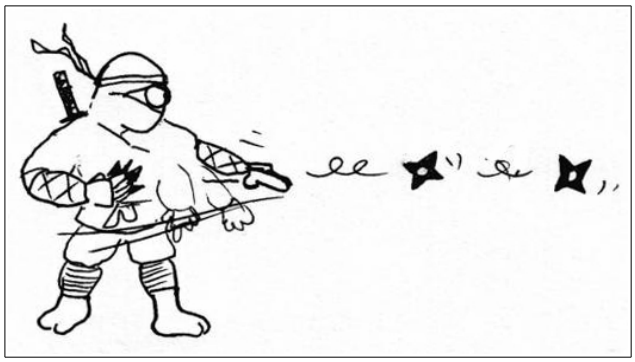
-Tú tienes una parte crucial Float – dijo John – tú debes integrar todo lo que vamos creando los demás. Creo que esa será una tarea continua a lo largo de todo el Sprint, aunque eso es algo implícito.

Peter carraspeó:

-No es por meterme en tu terreno, pero la clase principal debes mantenerla lo más simple posible. Yo te pondría dos tareas: una clase que se intermediaria entre la principal y todo el dominio que contenga la lógica del juego y por otro lado una clase que te ayude a simplificar la E/S con el usuario.

-Está bien que te metas en su terreno Peter – sonrió John – en las metodologías ágiles esa es una idea fundamental: el código es en cierto modo de todos, y conviene que todos conozcamos el código de los demás con técnicas como el Pair Programming.

6.3. Hands on code



Peter Class crea una pequeña clase para generar números aleatorios emulando un dado de 6 caras:

```
package info.pello.tooomb;

import java.util.Random;

/**
 * a class that behaves like a D6 dice.
 * @author Peter Class
 *
 */
public class D6 {
	Random random = new Random();

	/**
	 * default constructor
	 */
	public D6 () {

	}

	/**
	 * roll the D6 dice, gives a number between 0 and 5
	 * @return
	 */
	public int roll () {
		return random.nextInt(6);
	}

}
```

Y su clase de testeo correspondiente, se podría hacer con un RepeatedTest pero se simplifica.

```
package info.pello.tooomb.tests;

import info.pello.tooomb.D6;
import static org.junit.Assert.*;

import org.junit.AfterClass;
import org.junit.Test;

/**
 * Testing class for Maze
 * @author Peter Class
 *
 */
public class D6Test {

	D6 d6;

	/**
	 * we test the dice rolling 100 times
	 */
	@Test
	public void testRoll() {
		d6 = new D6();

		for (int i = 0;i < 100;i++) {
			assertTrue(d6.roll() >= 0 && d6.roll() <= 5);
		}
	}

}
```

Como esta clase se va a utilizar mucho y no interesa generar tropecientas instancias para una tarea así, Peter aplica el patrón Singleton para garantizar que solamente existirá una instancia de D6. Este patrón básicamente genera una instancia interna estática de si misma y oculta su constructor. Para usarla se llama a un método que devuelve esa instancia única y desde ella se usan las funciones que se precisen:

```
package info.pello.tooomb;

import java.util.Random;
```

```
/**
 * a class that behaves like a D6 dice
 * following Singleton pattern
 * @author Peter Class
 *
 */
public class D6 {
	Random random = new Random();
	private static D6 d6;

	/**
	 * private constructor
	 */
	private D6 () {

	}

	/**
	 * this gives access to unique instance of D6
	 * @return
	 */
	public static D6 getD6 () {
		if (d6 == null) {
			d6 = new D6();
		}

		return d6;
	}

	/**
	 * roll the D6 dice, gives a number between 0 and 5
	 * @return
	 */
	public int roll () {
		return random.nextInt(6);
	}

}
```

Y el test unitario cambia un poco:

```
…
	/**
	 * we test the dice rolling 100 times
	 */
	@Test
	public void testRoll() {

		for (int i = 0;i < 100;i++) {
			assertTrue(D6.getD6().roll() >= 0 && D6.getD6().roll() <= 5);
		}
	}
…
```

