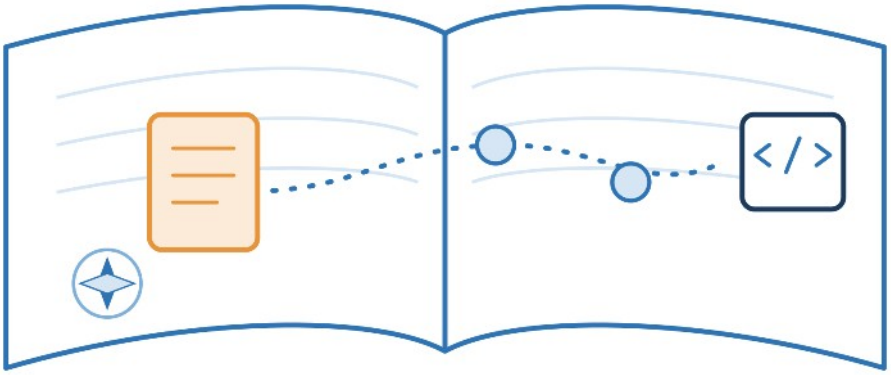
Scrum Manager
Uncovering better ways

Guía

Spec Driven Development

en equipos ágiles

Desarrollo de los temas del State of the art



Actualizado a junio de 2026

Scrum Manager® · Skill Arena

Sobre este documento

Este documento desarrolla en profundidad los temas del mapa de referencia (State of the art) para dirigir el desarrollo de software con IA mediante especificaciones (Spec Driven Development) en equipos ágiles, que está disponible en: Skill Arena.

Úsalo como material de consulta para mantener y contrastar tu práctica con el conocimiento profesional actual.

Se complementa con la plataforma de entrenamiento y evaluación en Skill Arena. En el área Spec Driven Development en equipos ágiles puedes realizar pruebas de entrenamiento para contrastar y mejorar tu nivel de conocimiento y si lo deseas también puedes obtener un diploma que acredita curricularmente la solvencia y vanguardia profesional en esta área.

Cada capítulo desarrolla uno de los temas del mapa, indica su estado de adopción y se cierra con las capacidades que un profesional debe ser capaz de demostrar, los errores frecuentes y referencias para profundizar.

Estado del conocimiento

El conocimiento sobre Spec Driven Development evoluciona de forma extremadamente rápida: método, herramientas y prácticas se renuevan en cuestión de meses, y aunque algunos fundamentos ya están asentados, buena parte del ecosistema sigue en plena formación. En los distintos apartados del documento, las etiquetas (ESTABLECIDO, EN CONSOLIDACIÓN, EMERGENTE) ayudan a identificar la madurez de cada concepto:

ESTABLECIDO consenso asentado; conocimiento que se da por necesario.

EN CONSOLIDACIÓN gana adopción con rapidez; aún no universal pero ya relevante.

EMERGENTE frontera reciente; alta relevancia y alta volatilidad.

Índice

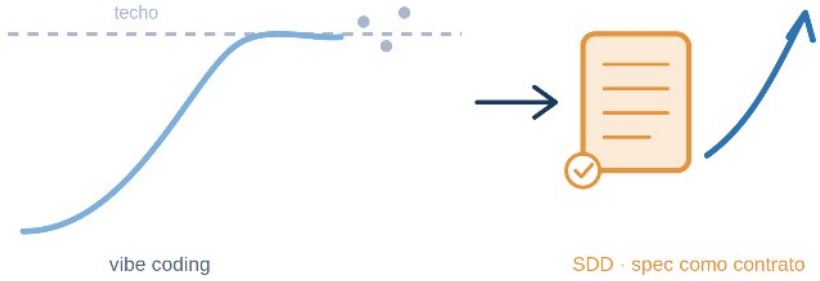
Capítulos del desarrollo, en el orden del State of the art.

BLOQUE A · EL PARADIGMA: POR QUÉ SDD

Capítulo 1. Del vibe coding al desarrollo dirigido por especificación

ESTABLECIDO

El vibe coding es productivo para prototipos pero tiene un techo; SDD es la respuesta metodológica a sus límites.



El vibe coding tiene un techo; SDD escribe una spec como contrato antes de generar código.

Introducción

En febrero de 2025, Andrej Karpathy popularizó el término «vibe coding» para describir una forma de programar con IA: describes lo que quieres en lenguaje natural, el modelo genera el código, lo pruebas y, si falla, le pegas el error hasta que sale del paso. No hay diseño previo, no hay especificación: el prompt es la intención, el código es la respuesta y la iteración conversacional es el método. Este capítulo explica por qué ese enfoque funciona hasta cierto punto y por qué SDD surge como respuesta a sus límites.

1.1 Dónde funciona el vibe coding

Conviene reconocer su mérito: para prototipos, exploración de ideas, herramientas internas rápidas y todo aquello donde importa más llegar pronto que llegar bien, el vibe coding es extraordinariamente productivo. El problema no es el vibe coding en sí, sino que tiene un techo que aparece antes de lo que la mayoría espera.

1.2 Dónde deja de funcionar

Las mediciones del sector son contundentes. Un ensayo controlado de la organización METR (julio de 2025) con desarrolladores experimentados encontró que, usando IA, tardaban un 19 % más en completar sus tareas, pese a creer que iban un 20 % más rápido: la percepción de velocidad no coincidía con la realidad medida. Sobre calidad, el informe de Veracode de 2025 halló que en el 45 % de los casos el modelo introducía una vulnerabilidad cuando podía elegir entre una solución segura y una insegura; y el análisis de GitClear mostró que, por primera vez en 2024, el código duplicado superó al refactorizado.

El coste real es cognitivo. Cuando escribes código, tu memoria de trabajo retiene el contexto de lo que acabas de escribir. Cuando revisas código generado, la carga de comprensión es distinta y se acumula con cada iteración. La paradoja del vibe coding es que la herramienta diseñada para ahorrar esfuerzo termina generando esfuerzo de verificación, más agotador que el de construcción.

1.3 La respuesta: Spec-Driven Development

SDD invierte el orden: en lugar de saltar al código, se produce una especificación que define qué se construye, cómo y en qué pasos, y solo después de revisarla y aprobarla se implementa. No es una

herramienta ni un framework: es una metodología, una forma de organizar el trabajo que puede implementarse con distintos instrumentos. La definición de Böckeler es escueta: «SDD significa escribir una spec antes de escribir código con IA». El cambio de fondo: donde el vibe coding trata al agente como interlocutor conversacional, SDD lo trata como ejecutor que recibe un contrato claro.

Conviene despejar malentendidos: SDD no es documentar después de construir (el orden importa), no es escribir pliegos de 200 páginas (una buena spec es inteligente, no larga) y no exige abandonar la agilidad (es, de hecho, su aplicación coherente cuando parte del equipo son agentes, como desarrolla el capítulo 3).

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar qué es el vibe coding y en qué contextos es la herramienta adecuada.
- Reconocer los límites del vibe coding a escala: la divergencia entre velocidad percibida y real, y los problemas de calidad documentados.
- Definir SDD como metodología (no herramienta) y enunciar su premisa: la spec precede al código.
- Distinguir SDD de la documentación posterior, de los pliegos exhaustivos y de un supuesto abandono de la agilidad.

Errores y antipatrones frecuentes

Tratar el vibe coding como enemigo a erradicar. Es la herramienta correcta para prototipos y exploración; SDD lo complementa donde las vibes no escalan, no lo sustituye en todo.

Confundir velocidad percibida con productividad. La sensación de ir rápido con IA no equivale a ir rápido; conviene medir, no suponer.

Creer que SDD es una herramienta que se instala. Es una forma de organizar el trabajo; las herramientas la soportan, no la definen.

