



Gestión del desarrollo software

Requisitos, Infraseres y otras bestias

Juan M. Casillas
14 de Marzo de 2025

hello_world\n
¡Bienvenidos!

t3chfest



Introducción





- Empecé bastante joven, tecleando listados [Apps, 1984] para poder jugar.
- Tuve claro bien pronto que me quería dedicar a los *cacharritos*:
- Universidad Carlos III de Madrid:
 - Ingeniería Informática Técnica de Gestión.
 - Ingeniería Superior (Redes y Sistemas).
 - GSyC.
 - Becario de aulas Linux.



whoami

Introducción

Empiezo a trabajar *fuertecito*:

- BSCH.
- Demasiado.com, EresMas, Periódicos digitales...
- AEMET.
- EADS.
- Actualmente: Senado de España.

Cuando no estoy trabajando:

- Desarrollo algunas cosillas por hobby
- Salgo al monte: Capitán Penurias



whoami

Introducción

Al final en estos 20 años he visto bastantes entornos:

- He trabajado tanto en sistemas como en desarrollo.
- He desarrollado *backend*, *frontend* y *middleware*.
- He visto varios modelos de desarrollo software.
- He trabajado en equipos grandes y pequeños.
- He ocupado distintos puestos.

En la mayoría de los sitios, se repiten los mismos patrones, sobre todo a la hora de desarrollar.



Antes que nada...

Introducción

Todo lo que voy a contar son experiencias y opiniones personales. Cada proyecto, cada equipo y cada empresa es un mundo. Lo que puede funcionar en un caso, puede que no funcione en otro. De ahí que sea muy importante experimentar. No todos los sitios son terribles, ni todos los compañeros *infrásteres*.

Es posible que se me escape alguna expresión “rara”. Pido disculpas por adelantado.

La idea es que la charla sea más o menos **interactiva**.

¿Por qué esta charla?

Introducción

“La informática es hacer más fácil el trabajo de las personas”

- Gestionar proyectos software puede ser interesante, divertido y además permitir fabricar algo útil.
- Sin embargo, la mayoría de las veces es simplemente justo al revés: estrés, frustración, mal ambiente laboral, malos productos.
- Debido generalmente a factores humanos (malas decisiones).
- Hemos normalizado *liarla parda*.
- Las cosas se pueden hacer si no “bien”, si de otra manera mejor.



Acto I

Entorno



¿Qué es desarrollar software?

Acto I: Entorno

- Fabricar un producto que solucione las necesidades del usuario.
- Las “necesidades del usuario” están definidas en forma de *requisitos*.
- Tenemos unas restricciones económicas, temporales y de recursos.
- Disponemos de ciertos recursos humanos y tecnológicos para fabricar el producto.
- Se suele percibir una remuneración.

En estos términos parece bastante sencillo. Sin embargo, en la mayoría de los desarrollos nos encontramos bastantes problemas de forma recurrente que hasta pueden impedir la entrega del producto.

¿Dónde está el truco?

Acto I: Entorno

En prácticamente todos los aspectos:

- Hay que conseguir unos *requisitos* “adecuados”.
- Después, hay que analizarlos y diseñar una solución.
- Que el usuario acepte dicha solución: (producto, coste y plazo).
- Ser capaces de fabricar la solución en esos términos.
- Que la solución fabricada sea como le dijimos al cliente.



Sigo sin ver el problema

Acto I: Entorno

El primer problema es no ver que son todo posibles problemas:

- Los requisitos los formula el usuario en sus propios términos, y tenemos que adaptarlos a nuestro idioma. A veces no sabe expresar lo que quiere (sabe de su negocio, no de software).
- Los requisitos se analizan para diseñar una solución (los famosos analistas).
- Tras el diseño un responsable **estima** costes (económicos, temporales y de recursos). (Ajustadas al usuario, no a la realidad).
- Fabricamos el producto en esos términos. ¿qué puede salir mal?.
- El usuario recibe un producto defectuoso. Se *disgusta* y con razón.

Encima el *responsable* dirá que “el proyecto ha sido todo un éxito”.



- No todos los desarrollos software son así de *chungos*.
- Hay sitios en los que se hacen las cosas bien.
- Pero en mi experiencia tengo que decir que he visto más mal que bien.
- El problema suele estar más *por arriba* que *por abajo* en la cadena de mando.