La clase Character

John Lisp sen encarga de la clase Character para representar al jugador. En principio hace una clase genérica deprisa y corriendo con una serie de atributos comunes:

```
package info.pello.tooomb;
```

```java
/**
 * Represents the player character
 * code smells everywhere
 * @author John Lisp
 *
 */
public class Character {
	private String name;
	private int strength;
	private int speed;
	private int intelligence;
	private int type;
	private int life;

	/**
	 * default constructor, calls initialize
	 * @param name of character
	 * @param type of character
	 */
	public Character(String name, int type) {
		this.name = name;
		this.type = type;
		initialize();
	}

	/**
	 * initializes attributes
	 */
	private void initialize () {
		strength = D6.getD6().roll();
		speed = D6.getD6().roll();
		intelligence = D6.getD6().roll();
		life = strength * 2;
	}

	/**
	 * the character attacking roll
	 * @return  attack points
	 */
	public int attack () {
		int points = 0;
		// the attack calculation depends on
		// the type of character -code smell-
		switch (type) {
			case 0: // barbarian
				points = strength + D6.getD6().roll();
				break;
			case 1: // elf
				points = speed + D6.getD6().roll();
				break;
			case 2: // dwarf
				points = strength + D6.getD6().roll();
				break;
			case 3: // wizard
				points = intelligence + D6.getD6().roll();
				break;
			default:
				points = D6.getD6().roll();
```

```
                              break;
                }

                return points;
        }

        /**
         * the character defense roll
         * @return defense points
         */
        public int defend () {
                int points = 0;
                // the defense calculation depends on
                // the type of character -code smell-
                switch (type) {
                        case 0: // barbarian
                                points = (speed + strength)/2 + D6.getD6().roll();
                                break;
                        case 1: // elf
                                points = (speed + intelligence)/2 +
D6.getD6().roll();
                                break;
                        case 2: // dwarf
                                points = strength + D6.getD6().roll();
                                break;
                        case 3: // wizard
                                points = intelligence + D6.getD6().roll();
                                break;
                        default:
                                points = D6.getD6().roll();
                                break;
                }

                return points;
        }

        /*
         * shows character info
         */
        public String info () {
                return name +" [strength=" + strength + ", speed=" + speed
                                + ", intelligence=" + intelligence + ", type=" +
getType()
                                + ", life=" + life + "]";
        }

        /**
         * @return the name
         */
        public String getName() {
                return name;
        }

        /**
         * takes some life points
         * @param points to take
         */
        public void takeLife(int points) {
                life -= points;
```

```
	}

	/**
	 * @return remaining life points
	 */
	public int getLife() {
		return life;
	}

	/**
	 * given type code returns Its name
	 * @return the name of player type
	 */
	public String getType() {
		String typeName = "";

		switch (type) {
		case 0: // barbarian
			typeName = "Barbarian";
				break;
		case 1: // elf
			typeName = "Elf";
				break;
		case 2: // dwarf
			typeName = "Dwarf";
				break;
		case 3: // wizard
			typeName = "Wizard";
				break;
		default:
			typeName = "Unknown";
				break;
		}

		return typeName;
	}
}
```

Esta clase tiene **un montón** problemas que se pondrán sobre la mesa en el Scrum Diario.

Retoques en la clase principal Tooomb

Con la inclusión de esta clase Character hay que hacer unos cambios en la principal. Primero habrá que solicitar al usuario qué tipo de personaje quiere:

…

```
		// Choose character data
		System.out.println("Please choose a name");
		name = reader.next();
		System.out.println("Please choose character type:");
		System.out.println(" 0: Barbarian, 1: Elf, 2: Dwarf, 3: Wizard");
		characterType = reader.nextInt();
		player = new Character(name, characterType);
```

…

Esto es un adefesio que se comentará en el Scrum diario.