Para profundizar

· Martin Fowler — Understanding Spec-Driven Development

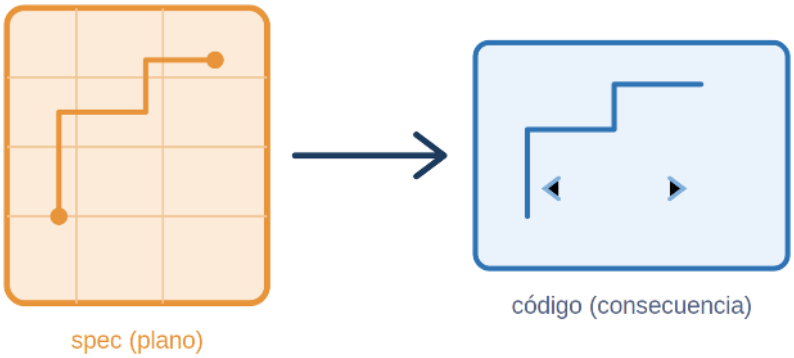
· Addy Osmani — How to write a good spec for AI agents

BLOQUE A · EL PARADIGMA: POR QUÉ SDD

Capítulo 2. La spec como artefacto primario del desarrollo

ESTABLECIDO

El centro del trabajo se desplaza de escribir código a especificar con claridad qué se construye y por qué.



La spec es el plano; el código es la consecuencia que se deriva de ella.

Introducción

Addy Osmani lo resume: «la IA no es el cuello de botella; tu spec lo es». Durante décadas, la competencia central del profesional del software fue escribir código. SDD propone que esa competencia se desplaza: sin desaparecer —entender código sigue siendo necesario para revisar y decidir—, deja de ser el acto central. Lo que ahora hace productivo es la capacidad de especificar con claridad qué construir, por qué, bajo qué restricciones y con qué criterios de éxito.

2.1 El código como consecuencia

La analogía es la arquitectura: un arquitecto no coloca ladrillos, diseña planos que otros ejecutan. La calidad del edificio depende de la calidad de los planos. La spec es el plano; el código es el edificio. Cuando algo falla, la primera pregunta no es «¿qué bug tiene el código?» sino «¿qué no especificamos bien?». El agente es literal: cada decisión no especificada es una decisión que toma por su cuenta, y cada una es un punto potencial de divergencia entre lo que querías y lo que obtienes.

2.2 La spec como contrato, no como documento de mando

Lo que distingue SDD de la simple documentación previa es que la spec funciona como contrato entre las partes del sistema. Los humanos aprueban el contrato en las puertas de fase, los agentes lo ejecutan, y lo que se entrega es código que cumple el contrato. La apuesta de fondo: contratos explícitos en la granularidad correcta permiten que el desarrollo dirigido por IA a escala de equipo avance más rápido, no más lento.

2.3 Qué cambia para cada rol

El desplazamiento afecta a todo el equipo, no solo a quien programa:

- **Product owner / product architect:** la exigencia de precisión sube. Los criterios de aceptación pasan de guía para la conversación a contrato de ejecución.
- **Desarrollador / product builder:** la competencia se desplaza del lenguaje de programación al diseño de specs y la orquestación de agentes. Se parece más a un tech lead que a un programador.

- **Agile enabler:** aparecen las puertas de aprobación como puntos de inspección que facilitar sin que se vuelvan cuellos de botella.
- **Stakeholders:** ganan visibilidad: la spec es legible por no técnicos en una medida que el código nunca lo fue.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar el desplazamiento de la competencia central del código a la especificación.
- Aplicar la analogía del plano y el edificio para razonar sobre la relación spec-código.
- Distinguir la spec como contrato (aprobado en puertas, ejecutado por agentes) de un documento de mando.
- Anticipar cómo cambia el trabajo de cada rol del equipo, incluidos los no técnicos.

Errores y antipatrones frecuentes

Creer que el código deja de importar. Hay que revisarlo, probarlo y validarlo; lo que cambia es la dirección causal: la spec produce el código, no al revés.

Especificar con la imprecisión del desarrollo tradicional. Una historia que un humano interpretaría con sentido común, un agente literal la implementará al pie de la letra o inventará lo que falte.

Tratar la spec como documento de mando unidireccional. Funciona como contrato co-creado y aprobado, no como una orden que se lanza al agente.

Para profundizar

· Addy Osmani — How to write a good spec for AI agents

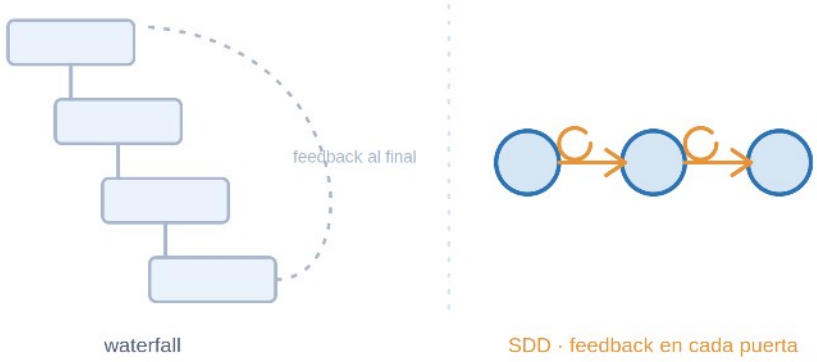
· Martin Fowler — Exploring Gen AI

BLOQUE A · EL PARADIGMA: POR QUÉ SDD

Capítulo 3. SDD y agilidad: no es el regreso de waterfall

EN CONSOLIDACIÓN

SDD usa fases y puertas, pero difiere de waterfall en alcance, feedback y reversibilidad.



No es waterfall: SDD difiere en alcance, feedback en cada puerta y reversibilidad.

Introducción

Cualquier profesional ágil pensará, tarde o temprano, que SDD «suena a waterfall»: fases secuenciales, puertas de aprobación, documentación antes del código. La objeción es razonable y resolverla es lo que permite adoptar SDD sin traicionar la agilidad. Este capítulo la aborda de frente.

3.1 Qué comparte con waterfall y qué no

Comparte la secuencialidad: requisitos antes que diseño, diseño antes que tareas. Pero difiere en tres ejes decisivos:

1. **Alcance.** Waterfall aplicaba la secuencia a todo el proyecto (meses por fase). SDD la aplica a cada incremento: el ciclo completo puede medirse en horas o días.
2. **Feedback.** En waterfall llega al final. En SDD, en cada puerta de aprobación; cada ciclo es un ciclo de aprendizaje completo.
3. **Reversibilidad.** Volver atrás en waterfall es caro. En SDD, las fases previas a la implementación son documentos de texto: modificarlos es trivial frente a reescribir código.

3.2 Una nueva categoría: documentación operativa

La documentación tradicional era informativa (transmite conocimiento entre personas) o administrativa (satisface requisitos del proceso). El Manifiesto Ágil cuestionó con razón su prioridad sobre el software funcionando. SDD introduce una tercera categoría: documentación operativa, la spec que el agente consume directamente para generar código. No informa a una persona ni satisface un trámite: es el input del proceso de producción. Por eso la objeción del Manifiesto no le aplica del mismo modo.

3.3 SDD como consecuencia de los principios ágiles

Más allá de refutar la objeción, SDD puede leerse como la aplicación coherente de los principios ágiles cuando parte del equipo son agentes: feedback rápido (puertas), entrega incremental (una spec por incremento), colaboración alrededor de artefactos legibles (la spec) y respuesta al cambio (es más barato modificar una spec que reescribir código). Thoughtworks lo sitúa en «Assess»: aporta valor con requisitos ambiguos, equipos distribuidos o necesidad de trazabilidad.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Articular las tres diferencias entre SDD y waterfall: alcance, feedback y reversibilidad.
- Explicar la categoría de documentación operativa y por qué la objeción del Manifiesto no le aplica igual.
- Argumentar SDD como aplicación coherente de los principios ágiles en equipos con agentes.
- Situar el juicio de Thoughtworks sobre cuándo SDD aporta valor y cuándo el overhead no compensa.

Errores y antipatrones frecuentes

Rechazar SDD por «parecer waterfall». La semejanza es superficial; las diferencias de alcance, feedback y reversibilidad son las que importan.

Aplicar la secuencia al proyecto entero. Eso sí sería waterfall; SDD aplica la secuencia a cada incremento pequeño.