El equipo humano

Acto I: Entorno

- Lo normal es que el desarrollo software se haga por un equipo de personas.
- Cada persona tiene un rol (a veces varios).
- Cada rol tiene asociado una responsabilidad y una remuneración.
- Lo “normal” es asignar estos roles en base a la experiencia *demostrada* y a las capacidades de cada persona.
- En la realidad, imaginad por qué la charla lleva la palabra *infraseres y otras bestias* en el título.

Mucha remuneración = mucha responsabilidad.

Mucha responsabilidad = tomar decisiones que afectan a la gestión del desarrollo.



El jefe de todo esto

Acto I: Entorno

- El 99 % de los problemas se deben a roles con mucha responsabilidad tomando malas decisiones.
- (Rol con mucha responsabilidad = jefe).
- Sus motivaciones no tienen nada que ver con los objetivos del desarrollo.
- Suelen generar confrontamientos, mal ambiente laboral, tensión y problemas.
- No suelen tener perfil técnico. De hecho suelen jactarse de que no es necesario para “dirigir”.
- Dicho esto: hace falta tener un jefe (bueno) que se encargue de dirigir el proyecto.

Hay una película de Lars Von Trier que recomiendo encarecidamente.



Resumiendo

Acto I: Entorno

- Tenemos varias fases durante el desarrollo software que pueden salir mal.
- Hay que tomar decisiones importantes en cada fase, que dependen mucho de la experiencia y las capacidades.
- A veces las decisiones importantes las toma el que manda, y no el que sabe.
- Si trabajas en una empresa o en un departamento que se dedica al desarrollo software en exclusiva, estos pasos se repetirán *ad nauseam* en tu vida laboral. En caso de que el proceso funcione mal, la mayoría de las veces, a medio-corto plazo puede acabar influyendo en tu salud (depresión, *burn out*...)

¿Podemos hacer algo?

Acto I: Entorno

Depende del rol que tengas en el equipo y de tu organización.

- Un rol con mucha responsabilidad podrá tomar decisiones que afecten a nivel global el desarrollo.
- Un rol con poca responsabilidad afecta también al desarrollo del proyecto, pero en un nivel más técnico.
- Otras veces no se puede hacer nada (e.g. en el sector público). Solo queda aceptar la situación o cambiarse de trabajo.
- La experiencia se adquiere practicando. Los proyectos personales te permiten probar varios roles y aprender de los errores.

Acto II

Comprender al usuario



Qué quiere el usuario

Acto II: Comprender al usuario

“El usuario no sabe lo que quiere.” – Una jefa que tuve

- Sabe de su negocio, pero no sabe de software. Así que explica las cosas en sus términos. Pueden faltarle cosas por explicar que el considera “evidentes”.
- Suele querer una aplicación que le quite trabajo. A veces les es impuesta.
- Suelen ser reticentes al cambio.
- No suele ser gente técnica. Hay que adaptarse al usuario y sus “limitaciones”.



Dont's

Acto II: Comprender al usuario

Cosas que **no** hay que hacer:

- Tratar al usuario como si fuese *imbécil*. (“El usuario no sabe lo que quiere”).
- Capturar los requisitos a nivel de dirección y no de usuario.
- Explicar al usuario como funciona su negocio (ver primer punto).
- No recoger toda la información necesaria. (procesos, lo que cuenta y *lo que no*).
- Utilizar frases como “esto es imposible”, “no se puede hacer”, etc. Incluso antes de saber qué hay que hacer.



Do's

Acto II: Comprender al usuario

Cosas que **si** hacer:

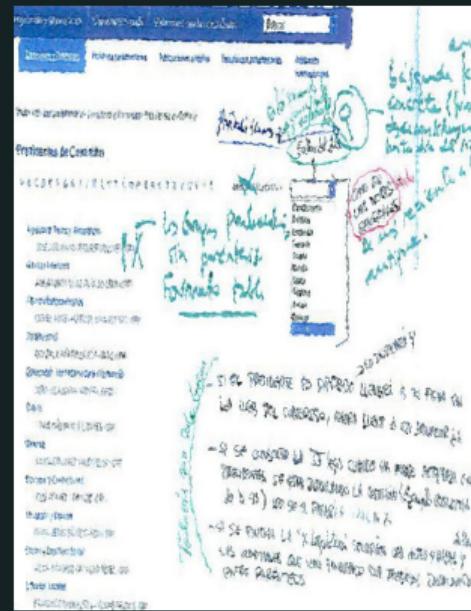
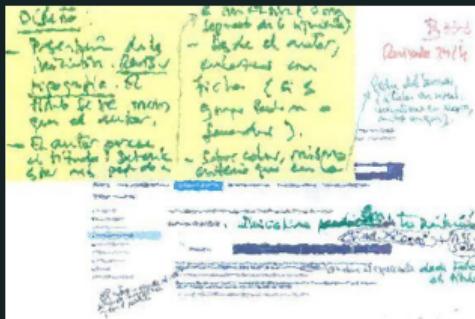
- Hablar con el usuario final que va a usar el producto.
- Que explique como hace las cosas, o mejor que nos lo enseñe.
- Tener las reuniones que sean necesarias para capturar todos los requisitos.
- Preguntar por ciertas cosas que pueden parecer “evidentes” (e.g. ¿tiene que poder imprimir?).
- Identificar y registrar los requisitos con algún tipo de herramienta.
- Intentar llegar a requisitos indivisibles.
- Empieza a usar una **herramienta de control de versiones** lo antes posible.