Luego se modifica la forma de mostrar la información de usuario:

…

```
System.out.println("Player: " + player.info() + "\n" + " Moves: " + moves);
```

...

Y por último hay que cambiar la forma de quitar vida y de verificar si el usuario está muerto:

```
...
      player.takeLife(maze.getDamage());
                        if (player.getLife() <= 0) {
                              System.out.println("You are dead");
                              System.exit(1);
                        }
...
```

La clase Treasure

Por su parte Peter Class crea la clase para representar los tesoros que se puede encontrar el jugador en el juego. Le queda algo así:

```
package info.pello.tooomb;


/**
 * Represents treasures that you can find in Dungeons
 * there are three types: GOLD, WEAPON, POTION
 * @author Peter Class
 *
 */
public class Treasure {
      private int type;
      private String name;
      private int value;
      public static final int GOLD = 0;
      public static final int WEAPON = 1;
      public static final int POTION = 2;

      /**
       * default constructor
       * @param type
       * @param name
       * @param value
       */
      public Treasure (int type, String name, int value) {
            this.type = type;
            this.name = name;
            this.value = value;
      }


      /**
       * @return the type
       */
      public  int getType() {
            return type;
      }

      /**
       * @return the name
```

```
	 */
	public String getName() {
		return name;
	}
	/**
	 * @return the value
	 */
	public int getValue() {
		return value;
	}

	/**
	 * applies only for weapons
	 * @return random number of damage
	 */
	public int hitPoints () {
		return D6.getD6().roll()/2;
	}

	/**
	 * applies only with potion
	 * @return life points
	 */
	public int heal() {
		return value + D6.getD6().roll();
	}

	/* (non-Javadoc)
	 * @see java.lang.Object#toString()
	 */
	@Override
	public String toString() {
		return "Treasure [type=" + type + ", name=" + name + ", value=" +
value
				+ "]";
	}


}
```

Este código también huele y Peter es consciente que tendrá que retocarlo. Eso de que dependiendo del tipo se pueda llamar a una clase sí y a la otra no...

6.4. Scrums diarios

John Lisp convocó a las huestes a la reunión diaria. Bueno, ya no hacía falta llamarlos, los fieles acudían a la cita de forma natural o casi ritual.

-Bien ya conocéis las preguntas: ¿qué habéis hecho?¿qué vais a hacer?¿qué problemas habéis tenido?

Peter comenzó:

-Yo ya he hecho la clase Singleton para el dado de 6 caras. Bueno, en realidad puede servir para cualquier operación aleatoria. No he tenido problema. La idea ahora era o crear otros dados o crear un dado genérico al que se le pudieran pasar las caras como parámetro. Por otro lado tengo hecho lo del Tesoro. Aunque sin testear.

-Yo ya he creado la clase Character para el personaje, es muy genérica y el tipo se distingue por un atributo de tipo entero. ¿y tú Float? - preguntón John.

-Pues he recogido esos cambios y he tenido que tocar la clase principal. Ya está todo integrado. Pero me doy cuenta...

-Te das cuenta de que se te está apelotonando mucho código y la cosa empieza a quedar farragosa para ser un Main. Debes crear una especie de clase de fachada para interactuar con las clases de domino. La idea es que desde la clase principal, en lugar de llamar a todas las clases de dominio (Maze, Dungeon, etc..) se llame a los métodos de una única clase. De esta forma evitamos aún más las llamada directas o llamadas a clases delegadas. Y la independencia entre el juego y el interfaz será mucho mayor.

-Espera -dijo Float- no es por malmeter pero esa clase Treasure que ha hecho Peter es una cutrez. Tiene muy mala pinta eso de que haya funciones para un tipo sí y para otro no.

Simon rompió su silencio:

-Es cierto, yo no he visto gran cosa de programación orientada a objetos pero me esperaba alguna especie de superclase Tesoro y luego subclases concretas como Oro, Arma, Poción,...