Leer el Manifiesto como «nada de documentación». Dice «sobre», no «en lugar de»; la documentación operativa contribuye directamente al software funcionando.

Para profundizar

· Thoughtworks Technology Radar — SDD

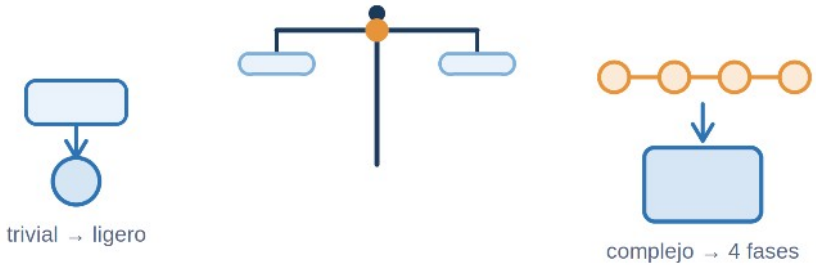
· Manifiesto Ágil

BLOQUE A · EL PARADIGMA: POR QUÉ SDD

Capítulo 4. El principio de proporcionalidad

ESTABLECIDO

Ajustar la intensidad del proceso al tamaño del problema; SDD mal calibrado se vuelve burocracia.



Se ajusta la intensidad del proceso al tamaño del problema: ni un mazo para una nuez, ni al revés.

Introducción

SDD no es un formulario que se rellena mecánicamente. El principio que gobierna su aplicación sensata es el de proporcionalidad: ajustar la intensidad del proceso al tamaño del problema. Un bug menor no necesita una spec de cuatro fases; una funcionalidad compleja que afecta a múltiples partes del sistema sí.

4.1 El riesgo de la sobreaplicación

La propia Böckeler documentó un caso revelador: al pedir a una herramienta que arreglara un bug pequeño, el documento de requisitos generado convirtió la tarea en cuatro historias de usuario con dieciséis criterios de aceptación. «Era como usar un mazo para romper una nuez». SDD mal calibrado se convierte en una burocracia de especificación que ralentiza en lugar de acelerar. La solución no es rechazar SDD, sino ajustar su intensidad.

4.2 La pregunta de calibración

La heurística práctica: ¿qué ocurre si el agente toma la decisión equivocada en este punto? Si la respuesta es «lo corrijo en dos minutos», no necesita especificarse. Si es «pierdo horas o introduzco un bug sutil», sí. SDD aporta valor neto cuando el cambio es sustancial, hay ambigüedad, hay riesgo, interviene más de una persona o agente, o el código debe mantenerse. Cuando ninguna de esas condiciones se cumple, el vibe coding sigue siendo la herramienta adecuada.

Saber cuándo NO usar SDD es parte de dominarlo. La sabiduría ágil es contextual: no hay práctica siempre buena o siempre mala. El principio de proporcionalidad reaparece en la calibración de la spec (capítulo 9) y en los antipatrones de equipo (capítulo 15).

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Enunciar el principio de proporcionalidad y aplicarlo para decidir la intensidad de SDD ante un cambio dado.
- Usar la pregunta de calibración (coste de una decisión equivocada) para decidir qué especificar.
- Identificar las condiciones en las que SDD aporta valor neto frente a las que no.

- Reconocer la sobreaplicación de SDD como un modo de fallo, no como rigor.

Errores y antipatrones frecuentes

Aplicar SDD con la misma intensidad a todo. El bug trivial y la funcionalidad compleja no merecen el mismo proceso; igualarlos genera burocracia.

Confundir exhaustividad con calidad. Más fases y más criterios no es mejor SDD; es, a menudo, el mazo para la nuez.

Olvidar que el vibe coding sigue teniendo su sitio. Para scripts, prototipos y cambios triviales, especificar es overhead innecesario.

Para profundizar

· Martin Fowler — SDD tools

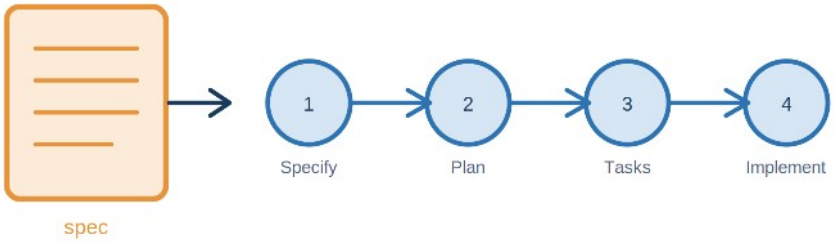
· Scrum Manager — Scrum en la era de la IA

BLOQUE B · EL MÉTODO: FASES Y PUERTAS

Capítulo 5. El flujo de cuatro fases

ESTABLECIDO

Requisitos → Diseño → Tareas → Implementación: la secuencia común a todas las implementaciones de SDD.



El flujo converge en cuatro fases: requisitos, diseño, tareas e implementación.

Introducción

Independientemente de la herramienta, todas las implementaciones de SDD convergen en la misma estructura: Requisitos → Diseño → Tareas → Implementación. Kiro las llama Requirements, Design, Tasks; GitHub Spec Kit las llama Specify, Plan, Tasks. La estructura subyacente es la misma porque refleja la secuencia lógica mínima para pasar de una intención a software funcionando cuando el constructor es un agente.

5.1 Las cuatro fases y sus entregables

4 **Requisitos.** Qué se construye, desde la perspectiva del usuario y del negocio. No es técnico: describe comportamiento esperado, condiciones y cómo se verificará.

5 **Diseño.** Cómo se construye. Aquí sí hay decisiones técnicas: patrones, componentes afectados, estructura de datos, coherentes con el codebase existente.

6 **Tareas.** El diseño se descompone en unidades atómicas de implementación, cada una con un prompt estructurado.

7 **Implementación.** Los agentes ejecutan las tareas y producen código, tests y commits. Paradójicamente, es la fase donde el humano interviene menos si las anteriores se hicieron bien.

5.2 El principio rector

El principio que gobierna el flujo es: revisa en las puertas de fase, no durante la implementación. En el trabajo conversacional, el desarrollador aprueba microdecisiones una a una, lo que genera fatiga de aprobación. SDD concentra la revisión humana donde la información es más valiosa y el coste de corrección menor: entre fases. Una vez aprobadas las tareas, la implementación se ejecuta con mínima intervención porque el contrato ya está claro.

No es todo o nada. Las cuatro fases representan el flujo completo, pero no todos los problemas las necesitan con la misma profundidad. Una funcionalidad mediana puede llevar requisitos y tareas con un diseño ligero. Calibrar la intensidad es el principio de proporcionalidad del capítulo 4 aplicado al flujo.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Enumerar las cuatro fases y el entregable de cada una.
- Explicar por qué la estructura es común a todas las herramientas de SDD.
- Aplicar el principio rector: revisar en las puertas, no durante la implementación, y razonar por qué reduce la fatiga de aprobación.
- Calibrar la profundidad de cada fase según el tamaño del problema.

Errores y antipatrones frecuentes

Mezclar el qué y el cómo. Decisiones técnicas en la fase de requisitos son uno de los caminos más rápidos a la sobreespecificación.

Revisar durante la implementación en vez de en las puertas. Reintroduce la fatiga de aprobación que SDD busca eliminar.

Aplicar las cuatro fases con igual profundidad siempre. El flujo se calibra; no es un formulario obligatorio.

Para profundizar

· GitHub Spec Kit

· AWS Kiro

BLOQUE B · EL MÉTODO: FASES Y PUERTAS

Capítulo 6. Las puertas de aprobación

ESTABLECIDO

Los puntos donde el humano ejerce criterio entre fases; el mecanismo que hace viable el flujo.

El criterio humano se concentra en las puertas entre fases, donde corregir cuesta menos.

Introducción

Si las cuatro fases son el cuerpo de SDD, las puertas de aprobación son su sistema nervioso. Son los puntos donde el humano ejerce criterio, se detectan problemas antes de que sean caros y el equipo se alinea. Hay tres puertas principales, una entre cada par de fases, más una revisión final al cierre de la implementación.