Si los requisitos son malos, el producto va a ser malo.

Por ejemplo...

Acto II: Comprender al usuario

Captura de requisitos para “una web”





Metodologías

Acto II: Comprender al usuario

Según la metodología se usan unas herramientas u otras para la captura de requisitos y modelado del diseño e implementación. Suele funcionar bien un enfoque híbrido entre *agile* [Mike Beedle, 2001] y *waterfall* [Schach, 1990]:

- Captura de requisitos detallada y bien documentada.
- Requisitos trazables.
- Mantenerse flexible en el diseño (y en la implementación). Siempre se escapa algo que hay que modificar después.
- La gestión del desarrollo software **no es una democracia**.
- Establecemos unos planes y una documentación.

La metodología es un medio, no un fin por si mismo. Elije lo que mejor se adapte al desarrollo y a tu equipo.



Agile vs Waterfall

Acto II: Comprender al usuario

Agile:

- Las metodologías *agile* funcionan en equipos pequeños, cohesionados y con experiencia.
- Son una fuente excelente de excusas para perder el tiempo en reuniones, sprints y daily's.
- Funcionan bien si siempre haces el mismo tipo de producto.
- No funciona bien ante problemas desconocidos.
- Muy complicado de estimar.
- Es “lo que se lleva”.



Agile vs Waterfall

Acto II: Comprender al usuario

Waterfall:

- Se usa desde 1970. Es “algo rígida”.
- Los errores se arrastran de principio a fin.
- Documentación para todas las fases y productos.
- Planificaciones y estimaciones detalladas.
- Lento. Requiere de varias iteraciones.
- Fácil de gestionar porque está todo definido y detallado.



Dicho esto...

Acto II: Comprender al usuario

- Dedica el tiempo que sea necesario a esta fase.
- Usa una herramienta para la gestión de requisitos.
- Elige una metodología:
 - La que mejor se adapte al proyecto y al equipo de desarrollo.
 - El equipo tiene que tener experiencia aplicándola.
 - Ajústala a las necesidades del proyecto y del equipo.

Acto III

Diseñar la solución

¿Que es diseñar?

Acto III: Diseñar la solución

Traducir las necesidades del usuario en elementos que puedan ser fabricados, para dar lugar a la solución.

- Muchísimas herramientas, metodologías y buenas prácticas.
- Se suelen encargar los *analistas funcionales*.
- Los analistas son programadores que han ascendido en el escalafón.
- A muchos de ellos no les gusta, o no saben programar.
- Al no tener experiencia programando, suelen diseñar... *engendros*.



El precio del diseño

Acto III: Diseñar la solución

Un buen diseño es clave para poder implementar un buen producto, pero el diseño no es el producto.

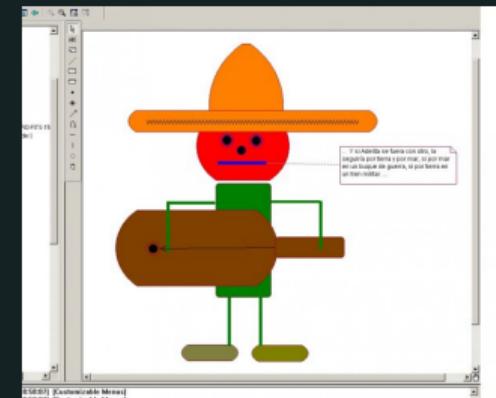
- Es interesante disponer de una herramienta de modelado.
- El diseño permite entender los requisitos y plantear su implementación.
- Hace las estimaciones más realistas.
- Ir de lo global a lo particular.
- El detalle depende de la experiencia del equipo.



