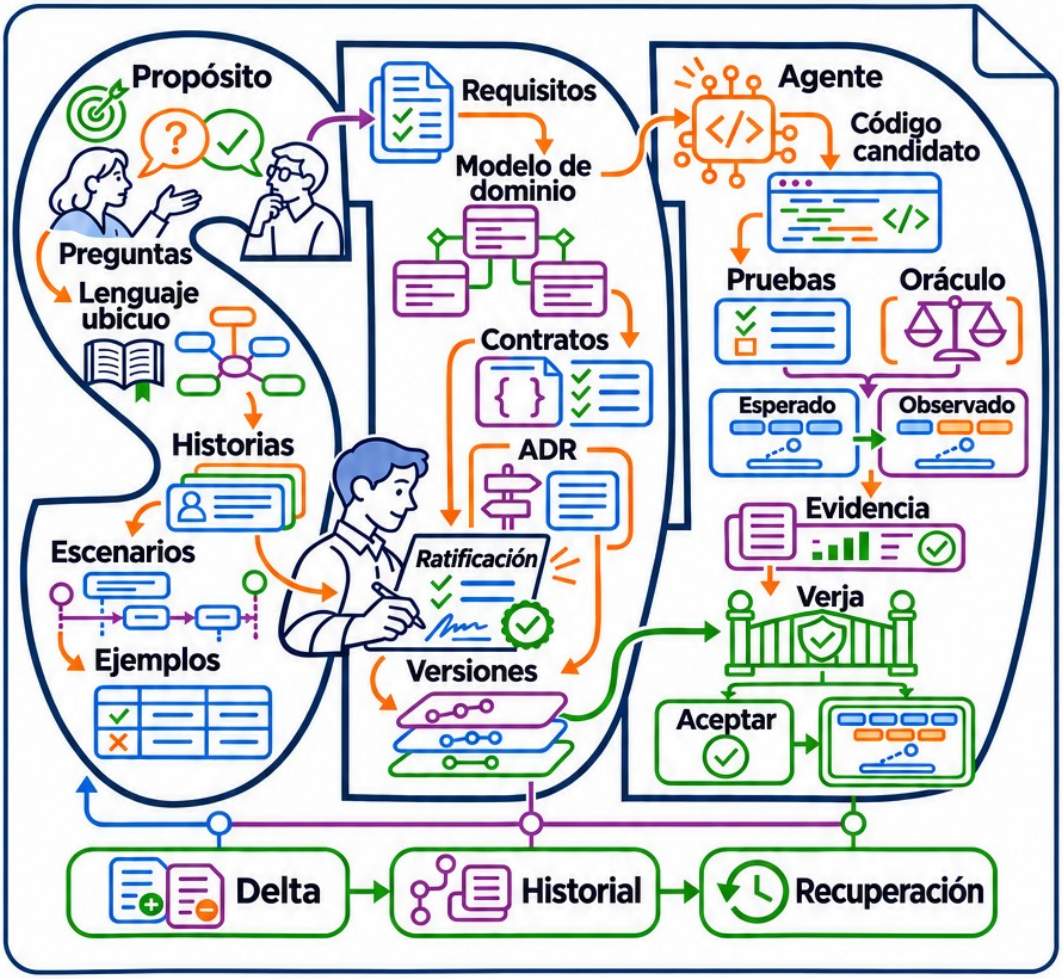
Desarrollo agéntico de software basado en especificaciones

Cómo guiar a los agentes de IA para construir soluciones verificables



Roberto Canales Mora

Desarrollo agéntico de software basado en especificaciones

Cómo guiar a los agentes de IA para construir soluciones verificables

Especificaciones versionadas, agentes generativos y verjas ejecutables.

Un caso de estudio práctico sobre un clon de Arkanoid.

Roberto Canales Mora

Versión 0.1 · edición preliminar · noviembre de 2026

© 2026 The Audience Megaphone System SL

B-87572772

Todos los derechos reservados.

Ninguna parte de esta publicación puede ser reproducida, distribuida o transmitida en ninguna forma ni por ningún medio, incluidos fotocopiado, grabación u otros métodos electrónicos o mecánicos, sin el permiso previo por escrito del autor, excepto en el caso de breves citas incorporadas en reseñas críticas y ciertos otros usos no comerciales permitidos por la legislación de propiedad intelectual.

Los nombres de productos, herramientas y marcas mencionados en este libro (entre otros, GitHub, GitHub Spec Kit, AWS Kiro, OpenSpec, Tessl, Structura, Obsidian, Claude, Claude Code, Codex, Git y Arkanoid) son marcas registradas de sus respectivos propietarios. Su mención tiene fines exclusivamente informativos y no implica afiliación ni respaldo por parte de dichos titulares.