6.1 Qué se revisa en cada puerta

Puerta 1 (Requisitos → Diseño). ¿Resolvemos el problema correcto? Historias que reflejan necesidades reales, criterios verificables, alcance manejable, requisitos implícitos detectados (seguridad, rendimiento, accesibilidad).

Puerta 2 (Diseño → Tareas). ¿Es viable y coherente? El diseño respeta los patrones del codebase, las decisiones están justificadas, las dependencias y riesgos identificados.

Puerta 3 (Tareas → Implementación). ¿Estas tareas en este orden producen el diseño? Cobertura sin huecos, dependencias correctas, prompts precisos, criterios de éxito evaluables.

6.2 El coste de un error según la fase

Hay una razón económica detrás de las puertas: el coste de corregir un error crece de forma acelerada por fase. Un criterio de aceptación ambiguo es trivial de corregir en requisitos (se reescribe una frase); si llega al diseño, hay que repensar la solución; si llega a las tareas, rehacer la descomposición; si llega a la implementación, descartar y regenerar código. Cada minuto invertido en revisar un requisito ahorra horas de reimplementación.

6.3 Revisar una spec frente a revisar código

Las puertas 1-3 revisan documentos de texto, no código. Esto tiene tres implicaciones: accesibilidad (más personas del equipo, incluidos no técnicos, pueden participar), velocidad (leer una spec es más rápido que leer código) y foco (la pregunta es «¿esto es lo que queremos?», separada de «¿está bien implementado?», que se reserva a la revisión de implementación). Separar ambas preguntas hace cada revisión más eficaz.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Describir qué se revisa en cada una de las tres puertas y en la revisión final.
- Explicar la lógica económica: el coste de corregir crece de forma acelerada por fase.
- Justificar por qué revisar specs (texto) es más accesible, rápido y enfocado que revisar código.

- Asegurar que la aprobación es un acto deliberado y explícito, no un trámite.

Errores y antipatrones frecuentes

Saltarse puertas por presión de calendario. Es donde el error es más barato de corregir; saltarlas traslada el coste a fases caras.

Aprobar sin leer (teatro de aprobación). Una puerta que no se ejerce de verdad no detecta nada; es peor que no tenerla, porque da falsa garantía.

Mezclar «¿es lo que queremos?» con «¿está bien implementado?». Separar ambas preguntas es lo que hace eficaz cada revisión.

Para profundizar

· Martin Fowler — Exploring Gen AI

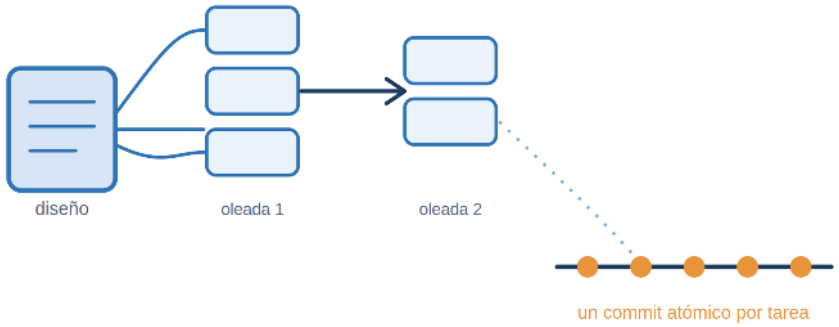
· GitHub Spec Kit

BLOQUE B · EL MÉTODO: FASES Y PUERTAS

Capítulo 7. Tareas atómicas, oleadas y commits atómicos

EN CONSOLIDACIÓN

Descomponer el diseño en unidades pequeñas, ejecutarlas en oleadas y registrar un commit por tarea.



Tareas pequeñas en oleadas paralelas, con un commit atómico por tarea.

Introducción

La fase de tareas transforma la complejidad de un cambio grande en una serie de cambios pequeños y manejables. La complejidad no desaparece, pero se convierte en una secuencia de piezas con fronteras claras. Este capítulo desarrolla la anatomía de una tarea atómica, su organización en oleadas y la práctica del commit atómico.

7.1 Anatomía de una tarea atómica

Una tarea bien definida afecta normalmente a entre uno y tres ficheros —si necesita más, probablemente se puede descomponer—. Incluye un prompt estructurado con cuatro campos: rol (qué papel asume el agente), tarea (qué hacer concretamente), restricciones (qué no hacer o qué límites respetar) y criterios de éxito (cómo se verificará). Y es independiente verificable: su resultado puede evaluarse de forma aislada, sin haber completado las demás.

7.2 Dependencias y oleadas

Las tareas no son necesariamente secuenciales. Las que no dependen entre sí se agrupan en una misma oleada y se ejecutan en paralelo, cada una por un subagente con contexto limpio; las oleadas se ejecutan en secuencia. Este modelo aprovecha la capacidad de los agentes para trabajar en paralelo sin contaminación de contexto entre tareas.

7.3 Commits atómicos

Cada tarea genera un commit independiente, lo que tiene consecuencias profundas: reversibilidad granular (revertir una tarea defectuosa sin afectar a las demás), trazabilidad (cada commit se vincula a una tarea, a un diseño y a unos requisitos: la trazabilidad que los marcos regulatorios exigen) y bisección eficaz (identificar con precisión qué tarea introdujo un bug).

El riesgo de la descomposición excesiva. Si el prompt que describe una tarea es más largo que el código que producirá, la descomposición ha ido demasiado lejos. El tamaño correcto: el agente trabaja con contexto completo y el resultado es revisable de un vistazo, implementable y verificable en minutos, no en horas.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Definir una tarea atómica por su alcance limitado, su prompt estructurado de cuatro campos y su independencia verificable.
- Organizar tareas en oleadas según dependencias, aprovechando el paralelismo de subagentes con contexto limpio.
- Justificar el commit atómico por su reversibilidad granular, trazabilidad y utilidad para la bisección.
- Reconocer el tamaño correcto de tarea y evitar tanto la sobrecarga como la descomposición excesiva.

Errores y antipatrones frecuentes

Tareas que tocan demasiados ficheros. Reducen la precisión de la implementación; conviene descomponer más.

Descomposición excesiva. Si el prompt es más largo que el código que produce, el overhead supera el beneficio.

Commits que mezclan varias tareas. Pierden reversibilidad granular y trazabilidad, y hacen inútil la bisección.

Para profundizar

· GitHub Spec Kit

· Anthropic — Building effective agents

BLOQUE B · EL MÉTODO: FASES Y PUERTAS

Capítulo 8. El Impact Report en codebase existente (brownfield)

EN CONSOLIDACIÓN

Analizar el código actual antes de escribir requisitos, para no duplicar ni contradecir lo existente.

En brownfield se analiza el código actual antes de especificar: qué se ve afectado y qué respetar.

Introducción

La mayor parte de la literatura sobre SDD asume proyectos greenfield (desde cero). Pero la realidad de la mayoría de equipos es el trabajo sobre codebases existentes, a menudo grandes y heredados. En esos entornos —brownfield— la primera fase no empieza escribiendo requisitos, sino analizando el código actual. Es donde SDD aporta más valor, porque un agente que ignora el contexto existente causa más daño.

8.1 Qué es el Impact Report

El Impact Report (informe de impacto) responde a tres preguntas antes de escribir un solo requisito: ¿qué ficheros se verán afectados por este cambio?, ¿qué patrones y convenciones existen que debemos respetar?, ¿qué efectos colaterales podría tener en otras partes del sistema? Se genera mediante búsqueda en el codebase —estructural y semántica— y el equipo lo revisa.

8.2 Su función preventiva

El Impact Report evita que los requisitos pidan algo que duplica lo existente, contradice patrones establecidos o ignora dependencias. Si el sistema ya tiene un mecanismo de envío de correos, el requisito no debería pedir «construir un sistema de correos» sino «extender el sistema existente para soportar el nuevo canal». Es la diferencia entre diseñar sobre un terreno que conoces y diseñar sobre uno que imaginas.