Coste / Beneficio

Acto III: Diseñar la solución

- La fase de diseño está genial porque como Powerpoint, lo soporta todo.
- No hay que quedarse corto (“the system”) ni pasarse.
- El objetivo es fabricar un producto usando el diseño.
- **UML** Es una buena opción siempre que se adapte al proyecto. [James Rumbaugh, 1997]
- Hay buenas guías [Fowler, 2003].
- Si usas UML, elige una buena herramienta para gestionar el modelo.
- El diseño cuesta tiempo.



Modelado creativo / EADS



Afinando el diseño

Acto III: Diseñar la solución

- Conviene pensar cómo vamos a fabricar el producto: La tecnología impone restricciones de diseño.
- Anticipa que habrá un par de iteraciones (o más): coste temporal.
- El diseño tiene que ser comprensible y razonablemente completo.
- Conviene dividirlo a nivel lógico para poder manejarlo.
- Trazar los requisitos a nivel de diseño es un *plus*.
- Teniendo el diseño, podemos empezar a estimar el **coste** de fabricación.



Estimaciones

Acto III: Diseñar la solución

- Sin recursos infinitos (e.g. sector público) es necesario estimar costes.
- Un buen diseño y un equipo con experiencia permiten realizar estimaciones precisas.
- Es mejor estimar con cierta holgura: *shit happens*.
- Si vas a innovar, planifica un 40 % más del tiempo para esas tareas.
- Estimar económicamente es complicado. No estimes en *horas/hombre*.
- Plantea estimaciones honestas, no lo que quiere oír el cliente.



En resumen

Acto III: Diseñar la solución

- La fase de diseño es crítica. Hay que dedicarle tiempo pero no *todo* el tiempo.
- Modela lo imprescindible.
- Tener todo el producto modelado a bajo nivel es genial, pero carísimo.
- Conviene centrarse en diseñar bien las partes críticas y complejas.
- El modelo tiene que estar **actualizado**.
- El producto se genera a partir del modelo **y no al revés**.
- Traza los requisitos al modelo.
- Pensar en como se va a implementar ayuda en el diseño.
- Si piensas que solo con el diseño se puede subcontratar la implementación: buena suerte.



Acto IV: Fabricar el producto



Manos a la obra

Acto IV: Fabricar el producto

El cliente aceptó plazos, costes y diseño. Toca fabricar el producto.

- Muchos responsables de proyecto piensan que esta fase no tiene “valor añadido” y puede ser incluso subcontratada.
- En 1976 con metodología *waterfall* quizás tuviese algo de sentido subcontratar. Las razones actuales son costes o incapacidad.
- No se trata de *traducir* las instrucciones dadas por los **analistas** en código sino de *interpretar* las instrucciones de diseño para implementar la solución de la **mejor manera** posible.



Inciso

Acto IV: Fabricar el producto

- Pensar que el desarrollo se basa en diseñar y externalizar la implementación, es erróneo.
- La “gracia” de desarrollar soluciones es ser capaz de dar respuesta a las necesidades de un modo eficaz y eficiente.
- **Diseñar e implementar va junto.** No se puede diseñar bien si no sabes implementar bien, y al revés.
- Los arquitectos no ponen ladrillos. Pero los arquitectos que *saben* poner ladrillos hacen mejores diseños.
- A veces la persona no sabe poner ladrillos pero la ascienden a arquitecto.
- Fabricar productos \neq escribir código.
- Cuidado con las *software factories* que venden *churreras*.



Herramientas

Acto IV: Fabricar el producto

Para fabricar productos hacen falta *herramientas*. A nadie se le ocurriría fabricar un coche solo con un martillo, ¿verdad?. Han salido algunas:

- Herramienta de control de versiones.
- Herramienta de gestión de requisitos.
- Herramienta de gestión del modelo de diseño.

A día de hoy, no utilizar una herramienta de control de versiones (e.g. [git](#)) es una temeridad. *no version control, no party*.



Herramientas, el mínimo

Acto IV: Fabricar el producto

Como mínimo:

- Herramienta para escribir el código (**IDE**).
- Herramienta de gestión de incidencias.
- Herramienta para generar el producto y despliegue.
- Herramienta para ejecutar las pruebas.

Delicatessen:

- Herramientas de depuración / profiling.
- Herramientas de integración continua.
- Herramientas de análisis de calidad.