-Soy consciente del Code Smell. De hecho lo voy a refactorizar, pero lo mismo le debiera decir a John. Su clase Personaje también tiene un tipo y está llena de condicionales switch dependiendo de él.

-Recojo el guante, creo que los dos haremos una refactorización parecida. Igual nos sentamos los dos para hacerlo, vamos a tener que aplicar algunos patrones del GoF.

-Que Dios los tenga en su gloria.

6.5. Refactor Mercilessly

Peter y John se sientan juntos ante la clase Treasure. Tiene algo evidentemente cutre y es que hay métodos que se invocan solo según el valor de un atributo: type. Es decir, si la clase es de tipo POTION, entonces se puede invocar el método heal. Bueno, ni eso, porque no hay un control. La cuestión es que en la clase Treasure hay cosas que se aplican solo a ciertos tipos. Esto nos llevará a las refactorizaciones:

Refactorización: Replace Type Code with Subclasses y Push Down Method/Field

-Esto está claro- dijo Peter- vamos a cambiar ese atributo tipo por subclases. Además así podremos mover metodos y campos a subclases.

La nueva clase Treasure, obviando los pasos intermedios al refactorizar, quedaría así:

```java
package info.pello.tooomb;

/**
 * Represents treasures that you can find in Dungeons
 * there are three types: GOLD, WEAPON, POTION
 * @author Peter Class, John Lisp
 *
 */
public abstract class Treasure {
	private int type;
	private String name;
	private int value;
	public static final int GOLD = 0;
	public static final int WEAPON = 1;
	public static final int POTION = 2;

	/**
	 * Constructor for Treasure
	 * @param name
	 * @param value
	 */
	public Treasure (String name, int value) {
		this.name = name;
		this.value = value;
	}

	/**
	 * factory method to generate treasures depending on type
	 * @param type
	 * @param name
	 * @param value
	 * @return
	 */
	public static Treasure create(int type, String name, int value) {
		switch (type) {
		    case GOLD:
		       return new Gold(name,value);
		    case WEAPON:
		         return new Weapon(name,value);
		    case POTION:
		         return new Potion(name,value);
		      default:
		       throw new IllegalArgumentException("Type error");
		}
	    }


	/**
	 * @return the type
	 */
	public abstract int getType();

	/**
	 * @return the name
	 */
	public String getName() {
		return name;
	}
	/**
```

```
	 * @return the value
	 */
	public int getValue() {
		return value;
	}

	/* (non-Javadoc)
	 * @see java.lang.Object#toString()
	 */
	@Override
	public String toString() {
		return "Treasure [type=" + type + ", name=" + name + ", value=" +
value
				+ "]";
	}

}
```

Como se puede ver, ahora Treasure es una clase abstracta y por tanto no se puede instanciar, y para crear objetos más concretos usamos el método factory create.

En las subclases, lo que hacemos es extender Treasure, implementar el métiodo abstracto getType y hacer un **Push Down** de los métodos que solo conciernen a determinadas clases.

La clase Weapon

```
package info.pello.tooomb;
/**
 * @author Peter Class
 *
 */
public class Weapon extends Treasure {

	/**
	 * constructor calls super class
	 * @param name
	 * @param value
	 */
	public Weapon(String name, int value) {
		super(name, value);
		// TODO Auto-generated constructor stub
	}

	/**
	 * applies only for weapons
	 * @return random number of damage
	 */
	public int hitPoints () {
		return D6.getD6().roll()/2;
	}

	public int getType () {
		return Treasure.GOLD;
	}

}
```

Y la clase Potion

```
package info.pello.tooomb;

/**
 * @author luser
 *
 */
public class Potion extends Treasure {

	/**
	 * constructor calls super class
	 * @param name
	 * @param value
	 */
	public Potion(String name, int value) {
		super(name, value);
		// TODO Auto-generated constructor stub
	}

	public int getType () {
		return Treasure.GOLD;
	}

	/**
	 * applies only with potion
	 * @return life points
	 */
	public int heal() {
		return getValue() + D6.getD6().roll();
	}
}
```

Refactorización: Replace Type Code with Subclasses y Replace Conditional with Polymorphism

La clase Character también es un campo abonado para la refactorización. Entre Peter y John no tienen duda que lo primero que debe hacerse es lo mismo que se ha hecho con Treasure, que es una Replace Type Code with Subclasses. Pero ¿qué pasa con las funciones?