Greenfield y brownfield. En proyectos nuevos, el Impact Report no aplica en la primera iteración, pero sí a partir de la segunda funcionalidad: en cuanto hay código, hay contexto que respetar. El Impact Report alimenta también la fase de diseño, que lista qué ficheros se crearán y cuáles se modificarán.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar qué es el Impact Report y las tres preguntas que responde.
- Justificar su función preventiva: evitar duplicación, contradicción de patrones e ignorancia de dependencias.
- Distinguir el tratamiento de proyectos greenfield y brownfield respecto al Impact Report.

- Relacionar el Impact Report con la fase de diseño y la identificación de ficheros afectados.

Errores y antipatrones frecuentes

Escribir requisitos sin analizar el código existente. Lleva a pedir cosas que duplican o contradicen lo que ya hay.

Asumir greenfield cuando el proyecto es brownfield. La mayoría del trabajo real es sobre código heredado; ignorarlo es la fuente principal de efectos colaterales.

Tratar el Impact Report como un trámite. Si no se revisa de verdad, no cumple su función preventiva.

Para profundizar

· Martin Fowler — Exploring Gen AI

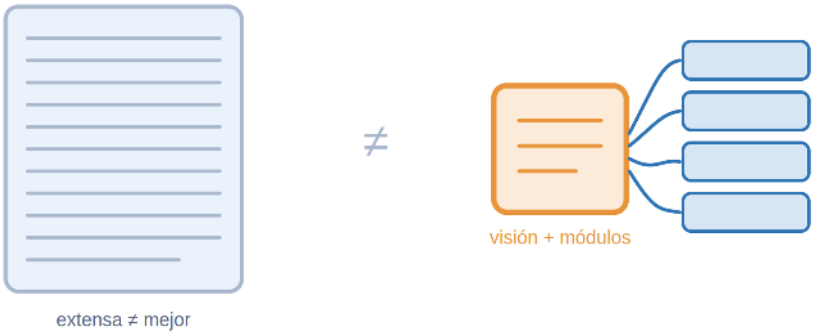
· GitHub Spec Kit

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 9. La spec inteligente, no extensa

ESTABLECIDO

Una buena spec cubre lo justo para guiar al agente sin abrumarlo: minimal no significa corto.



Minimal no significa corto: una buena spec guía al agente sin abrumarlo.

Introducción

Si una frase resume lo que separa una buena spec de una mala es esta: minimal no significa necesariamente corto. Una buena spec no escatima detalle cuando importa, pero no añade información que no contribuye al resultado. Addy Osmani destiló cinco principios, respaldados por el análisis de GitHub de más de 2.500 ficheros de configuración de agentes, que sirven de marco práctico.

9.1 Los cinco principios de Osmani

8 **Empieza con la visión de alto nivel y deja que la IA elabore los detalles.** El humano aporta el qué y el porqué; el agente propone el cómo con detalle; el humano valida. La spec se co-crea, no se entrega.

9 **Estructura la spec como un documento profesional.** Los agentes procesan mejor la información estructurada. El estudio de GitHub identificó seis áreas en las specs efectivas: comandos, testing, estructura del proyecto, estilo de código, flujo git y boundaries.

10 **Divide las tareas en prompts modulares.** Alimenta al agente con la porción de la spec relevante para la tarea, no con la spec completa. Conecta con la maldición de las instrucciones (capítulo 12).

11 **Incorpora autoverificación, restricciones y conocimiento experto.** Instruir al agente para que compare su resultado con la spec, anticipar dónde puede equivocarse y volcar el conocimiento de dominio que solo aporta el profesional.

12 **Trata la spec como un documento vivo.** Actualízala cuando se toman decisiones o se descubre información nueva. Una fuente de verdad desactualizada es peor que no tenerla (capítulo 13).

9.2 El principio de proporcionalidad aplicado a la spec

El principio del capítulo 4 vuelve aquí como heurística: ajusta el detalle de la spec a la complejidad de la tarea. La pregunta de calibración: ¿qué ocurre si el agente toma la decisión equivocada en este punto? Si se corrige en dos minutos, no necesita estar en la spec; si cuesta horas o introduce un bug sutil, sí.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Enunciar y aplicar los cinco principios de Osmani para escribir una spec.
- Cubrir las seis áreas de una spec efectiva identificadas por el estudio de GitHub.
- Calibrar el detalle de la spec con la pregunta sobre el coste de una decisión equivocada.
- Distinguir entre una spec inteligente y una spec simplemente larga.

Errores y antipatrones frecuentes

Añadir detalle para «cubrir más casos». Tiene rendimientos decrecientes y luego negativos; la spec inteligente cura, no acumula.

Entregar la spec completa para cada tarea. Sobrecarga el contexto; conviene alimentar solo la porción relevante.

Escribir la spec una vez y olvidarla. Una spec desactualizada induce a error a quien la consulta.

Para profundizar

· Addy Osmani — How to write a good spec for AI agents

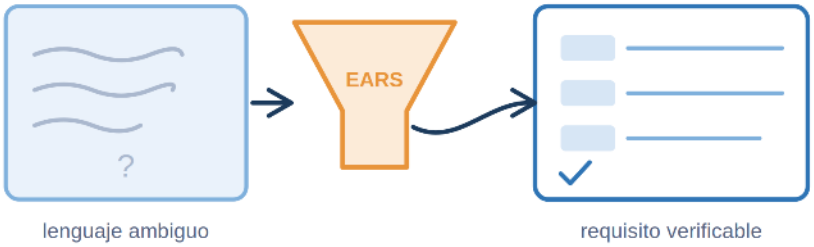
· AGENTS.md

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 10. La notación EARS para criterios de aceptación

EN CONSOLIDACIÓN

Patrones que restringen el lenguaje natural lo justo para eliminar ambigüedad sin sacrificar legibilidad.



EARS estructura el lenguaje natural lo justo para eliminar la ambigüedad sin perder legibilidad.

Introducción

Varias herramientas SDD usan la notación EARS (Easy Approach to Requirements Syntax) para estructurar los criterios de aceptación. Desarrollada por Alistair Mavin en Rolls-Royce y presentada en 2009, ha sido adoptada por organizaciones como Airbus, NASA y Siemens. No es compleja: es un conjunto de patrones basados en palabras clave que reducen la ambigüedad del lenguaje natural sin sacrificar legibilidad.

10.1 La plantilla y los cinco patrones

La plantilla básica: «Mientras [precondición], cuando [evento], el sistema debe [respuesta]». EARS define cinco patrones que cubren la mayoría de requisitos:

- **Ubicuo:** propiedad permanente del sistema. «El sistema debe cifrar todas las contraseñas con bcrypt y coste mínimo 12».
- **Dirigido por evento (cuando…):** se activa al ocurrir algo. «Cuando se publica una versión de un documento, el sistema debe encolar una notificación en 5 segundos».
- **Dirigido por estado (mientras…):** activo mientras se cumple una condición. «Mientras el usuario tiene activo “no molestar”, el sistema debe retener las notificaciones push».
- **Comportamiento no deseado (si…):** gestiona errores. «Si la contraseña se introduce mal tres veces, el sistema debe bloquear la cuenta 15 minutos».
- **Opcional (donde…):** funcionalidad condicionada a la configuración. «Donde la organización ha habilitado 2FA, el sistema debe pedir un código tras la contraseña».

10.2 Por qué funciona con agentes

Los agentes responden bien a requisitos estructurados con patrones consistentes. Un requisito en EARS reduce la probabilidad de malinterpretación porque la estructura elimina las ambigüedades más comunes del lenguaje natural. La notación no es obligatoria, pero es especialmente útil para que los criterios de aceptación sean verificables: debe ser posible determinar, al ver el resultado, si se cumplen o no.

Verificable, no vago. «La notificación es rápida» no es verificable. «La notificación llega al canal en menos de 30 segundos en el 99 % de los casos» sí lo es. EARS empuja hacia la segunda forma.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Reconocer y aplicar la plantilla EARS y sus cinco patrones a criterios de aceptación reales.
- Elegir el patrón EARS adecuado según la naturaleza del requisito.
- Explicar por qué los requisitos estructurados reducen la malinterpretación del agente.
- Redactar criterios verificables en lugar de vagos.