Más herramientas

Acto IV: Fabricar el producto

Herramientas \neq productos. A veces un par de *scripts* sirven. otras veces hay que comprar un producto, depende del proyecto. Las herramientas son el medio, no el fin:

- Si es posible, elegir herramientas *open source* que estén mantenidas.
- Menos es más, siempre. Muchas herramientas no implica mucha productividad.
- Las herramientas tienen que **facilitar** el trabajo.
- Tienen que adaptarse al flujo trabajo, no al revés.
- No adoptes nuevas herramientas durante el desarrollo.



Frameworks

Acto IV: Fabricar el producto

En general: **FRAMEWORKS, NO.** Ahora bien:

- Si el framework es *open source*, está mantenido y lo usa un número significativo de usuarios.
- Si soluciona un problema **real** existente.
- Si **facilita** el desarrollo.
- Si **conoces** como funciona el framework por dentro y eres capaz de **modificarlo** si surge la necesidad.

Reinventar la rueda y fabricar un framework propio suele ser tentador y una forma de tener dos problemas: el desarrollo de productos y el mantenimiento del framework para desarrollar productos.



Flujos de trabajo

Acto IV: Fabricar el producto

Para poder trabajar, hay que definir **cómo** trabajar:

- Definir flujos de trabajo sencillos, documentados y que se entienden.
- Definir flujos para las principales tareas, por ejemplo:
 - “Cómo se fabrica el producto desde el código fuente”
 - “Cómo se despliega el producto en desarrollo”
 - “Cómo se prueban los componentes”
 - “Cómo se cambia de versión”
- Habrá unos flujos que siempre serán los mismos, y otros por proyecto.
- No definir muchos flujos.

Es aconsejable centralizar la documentación en algún gestor accesible vía navegador. La documentación tiene que estar **actualizada**.



Entornos

Acto IV: Fabricar el producto

El producto **siempre** hay que probarlo:

- Pruebas unitarias, ejecutadas por la herramienta de pruebas.
- Pruebas de funcionalidad / integración. Pueden ser ejecutadas por herramientas o normalmente a mano.

Es necesario tener **siempre** al menos dos entornos:

- **Desarrollo**. Un entorno que es **igual** que producción, pero con datos *falsos* (que no malos).
- **Producción**. Donde se despliega el producto final. A efectos de desarrollo **no es accesible**.

Estos entornos tienen que estar **separados** y ser **independientes**, pero **semejantes**, de modo que las pruebas que se realicen tengan sentido.



Dicho esto...

Acto IV: Fabricar el producto

Con el diseño las herramientas los flujos de trabajo y los entornos sólo resta sentarse y *picar* el producto. Pero faltan un par de *detalles*:

- Cómo lo vamos a hacer.
- En qué lo vamos a hacer.

En mi experiencia, la mejor respuesta es **usar lo que mejor se adapte a las necesidades del proyecto**. Pero no suele ser así.



Aquí se usa X

Acto IV: Fabricar el producto

La mayoría de los equipos de desarrollo **siempre** desarrollan de la misma manera y en el mismo lenguaje.

- Existen múltiples paradigmas según la naturaleza del proyecto, pero aquí se usa X
- Existen múltiples lenguajes que se adaptan a las características del proyecto, pero aquí se usa Y
- La arquitectura y el framework *el de siempre*, por supuesto.
- ¿La razón? Alguien dijo en su momento que se hacía así.

¿Os suena el Golden Hammer?



Que se ve por ahí

Acto IV: Fabricar el producto

- Lo más extendido es JAVA. Algunos proyectos se hacen en C++
- En entorno web, se suele usar JAVA, Javascript y PHP.
- El paradigma más extendido suelen ser OOP.
- Se usan frameworks bien desarrollados *in house*, bien externos (e.g. spring, hibernate, vue, react, angular, node...)
- Las cosas se suelen hacer una forma determinada porque alguien tuvo *la visión*.
- Suele haber una falta de visibilidad completa del proyecto.
- Suele haber bastante *lío* con la gestión de dependencias.
- La generación del producto es algo casi *mágico*.



Poniendo el foco

Acto IV: Fabricar el producto

Es fácil *despistarse*. El objetivo es **fabricar** el producto. Qué suele funcionar:

- Implementar lo primero las pruebas.
- El ciclo *implementar-probar* poco a poco la funcionalidad.
- Las implementaciones complejas conviene **documentarlas**.
- La documentación hay que **mantenerla actualizada**. Mejor poco y bien que mucho y mal.
- Despliegues frecuentes en desarrollo. Lo ideal sería desplegar en local (los contenedores son muy útiles).
- Tener **todo** el desarrollo bajo control de versiones. *Workflows* sencillos.