Este libro reúne mi experiencia, el análisis de documentación pública y una propuesta de diseño. Las herramientas se describen tal como funcionaban en noviembre de 2026. Sus órdenes, ficheros y opciones cambian a menudo, así que lo descrito hay que contrastarlo con la versión que tengas instalada; no lo repetiré en cada capítulo. Los estudios conservan el alcance de sus casos y muestras; las cifras ilustrativas sirven para explicar un razonamiento.

Cuando un recorrido procede de documentación se identifica como tal, y cualquier aplicación que se construya a partir de este libro deberá comprobarse durante su construcción. Una prueba o una revisión aporta información sobre lo que ha examinado, no una garantía de todo el sistema. Esta valoración es del autor y no una evaluación oficial de los fabricantes; cada equipo tendrá que contrastarla con las necesidades de su proyecto.

Esta obra se ofrece “tal cual”, sin garantías de ningún tipo, expresas o implícitas. El autor no se hace responsable de los daños que pudieran derivarse del uso de la información contenida en ella.

Este libro ha sido corregido con la ayuda de las inteligencias artificiales Codex y Claude. La revisión de las sugerencias y la edición final corresponden al autor.

Primera edición: septiembre de 2026

Ficha del libro en Leanpub

Prefacio

Después de más de treinta años vinculado a la construcción de software, he pasado por muchas fases: programador, arquitecto, analista, preventa, responsable de desarrollo de un banco, formador, comercial, etc. Y, aunque durante años he procurado hacer pequeñas aplicaciones cuando salían nuevas tecnologías, últimamente me veía incapaz de competir con los técnicos por la amplitud y profundidad que conllevaba construir una aplicación empresarial.

Pero esto cambió hace apenas unos meses, con la evolución de los sistemas de **construcción agéntica** de software como Claude o Codex. Otra vez me parecía tener superpoderes y, sabiendo cómo se construye software bien (mi empresa lo ha hecho durante más de 20 años), solo tenía que decir a una herramienta lo que quería y esta lo hacía en un rato, mejor de lo que podría soñar.

Estaba estudiando un Máster oficial de IA en la UTAMED, y, como trabajo final, en lugar de quedarme en un estudio teórico decidí construir un sistema completo funcional: en vez de dedicar los 6 créditos que lleva la asignatura, estoy seguro de que dediqué en torno a 10 veces más. Esto es: mañanas, tardes y noches, gastando 200 euros al mes en una suscripción a Claude y 100 a Codex y con máquinas potentes (Mac Studio M3 Ultra con 256 GBytes de RAM) y MacBook Pro (M4 Max 64 GBytes). Podéis ver el resultado en la ficha de Solap