Errores y antipatrones frecuentes

Criterios de aceptación vagos. «Rápido», «intuitivo», «seguro» sin umbral no son verificables ni para un humano ni para un agente.

Forzar EARS donde no aporta. Es una herramienta, no una obligación; se usa donde reduce ambigüedad de verdad.

Confundir el patrón. Usar «cuando» para una propiedad permanente o «mientras» para un evento puntual desvirtúa el criterio.

Para profundizar

· Alistair Mavin — EARS

· GitHub Spec Kit

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 11. El sistema de boundaries: Always / Ask First / Never

EN CONSOLIDACIÓN

Un marco de delegación de tres niveles, no una lista plana de prohibiciones.

La delegación se gradúa en tres niveles: Always, Ask First y Never.

Introducción

El análisis de GitHub sobre 2.500 configuraciones de agentes reveló que las specs más efectivas no usan una lista plana de reglas, sino un sistema de tres niveles que da al agente un marco de decisión claro sobre cuándo actuar, cuándo consultar y cuándo detenerse. Funciona como la delegación a un profesional competente, con la diferencia de que con un agente las reglas deben ser explícitas porque carece de sentido común para inferirlas.

11.1 Los tres niveles

Always (siempre hacer). Acciones que el agente ejecuta sin preguntar: ejecutar tests antes de un commit, seguir las convenciones de nomenclatura, registrar errores, incluir manejo de errores en los endpoints.

Ask First (preguntar primero). Acciones que podrían ser correctas pero requieren aprobación por su impacto: modificar el esquema de base de datos, añadir dependencias nuevas, cambiar la configuración de CI/CD, modificar la API pública.

Never (nunca hacer). Líneas rojas absolutas: hacer commit de secretos, editar directorios de dependencias, eliminar un test que falla sin aprobación, modificar configuración de producción.

11.2 Por qué funciona y cómo evoluciona

El sistema da autonomía donde es segura (Always libera de consultar microdecisiones), crea puntos de control donde son necesarios (Ask First concentra la intervención en lo de alto impacto) y establece líneas rojas inequívocas (Never elimina categorías enteras de error). Además, no es estático: a medida que el equipo gana confianza, algo que hoy es Ask First puede pasar a Always. Es autonomía gradual, análoga a cómo un mánager delega cada vez más en un profesional que demuestra criterio.

Los boundaries operan en dos niveles: a nivel de proyecto (en la constitución, tipo CLAUDE.md o AGENTS.md) definen reglas generales; a nivel de tarea, restricciones específicas de esa implementación. La combinación crea un marco de delegación completo: consistencia global más precisión local.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Clasificar acciones en Always / Ask First / Never con criterio.
- Explicar por qué el sistema de tres niveles supera a una lista plana de reglas.
- Gestionar la evolución de los boundaries como autonomía gradual a medida que crece la confianza.
- Distinguir boundaries de proyecto (constitución) de boundaries de tarea.

Errores y antipatrones frecuentes

Una lista plana de prohibiciones. Sin niveles, el agente no sabe qué puede hacer solo, qué consultar y qué evitar.

Boundaries que nunca evolucionan. Mantener todo en Ask First indefinidamente desaprovecha la confianza ganada y satura de consultas.

Confundir el nivel de proyecto con el de tarea. «Nunca commitear secretos» es de proyecto; «no tocar los endpoints existentes» es de la tarea concreta.

Para profundizar

· Addy Osmani — How to write a good spec for AI agents

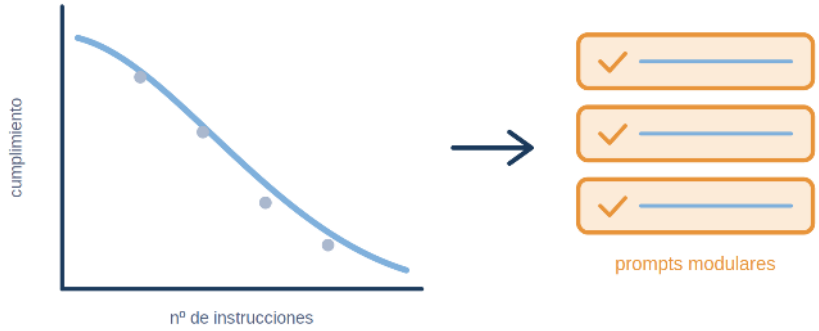
· AGENTS.md

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 12. La maldición de las instrucciones

EN CONSOLIDACIÓN

A más instrucciones en un prompt, menor probabilidad de que el modelo cumpla cada una.



A más instrucciones en un prompt, menor cumplimiento: la respuesta es dividir en módulos.

Introducción

Uno de los hallazgos más relevantes para SDD proviene de la investigación. El trabajo «Curse of Instructions» (2024) mostró un resultado regular: a medida que se acumulan instrucciones en un prompt, el rendimiento del modelo cumpliendo cada instrucción individual cae de forma predecible. El efecto se ha observado en los principales modelos: cumplen las primeras instrucciones con fiabilidad pero empiezan a incumplir las últimas a medida que la lista crece.

12.1 Implicaciones para el diseño de specs

Si presentas diez reglas detalladas, el agente podría cumplir las primeras y pasar por alto las últimas; con cincuenta, leerá la spec parcialmente. Las consecuencias prácticas:

- Una spec más inteligente, no más larga: añadir una instrucción más puede empeorar el cumplimiento de las anteriores.
- Descomponer en vez de acumular: cinco specs de diez instrucciones, asignadas a las tareas donde son relevantes, frente a una de cincuenta. Es lo que hace la fase de tareas.
- Priorizar: si no se puede dividir más, las instrucciones más importantes —seguridad, por ejemplo— van primero, donde el modelo les da más peso.
- Separar niveles: el sistema de boundaries crea categorías de urgencia distinta en lugar de una lista plana de igual peso.

12.2 La economía de la atención del agente

La analogía con la gestión de equipos: un mánager que da veinte directrices en una reunión no debería sorprenderse si el empleado olvida la mitad. Lo efectivo es pocas reglas claras para el día a día, directrices específicas por tarea y una referencia consultable para lo demás. SDD aplica ese modelo: la constitución contiene las pocas reglas que aplican siempre; el prompt de cada tarea, las específicas; y la spec completa existe como referencia, sin cargarse entera en cada interacción.

Señales de spec sobrecargada. El agente ignora restricciones escritas, cumple lo del principio pero no lo del final, mejora cuando se reduce la spec, o acierta con tareas aisladas y falla con varias juntas. Si aparecen, el problema no es «el agente es malo» sino «la spec le pide demasiado a la vez».

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar la maldición de las instrucciones y su efecto sobre el cumplimiento.
- Responder a la sobrecarga descomponiendo, priorizando y separando niveles, en lugar de añadiendo detalle.
- Reconocer las señales de una spec sobrecargada.
- Relacionar el hallazgo con la fase de tareas y el sistema de boundaries.

Errores y antipatrones frecuentes

Responder a un fallo añadiendo más instrucciones. Empeora el problema; la respuesta es dividir mejor, no escribir más.

Listas largas de reglas de igual peso. El modelo no las pondera; las críticas deben ir primero y separadas por nivel.

Cargar la spec completa en cada interacción. Satura el contexto; se alimenta solo la porción relevante.

Para profundizar

· Curse of Instructions (paper)

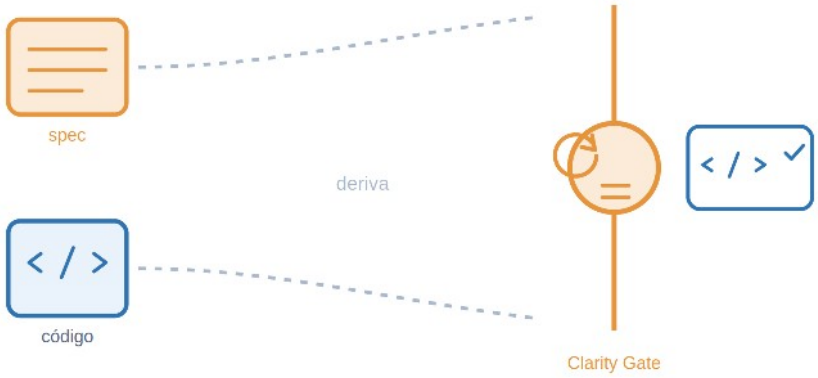
· Anthropic — Effective context engineering

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 13. Specs vivas, deriva y la Clarity Gate

EN CONSOLIDACIÓN

Mantener la spec alineada con el código que evoluciona; la deriva es el riesgo principal a medio plazo.