Por último

Acto IV: Fabricar el producto

- Un enfoque de gestión híbrido entre *waterfall* y *agile* funciona. Reunirse está bien, pero no hace falta que sea cada 15 minutos. No hace falta generar un acta por cada método implementado.
- Si se trazan los requisitos al código se puede medir lo que va implementado, y qué componentes afectan a qué funcionalidad.
- Si surgen dificultades conviene notificarlas **pronto** para poder proponer soluciones antes de que se vuelvan críticas.
- Implicar al usuario en algunas revisiones. Así obtenemos su *feedback* y además siente que le hacemos *casito*.

Solo resta dar vueltas a la *manivela* hasta que el proyecto esté completado.

Epílogo





Entrega del producto

Epílogo

Al final del desarrollo el cliente recibe su producto. Si todo ha ido bien:

- El cliente estará contento.
- El producto funcionará según sus necesidades.
- A veces contratan mantenimiento por lo que hay que asignar algunos recursos.
- “El proyecto ha sido un éxito”



En caso contrario

Epílogo

Pero si no ha ido bien:

- El cliente estará enfadado y querrá compensaciones, bien en funcionalidad, bien en forma de descuento.
- A veces está tan enfadado (o harto) que decide cancelar el proyecto.
- Otras veces se redefinen algunos requisitos y se vuelve a realizar una iteración completa a ver si esta vez *suena la flauta*.
- Quizás haya algunas reuniones para encontrar *responsables*.

En general iterar *echando otra monedita* sin saber que ha pasado es mala idea. Conviene hacer un análisis de las cosas mejorables para la próxima vez.



Recapitulando

Epílogo

- Gestionar un desarrollo software es complicado y tiene mucha incertidumbre.
- La gestión del equipo es **difícil**.
- No hay dos proyectos iguales. No existe la *churrera mágica*.
- Hay que adaptarse al cambio y evolucionar.
- El valor de un equipo de desarrollo son las personas que lo componen.
- Si coordinas un equipo tienes que *predicar con el ejemplo*.
- Las herramientas, metodologías y paradigmas no hacen el trabajo.
- Aunque parezca increíble, este trabajo *mola*.



Madurez del equipo

Epílogo

Los equipos de desarrollo pasan por distintos estados de madurez:

- En su *infancia* no tienen experiencia y hacen poca cosa con mucho esfuerzo.
- A medida que van madurando, comienzan a producir.
- En la adolescencia se revelan y les suele dar por reescribirlo todo y desarrollar sus propios frameworks y herramientas. Están muy dispersos y producen regular.
- Algunos equipos consiguen madurar y entran en una etapa *industrial*. Planifican bastante bien y cumplen con los plazos.

Dada la naturaleza de este trabajo, los equipos de desarrollo no tienden a alcanzar la madurez ya que la gente se suele cambiar de empresa bastante.



Entornos público vs privado

Epílogo

La principal diferencia es *quién paga las facturas*.

En el sector privado malgastar el dinero no es una opción. El desarrollo requiere estimaciones precisas y presupuestos ajustados. Siempre se producen *fallos*, pero se atajan con despidos y rescisiones de contrato.

En el sector público se junta “*que el dinero no es de nadie (sic)*” y que es complicado reclamar responsabilidades. Esto da lugar a proyectos con sobre-costes y fuera de plazo.

Además la selección de los puestos de responsabilidad está basada en criterios como la *antigüedad*, *afinidad* o peor aún, por *libre designación* sin tener en cuenta experiencia ni capacidades. Esto no significa que lo público sea malo, al contrario: la mayoría de los miembros de los equipos son gente formada y con experiencia que ha superado unos procesos selectivos complicados.



Apéndice

Bestiario



Infraseres

Bestiario

Para mi un *infraser* es una persona con pocos valores morales, que se aprovecha de otras personas. Durante estos años laborales me he encontrado personas que sin ser *mala gente*, si que interpretan un rol laboral de *infraser*. Lo curioso es que estos roles se repiten en distintas empresas.



Infraseres: “Amado Líder”

Bestiario

El jefe de todo esto. He conocido un par de ellos buenos... eso si, en 20 años.

Son autoritarios, sin formación técnica pero con capacidad organizativa y son el principal problema tecnológico. Se rodean de gente que les da la razón, por lo que los proyectos suelen fracasar con estrépito. Les gusta hacer depuración de responsabilidades e *informes de rendimiento*.