-Todas esas funciones con condicionales dan mucho el cante-dijo Peter.- Está claro que tenemos que cambiarlas y meter cada caso en la subclase correspondiente.

-Igual tendríamos que haber hecho el Replace Type Code with State/Strategy... - dijo John.

-Ummm, no lo creo, - contestó Peter – ese escenario tendría más sentido si el tipo puede cambiar, y me da a mí que un personaje de este juego nunca va a cambiar de tipo ¿No es así Simon?

-No, el transformismo no se contempla, esto es una laberinto de mazmorras y no Chueca. Si uno elige elfo, elfo se queda.

-Está claro, entonces en ese caso meteríamos una refactorización Replace Conditional with Polymorphism. - Concluyó Peter.- Cada subclase implementará la función correspondiente a su manera y quitamos esas condicionales de la superclase Personaje.

Así quedaría la clase Character:

```
package info.pello.tooomb;
```

```java
/**
 * @author Peter Class
 *
 */
public abstract class Character {

	protected String name;
	protected int strength;
	protected int speed;
	protected int intelligence;
	protected int life;
	public static final int BARBARIAN = 0;
	public static final int ELF = 1;
	public static final int DWARF = 2;
	public static final int WIZARD = 3;

	/**
	 * default constructor
	 * @param name
	 */
	public Character(String name) {
		this.name = name;
		initialize();
	}

	/**
	 * initializes attributes
	 */
	private void initialize () {
		this.strength = D6.getD6().roll();
		this.speed = D6.getD6().roll();
		this.intelligence = D6.getD6().roll();
		this.life = strength * 2;
	}
	/**
	 * factory method to create Character instances based on type
	 * @param type
	 * @param name
	 * @return
	 */
	public static Character newType (int type, String name) {
		switch (type) {
		case BARBARIAN:   return new Barbarian(name);
		case ELF:   return new Elf(name);
		case DWARF: return new Dwarf(name);
		case WIZARD:      return new Wizard(name);
		default: throw new RuntimeException("Incorrect Character");

		}
	}

	/**
	 * the character attacking roll
	 * @return  attack points
	 */
	public abstract int attack(Character character);

	/**
	 * the character defense roll
```

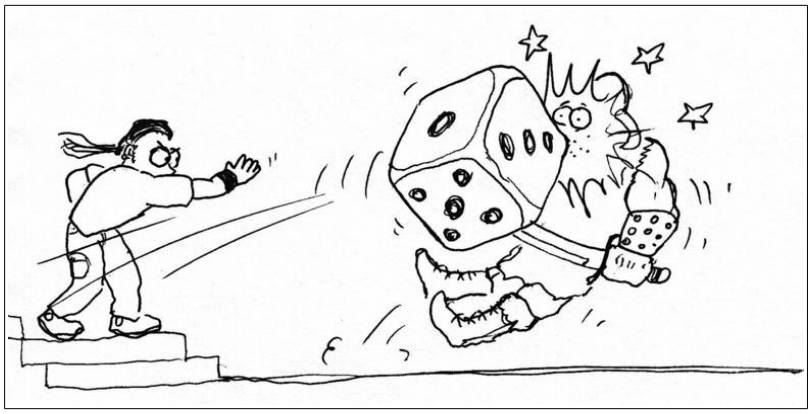
```
	 * @return defense points
	 */
	public abstract int defend(Character character);

	/**
	 * given type code returns Its name
	 * @return the name of player type
	 */
	public abstract String getType();

	/*
	 * shows character info
	 */
	public String info () {
		return this.name +" [strength=" + this.strength + ", speed=" +
this.speed
				+ ", intelligence=" + this.intelligence + ", type=" +
getType()
				+ ", life=" + this.life + "]";
	}

	/**
	 * @return the name
	 */
	public String getName() {
		return this.name;
	}

	/**
	 * takes some life points
	 * @param points to take
	 */
	public void takeLife(int points) {
		this.life -= points;
	}

	/**
	 * @return remaining life points
	 */
	public int getLife() {
		return this.life;
	}

}
```