La deriva spec-código es el riesgo principal; la Clarity Gate prueba si la spec basta para regenerar el código.

Introducción

Escribir una buena spec es la mitad del desafío; la otra mitad es mantenerla alineada con el código a medida que el proyecto avanza. La deriva ocurre cuando la spec dice una cosa y el código ha evolucionado hacia otra. Pasa rápido: basta una decisión menor del agente no cubierta por la spec, o un ajuste durante la implementación sin actualizar el documento. Una spec derivada no solo es inútil: es un riesgo activo, porque quien la consulta toma decisiones sobre información incorrecta.

13.1 Los tres niveles de vida de la spec

La taxonomía de Böckeler describe cómo distintos equipos gestionan el ciclo de vida:

- **Spec-first:** se escribe antes de codificar, guía la tarea actual y se descarta. La deriva no es problema porque no hay spec que mantener, pero se pierde la documentación como activo.
- **Spec-anchored:** se mantiene como documentación viva del sistema. La deriva es un riesgo real que exige disciplina: cuando el código cambia, la spec se actualiza.
- **Spec-as-source:** la spec es el artefacto principal; solo se edita la spec y el código se regenera. La deriva desaparece por diseño, pero la regeneración plantea retos de rendimiento y determinismo. Es todavía experimental.

13.2 La Clarity Gate

La Clarity Gate (puerta de claridad) es una prueba de calidad: ¿puede un agente diferente, o una sesión nueva del mismo agente, generar código funcionalmente equivalente solo con la spec? Si sí, la spec es clara y completa. Si no, tiene supuestos implícitos —conocimiento que vivía en el contexto de la conversación original pero no quedó capturado—, que son la causa principal de la deriva. Es una prueba que cualquier equipo puede aplicar: dar la spec a otro agente sin contexto y comparar el resultado.

13.3 Estrategias para mantener specs vivas

Para equipos en nivel spec-anchored: actualizar la spec en el mismo commit que el cambio («luego no existe»), versionar la spec con un historial de qué cambió y por qué, revisar el estado de las specs

en la retrospectiva, y para specs críticas, crear tests de conformidad que verifiquen que el código cumple la spec (Simon Willison los llama conformance suites).

La spec como diagnóstico. Cuando el código falla, el problema suele originarse en la spec. Si la spec es correcta y el código no la cumple, es problema de implementación: se regenera la tarea. Si la spec es incorrecta, se corrige la spec y se regenera. Es un diagnóstico más limpio que el debugging tradicional.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar la deriva y por qué una spec desactualizada es un riesgo activo.
- Distinguir los tres niveles de vida de la spec y elegir el adecuado al contexto.
- Aplicar la Clarity Gate para detectar supuestos implícitos en una spec.
- Emplear estrategias de mantenimiento: actualización inmediata, versionado, revisión en retrospectiva, tests de conformidad.

Errores y antipatrones frecuentes

«Lo actualizo luego». Luego no existe; la spec se actualiza en el mismo commit que el cambio o deriva.

Mantener viva toda spec por defecto. El mantenimiento tiene coste; una spec que nadie consulta ni actualiza es documentación zombi.

Asumir que la spec está clara sin probarlo. La Clarity Gate revela supuestos implícitos que no se ven desde dentro de la conversación original.

Para profundizar

· Martin Fowler — SDD tools

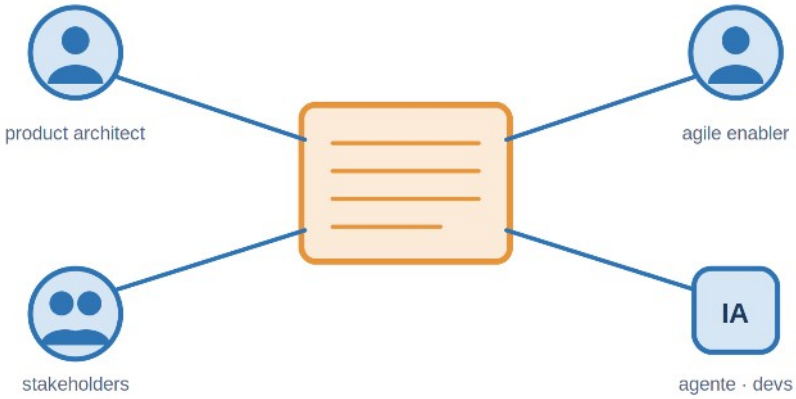
· Simon Willison — weblog

BLOQUE D · SDD EN EL EQUIPO Y SU ECOSISTEMA

Capítulo 14. Encaje en roles y ceremonias ágiles

EN CONSOLIDACIÓN

Cómo SDD redistribuye el trabajo del equipo, incluidos los roles no técnicos, y transforma las ceremonias.



La spec, legible por no técnicos, redistribuye el trabajo y abre la colaboración del equipo.

Introducción

SDD no es una capa que se añade sin modificar el trabajo existente: cambia la naturaleza del trabajo de cada rol y la forma de las ceremonias. La Guía de Scrum en equipos con IA define la estructura del equipo (roles, artefactos); SDD define cómo ese equipo construye software en el día a día. Son dos caras de la misma adaptación.

14.1 Los roles

Product owner / product architect. La exigencia de precisión sube: los criterios de aceptación pasan de guía a contrato. A cambio, las specs son artefactos que cualquier stakeholder puede leer, lo que acorta el ciclo de feedback con el negocio.

Desarrollador / product builder. La competencia se desplaza del código a analizar, especificar, orquestar y revisar. Gana la capacidad de producir a una velocidad antes imposible, pero solo si las fases previas se hicieron bien: la inversión se desplaza de la ejecución a la planificación.

Agile enabler. Facilita las puertas de aprobación como puntos de inspección y vigila los riesgos nuevos (sobreespecificación, puertas como cuellos de botella, teatro de especificación, rigidez excesiva), asegurando que el equipo mantiene el ritmo iterativo.

Stakeholders. Ganan visibilidad: pueden leer una spec y opinar sobre prioridades o conformidad antes de que se convierta en código. La spec reduce la asimetría de información entre quien construye y quien paga o usa.

14.2 Las ceremonias con SDD

El Sprint pasa a contener dos tipos de ítems: specs por crear (funcionalidades que deben pasar el flujo) y specs aprobadas listas para implementar. El Sprint Planning se centra en qué specs nuevas se desarrollan y qué specs aprobadas se implementan. La estimación se desdobla en esfuerzo de especificación (humano, el que consume capacidad real) y de implementación (mayoritariamente del agente, predecible). La Definition of Done se enriquece: el código cumple los criterios de éxito de las tareas, respeta el diseño, los commits son atómicos y la spec se ha actualizado si la implementación difirió. La Daily se centra en el estado de las specs y las desviaciones, no en el

repaso mecánico.

El doble carril y la Definition of Ready for AI. El carril de Discovery alimenta la fase de requisitos; el de Delivery es el territorio de diseño, tareas e implementación. La spec aprobada es el puente entre descubrir y construir, y ES el Definition of Ready para esa funcionalidad: el mismo concepto en el lenguaje de SDD.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Describir cómo cambia el trabajo de cada rol del equipo con SDD, incluidos los no técnicos.
- Explicar los dos tipos de ítem del backlog y el desdoblamiento de la estimación.
- Enunciar cómo se enriquecen la Definition of Done y la Daily con SDD.
- Relacionar el doble carril y la Definition of Ready for AI con el flujo de SDD.

Errores y antipatrones frecuentes

Tratar SDD como asunto solo de desarrolladores. La especificación es trabajo compartido; los roles no técnicos ganan voz y visibilidad.