Algunos se centran en sus cosas, dejando el rumbo tecnológico de la empresa a la deriva (o a manos de un subalterno) y otros se empecinan en seguir con la informática del 1976 ad infinitum: “no despiden a nadie por comprar IBM”.



Infraseres: Acólito

Bestiario

Por debajo del amado *líder* solemos encontrar buen número de acólitos. Básicamente aplauden y ensalzar las decisiones tomadas por el líder. Dado que no cuestionan sus decisiones, suelen ocupar puestos de cierta responsabilidad para seguir ejerciendo “el poder” aguas abajo y controlar a la “plebe”.

Son especialistas en reunirse y tienen muy poca capacidad técnica. Están escogidos entre el “vulgo” por haber hecho algo excepcional para el negocio (desde el punto de vista del amado *líder*). Son los típicos del “¿como vamos?” y “su rendimiento es insuficiente”.



Infraseres: Responsable-Ácolito

Bestiario

Han encontrado refugio al encargarse de alguna tarea abstracta como *gestor de prereleases* o *responsable del middleware de respaldo*, gestionar una herramienta que no se usa, cosas así. Su trabajo es importantísimo, pero no aporta nada para el desarrollo del producto.

No suelen ser muy hábiles técnicamente, sin embargo al no estar implicados en el desarrollo (o mantenimiento) pueden permitirse probar productos, asistir a conferencias, etc. Suelen tender a evangelizar con lo último que han visto en internet (e.g. IA, cloud, contenedores, agile, scrum, etc.)



Inciso

Bestiario

Curiosamente, llega un nivel en la jerarquía en la que desaparecen los infrásteros, para dar paso a la gente del equipo que *realmente* trabaja. En este nivel habrá más o menos tensiones personales (las normales en un equipo de trabajo) pero lo habitual es un espíritu trabajador y colaborativo.

Cuando los proyectos fracasan, los infrásteros suelen buscar responsables a este nivel, siendo lo más damnificados en las decisiones de reestructuración.



Infraseres: Perfiles I

Bestiario

Además de infraseres a nivel personal existen *roles*:

- **Arquitectos.** Solo diseña: no sabe programar, no analiza requisitos, nada. Hace diseños. No se le puede cuestionar: es el arquitecto. Esta bien considerado. Suele ser necesario rediseñar sus diseños para poder implementarlos.
- **Programadores.** El *yang* del arquitecto: solo programa. Hay que estar alerta porque suelen generar código bastante *barroco*. Suelen saltarse los flujos de trabajo.
- **Especialistas.** En algún momento adquirió un conocimiento en algo puntual y sólo se dedica a eso. Se niega a formarse en otra cosa porque sólo sabe de lo suyo.



Infraseres: Perfiles cont.

Bestiario

- **Visionarios.** Tiene *la visión*. Sabe a donde vamos. Recomienda todo lo que encuentra ya que será *la solución*, pero se queda en el plano teórico. Suelen identificarse como *visionarios*, *tecnólogos* o *gurúes del cambio*. Acaban promocionados a *Responsable-Acólito*.
- **Evangelistas.** Existen metodológicos, documentales, de calidad... etc. Encargados de ensalzar las ventajas de adoptar sus enseñanzas. Los problemas surgen al no adoptar sus mandamientos. Van cambiando de discurso siguiendo las tendencias de la moda.
- **Fatalistas.** Predican el juicio final. “no vamos a terminar el proyecto”, “no va a funcionar”, “nos va a despedir”, etc. Son pesimistas por naturaleza. Tener estos *infraseres* en el equipo genera un desgaste y una sensación de fracaso importante. suelen ser personas muy quemadas.