Y así una de las subclases, por ejemplo Dwarf

```
package info.pello.tooomb;

/**
 * @author Peter Class , John Lisp
 *
 */
public class Dwarf extends Character {

	public Dwarf(String name) {
		super(name);
		// TODO Auto-generated constructor stub
	}
```

```
	/* (non-Javadoc)
	 * @see
info.pello.tooomb.CharacterType#attack(info.pello.tooomb.Character)
	 */
	@Override
	public int attack(Character character) {
		return this.strength + D6.getD6().roll();
	}

	/* (non-Javadoc)
	 * @see
info.pello.tooomb.CharacterType#defend(info.pello.tooomb.Character)
	 */
	@Override
	public int defend(Character character) {
		return this.strength + D6.getD6().roll();
	}

	/* (non-Javadoc)
	 * @see
info.pello.tooomb.CharacterType#getType(info.pello.tooomb.Character)
	 */
	@Override
	public String getType() {
		return "Dwarf";
	}

}
```

6.6. Sprint Review



El Sprint había llegado a su fin, lo cual no significaba que todo estuviera acabado. Al menos ese era el ambiente que se respiraba.

-Bueno Simon- comenzón John- como ProductOwner debes evaluar si hemos alcanzado los objetivos de este Sprint.

-Solamente en parte. Había seleccionado el tema del jugador y de los tesoros. Y al final no se han incluído en el juego.

-Pero vamos a ver-interrumpió Peter- las clases están hechas: Treasure y todas sus subclases. Y también la clase Character y las subclases. Todas ellas refactorizadas y

mejoradas.

-Creo que se a lo que se refiere Simon- dijo Float- Por muchas clases que hayamos creado y por muy estupendas que sean todavía no están integradas en el juego. Eso es lo que falta y no es poco.

-Bueno, eso es algo que tú nos debieras explicar-soltó Peter con tono de reproche.

-Eh, para el carro-Float- que parece que la clase principal es una tontería pero conforme añadís más elementos al juego se va complicando. Bastante he tenido con facilitar la entrada/salida.

John terció:

-Está claro que en el siguiente Sprint vamos a tener que dedicarnos a integrar todo. Viendo la clase principal se huele que vamos a tener que crear como poco una clase intermediaria entre lo que es el interfaz de usuario y el dominio. Algo que separe los dos mundos e incluso podriamos añadir clases solo para la lógica del juego.

-Aquí huele a patrón – dijo Peter – Además esa separación nos puede venir bien si un día el juego cambia de interfaz de usuario.

-No me entero muy bien de lo que decís – murmuró Simon – el caso es que no se han cumplido los objetivos del Sprint. Para el próximo en lugar de agregar más historias vamos a terminar correctamente las actuales.

6.7. Sprint Retrospective

-Bien, una vez más llegamos en la última parte del ciclo. Y ahora es cuando... hacemos una retrospectiva pero fijándonos en cómo estamos usando el Scrum.

-Ahora es cuando John revisa la aplicación que estamos haciendo de Scrum y nos va a laminar moralemente por no cumplir con el Sprint-tradujo Float.

-Oh cielos – dijo Peter- espera que me baje los pantalones. Procuraré no hacer fuerza. Please, nice and easy John.

-¡Pero que no es eso! ¡Cuánto daño han hecho años de metodologías basadas en el control y tanta jerarquía! Scrum, como toda metodología ágil es flexible, adaptable, y ahora se trata de ajustar un poco nada más.