Estimar solo la implementación. El esfuerzo que consume capacidad del equipo es el de especificación, no el del agente.

Revisión de código sin contrato. En SDD se revisa contra la spec («¿cumple lo que dice?»), no solo contra el gusto del revisor.

Para profundizar

· Scrum Manager — Scrum en la era de la IA

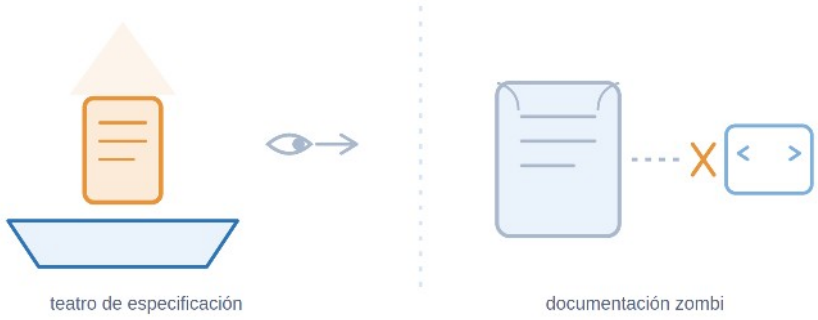
· Scrum.org — El Scrum Master y la IA

BLOQUE D · SDD EN EL EQUIPO Y SU ECOSISTEMA

Capítulo 15. Antipatrones: teatro de especificación y documentación zombi

EN CONSOLIDACIÓN

Los modos de fallo característicos de SDD mal aplicado, que el agile enabler vigila.



Teatro de especificación y documentación zombi: los modos de fallo de SDD mal aplicado.

Introducción

Reconocer los antipatrones es tan parte del dominio como conocer el método. Son los puntos donde SDD deja de aportar valor y empieza a estorbar, y aparecen con facilidad si no se calibra con el principio de proporcionalidad. El agile enabler es quien los vigila.

15.1 Los antipatrones principales

Sobreespecificación. El equipo dedica más tiempo a perfeccionar la spec que a producir software funcionando. Es el «mazo para la nuez» del capítulo 4 llevado al proceso de equipo.

Puertas como cuellos de botella. Una aprobación se bloquea porque quien debe aprobar no está disponible. La respuesta es facilitar mecanismos de delegación o aprobación asíncrona, no eliminar la puerta.

Teatro de especificación. Se escriben specs que nadie revisa de verdad; el proceso se ejecuta mecánicamente sin aportar el valor de la revisión. Una puerta que no se ejerce es peor que su ausencia, porque da falsa garantía.

Documentación zombi. Specs que nadie consulta ni actualiza. Como cualquier artefacto, una spec que no sirve no es documentación viva: es lastre. Parte de la gestión del backlog de documentación es decidir qué specs mantener y cuáles dejar morir.

Rigidez excesiva. Aplicar SDD con la misma intensidad a un bug trivial que a una funcionalidad compleja. Es la negación del principio de proporcionalidad.

15.2 El papel del agile enabler

Estos riesgos no se corrigen solos. El agile enabler asegura que se dedica tiempo real a la revisión sin que las puertas se vuelvan trámite, modera los desacuerdos sobre las specs, detecta cuándo el proceso genera overhead innecesario y ajusta la intensidad, y vigila que las specs no se conviertan en mini-waterfalls. Es la función que mantiene SDD ágil en la práctica.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Identificar los antipatrones principales de SDD mal aplicado.

- Distinguir el teatro de especificación y la documentación zombi de la práctica que aporta valor.
- Aplicar el principio de proporcionalidad para prevenir la sobreespecificación y la rigidez.
- Describir el papel del agile enabler en la vigilancia y corrección de estos riesgos.

Errores y antipatrones frecuentes

Confundir rigor con cantidad de documentación. Más specs no es mejor SDD; la sobreespecificación es un fallo, no una virtud.

Mantener puertas que nadie ejerce de verdad. El teatro de especificación da falsa garantía; conviene aprobar con criterio o replantear la puerta.

Acumular specs zombi. Una spec que nadie consulta ni actualiza es lastre; decidir qué dejar morir es parte de la gestión.

Para profundizar

· Marmelab — SDD: The Waterfall Strikes Back

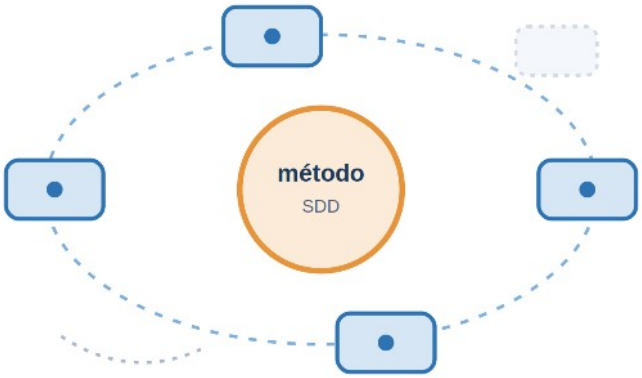
· Martin Fowler — SDD tools

BLOQUE D · SDD EN EL EQUIPO Y SU ECOSISTEMA

Capítulo 16. El ecosistema de herramientas

EMERGENTE

El instrumental de SDD crece deprisa; conviene no confundir el método con el instrumento.



El método es el núcleo estable; las herramientas orbitan y caducan: no confundir método e instrumento.

Introducción

El ecosistema de herramientas alrededor de SDD crece deprisa, y es la zona más volátil del tema: las versiones y la lista de herramientas cambian de mes en mes. Conviene tener presente la advertencia que recorre toda la metodología: SDD es el método, no la herramienta. Lo que sigue es una fotografía a la fecha del documento, sujeta a cambio rápido.

16.1 El panorama a junio de 2026

GitHub Spec Kit. La opción open source más adoptada por la comunidad. Es una CLI (llamada Specify) que soporta una treintena de agentes de codificación y publica versiones casi semanales; ha superado las 90.000 estrellas en GitHub. Es el punto de entrada habitual para equipos que quieren un flujo de SDD estructurado en su propio repositorio.

AWS Kiro. Un IDE dedicado a SDD, construido sobre Code OSS, que impone un flujo de fases. Apto para equipos que quieren un entorno integrado en lugar de añadir una CLI a su flujo existente.

Tessl. Explora el nivel spec-as-source, donde la spec es el artefacto principal y el código se regenera. Representa la frontera más experimental del campo.

Otras entradas. Han aparecido alternativas como BMAD o GSD, señal de un mercado en formación. La proliferación de herramientas es, en sí misma, indicio de que SDD se está consolidando como práctica.

16.2 El método por encima del instrumento

La conclusión práctica: conocer el panorama orienta la elección, pero la herramienta concreta caducará antes que el método. Los fundamentos de los bloques A, B y C —el paradigma, las fases y puertas, las buenas specs— son estables; las herramientas que los implementan son la capa que más se mueve. Invertir el aprendizaje en el método protege frente a la obsolescencia del instrumental.

Por qué este capítulo es «emergente». A diferencia del resto del documento, este capítulo caducará antes: las versiones, las cuotas de adopción y la propia lista de herramientas cambiarán en cada revisión del mapa. Tómalo como fotografía fechada, no como referencia permanente.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Situar las principales herramientas del ecosistema (Spec Kit, Kiro, Tessl y nuevas entradas) y qué resuelve cada una.
- Distinguir el método de SDD del instrumento que lo implementa.
- Razonar por qué invertir el aprendizaje en el método protege frente a la obsolescencia.
- Interpretar la proliferación de herramientas como señal de consolidación de la práctica.

Errores y antipatrones frecuentes

Confundir el método con la herramienta. Dominar Spec Kit no es dominar SDD; la herramienta cambia, el método permanece.

Elegir herramienta sin entender el método. Lleva a aplicar el flujo de forma mecánica sin criterio sobre cuándo y cómo.

Tomar este panorama como permanente. Es la capa más volátil; conviene contrastar con la edición vigente del mapa.

Para profundizar

· GitHub Spec Kit — releases

· AWS Kiro