Otras bestias

Bestiario

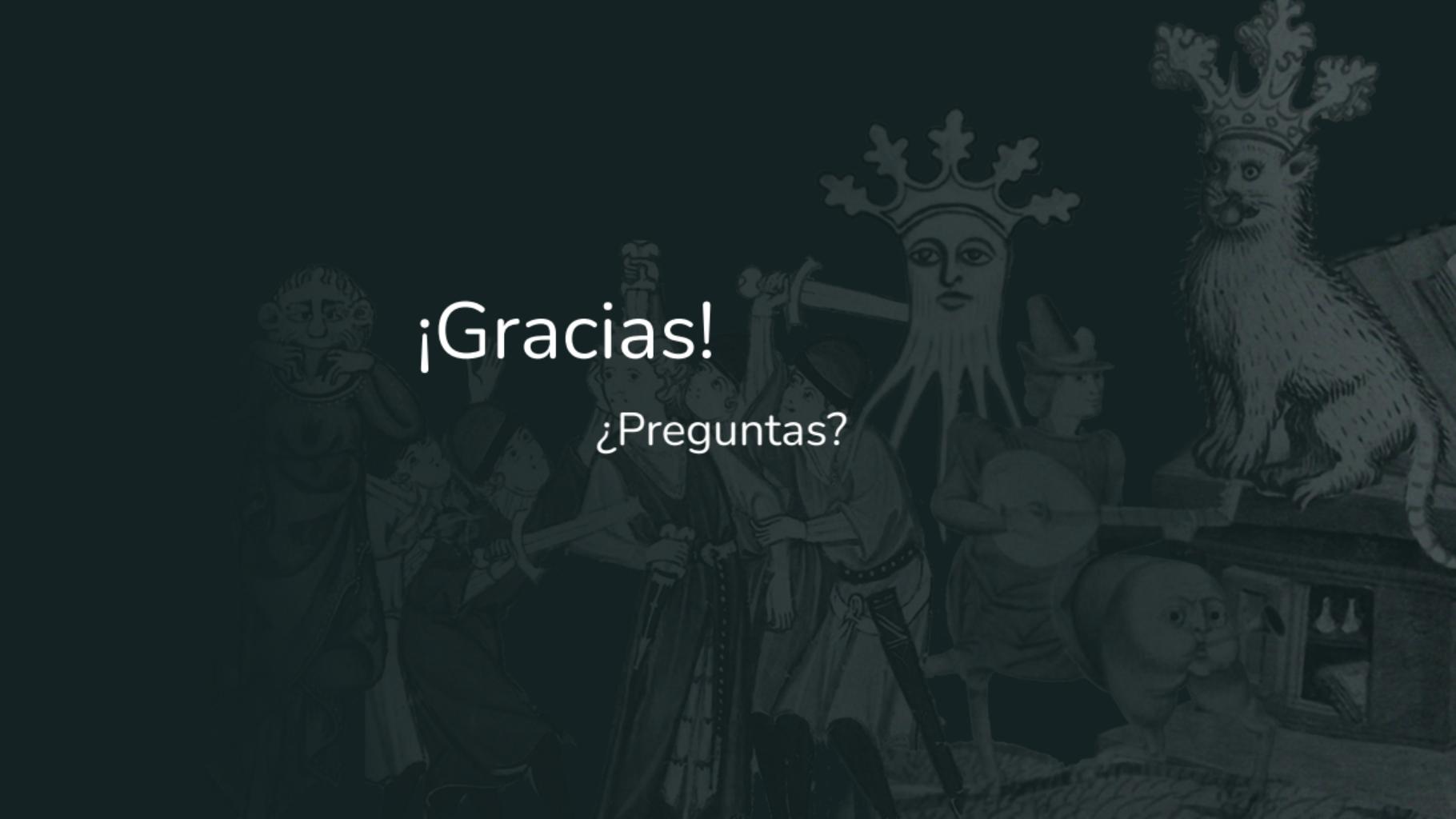
- Plazos.
- Presupuestos.
- Recursos.
- Tecnologías obsoletas.
- Tradición: “Siempre lo hemos hecho así”.
- Innovación: Reinventar la rueda.
- ¡Somos Especiales! ¡Nada nos sirve!.
- IA para todo.
- La nube. Adiós a la infraestructura.



Otras Bestias: subcontratación

Bestiario

- Se pierde el control de **cómo** se hacen las cosas (*know-how*).
- Se queda a expensas de la externalización para el futuro mantenimiento.
- Costes y plazos impuestos externamente.
- ¿Quién implementa las pruebas?.
- ¿Quién prueba?.
- ¿Quién mantiene?.
- No es barato.
- Es fácil entrar, difícil salir.



¡Gracias!

¿Preguntas?

Referencias

-  Apps, V. (1984).
40 juegos educativos para el spectrum.
<https://archive.org/details/40-juegos-educativos-para-el-spectrum>.
-  Fowler, M. (2003).
UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition).
The Addison-Wesley Object Technology Series. Addison-Wesley Professional, 3 edition.
MR: UML(2) schnell eingeführt.
-  James Rumbaugh, Grady Booch, I. J. (1997).
Unified modeling language spec 1.1.
<https://www.omg.org/spec/UML/1.1>.
-  Mike Beedle, Martin Fowler, R. J. (2001).
The agile manifesto.
<https://agilemanifesto.org/>.
-  Schach, S. R. (1990).
Software engineering.
Aksen Associates, USA.