-¿Vas a usar la metáfora del bambú, que si flexible pero fuerte a la vez? - se lamentó

Peter sujetando sus pantalones a media asta – Oh cielos. Nos vas a hacer un tracto rectal con una caña de bambú.

-Nooo. En el primer Sprint salimos todo exultantes porque cumplimos objetivos y nos creímos que ibamos a velocidad de crucero. Asumimos que podíamos abarcar mucho más trabajo en el siguiente Sprint y nos hemos equivocado. No pasa nada. Han surgido nuevas tareas y tendremos que ajustar las estimaciones para la próxima. Seguro que conforma hagamos más Sprint iremos acertando más.

7. Glosario

- ScrumMaster: en encargado de que se aplica Scrum correctamente y el facilitador de tareas.
- ProductOwner: el encargado de gestionar el Product Backlog
- Equipo Scrum: el equipo de desarrollo.
- Release Plan: el plan de entrega al inicio de un ciclo, que es cuando se decide qué objetivos se van a tratar de cumplir.
- Historia: un requisito, un objetivo, una característica del producto que se trata de conseguir.
- Product Backlog: la lista de Historias priorizadas para conseguir el product.
- Sprint Backlog: la lista de tareas a realizar en el Sprint
- Sprint: un ciclo de desarrollo en el que se intentan cumplir las tareas del Sprint Backlog
- Task: tarea que se lleva a cabo dentro del Sprint, que forma parte de una Historia.
- Burndown: gŕafico que muestra la relación entre historias
- Reglas: las normas que existen en Scrum, por ejemplo que no se puede cancelar un Sprint
- Artefactos: elementos que se generan al usar Scrum, como los Backlogs
- Bloques de tiempo: fases de un ciclo de Scrum, marcadas por las las reuniones.
- Incremento: cada nueva versión del producto con nuevas funcionalidades.
- History Points: puntos que se asignan a las historias.
- Velocity: el indicador de cuánto trabajo puede llevar a cabo un equipo en un Sprint. Si a cada historia se le asignan x puntos (History Points) y se cumplen varias historias en el sprint, la velocidad del equipo será la suma de esos puntos.
- Planning Poker: juego para estimar de forma grupal la dificultad de las tareas
- StakeHolders: gente implicada en el uso del proyecto que alimenta de información al ProductOwner
- Theme: un nombre que se usa para agrupar las historias.
- Vision: una frase que resume el objetivo del proyecto

8. Referencias

- Ken Schwaber & Jeff Sutherland: Scrum guide
- Kent Beck & otros: the Agile Manifesto.
- Roman Pilcher: Agile Product Management with Scrum
- Elizabeth Woodward, Steffan Surdek, Mathew Ganis: A Practical Guide to Distributed Scrum
- Kim Pries, Jon Quigley: Scrum Project Management
- Mike Cohn, Agile Estimating and Planning: Planning Poker.
- Jorge Serrano: Explicado Scrum a mi abuela
- Martin Fowler: Refactoring, improving the Design of Existing Code
- http://eugeniaperez.es/wordpress/?cat=33
- http://www.planningpoker.com
- http://www.scrumalliance.org/articles/87-writing-the-product-backlog-just-enough-and-just-in-time
- http://en.wikipedia.org/wiki/Test-driven_development
- http://en.wikipedia.org/wiki/Dependency_inversion_principle
- https://bitbucket.org/pello_altadill/tooomb : código de tooomb

- Fuckowsky: si te gustań las tristes historietas del mundo de la consultoría informática
- Wardog: lo mismo pero más orientado a sistemas
- BOFH: Bastard Operator From Hell, el origen de todas esas historias.
- Dilbert: míticas tiras cómicas sobre un departamento de programación y la fauna que le rodea
- Microsiervos: novela que ilustra el día a día, usos, costumbres y maneras de ser de los programadores. Deprimente por su realismo.