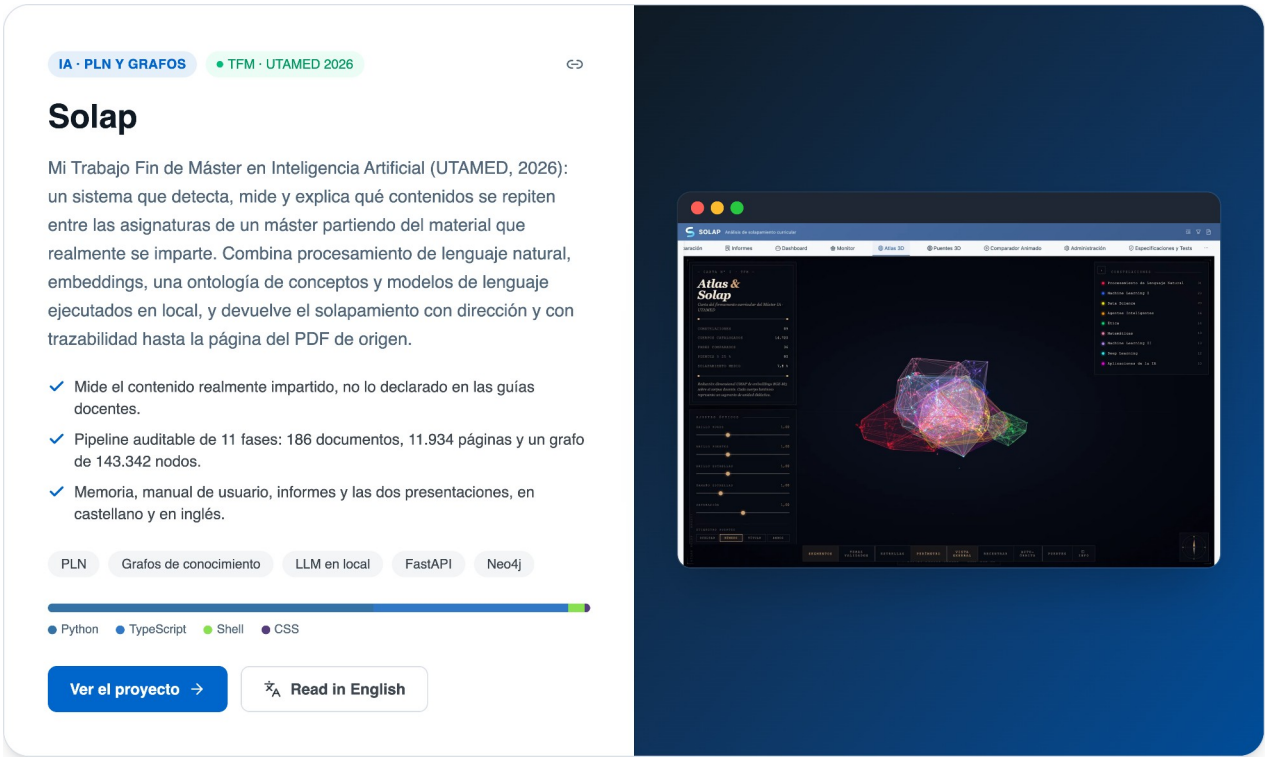


Figura 1. Solap, mi trabajo de fin de máster en Inteligencia Artificial en la UTAMED (2026).

Avanzaba a una velocidad que nunca había visto, y resultaba incluso adictivo esperar a ver qué era capaz de generar la IA a partir de unas instrucciones bastante genéricas.

Sin embargo, tras varias semanas trabajando en serio, el sistema empezaba a resultar **impredecible**, difícil de controlar y mantener.

Las pausas en el desarrollo en fines de semana o vacaciones creaban problemas. También al alternar el desarrollo entre un ordenador de sobremesa y un portátil. Gran parte del **contexto** permanecía en la conversación. Al abrir una nueva sesión, el **agente** recordaba solo parte de lo que se había decidido. Cuando la ventana de contexto se llenaba o se mezclaban distintas **intenciones**, aumentaban las **omisiones** y las **contradicciones**.

Pero **no tenía la seguridad de que las reglas y los algoritmos fueran estables**. ¿Qué cambiaba de una ejecución a otra? ¿Las soluciones eran las mejores o solo las primeras que aparecían? ¿Eran completas u omitía parte de lo pedido? ¿Podía llegar a conocer todas las reglas subyacentes?

También tuve la percepción de que algunos cambios cuando se liberaba un nuevo modelo alteraban su diligencia y precisión. Era lo que yo llamaba "**comportamientos perezosos**". Aunque poco a poco era más concreto en las peticiones, me parecía que los agentes buscaban atajos: dejaban tareas incompletas o solo atendían una parte de lo pedido.

Era frustrante porque no podía determinar a qué se debía esa diferencia. Podía proceder del modelo utilizado, de su configuración, del contexto disponible o de mi forma de pedir el trabajo. Entre las posibles causas contemplaba una bajada de rendimiento y calidad por cuantización al aparecer modelos mejores.

La cuantización reduce la precisión numérica con la que se representan los parámetros de un modelo para **ahorrar memoria** o facilitar su ejecución.

También podía echarme la culpa a mí mismo por no leer con suficiente detalle las preguntas y respuestas del agente (si te pregunta demasiadas cosas acabas diciendo siempre "sí, sigue"). No sabía cuál de esas posibilidades explicaba lo que me estaba pasando.

El ritmo de trabajo se reducía y aumentaba la frustración al ver que se repetían errores de interpretación que daba por superados. Lo gracioso era que, cuando se lo reprochaba, el agente me decía que llevaba razón, aunque aquello que había incumplido ya figuraba en sus propias instrucciones.

Pasada la novedad, construir agénticamente deja de parecer magia y aparecen tareas mecánicas, periodos de **cansancio**, pérdida de atención, **saturación** e interrupciones.

Además, las esperas se hacían cada vez más desesperantes y me animaban a arrancar otros proyectos en paralelo. Eso añadía el esfuerzo de cambiar de un proyecto a otro y aumentaba la saturación mental. Incluso diría que te puedes acabar despistando porque te llamen más la atención los otros proyectos que la obligación que tienes en el proyecto principal.

Tenía que mantener demasiadas cosas en la cabeza: lo que había pedido, las decisiones que se habían tomado y los cambios que aún debía revisar.

Esa carga mental me hizo pensar en la importancia de **planificar el relevo en los equipos** de construcción antes incluso de empezar un proyecto: las personas pueden cansarse, aburrirse o saturarse rápidamente. Yo incluso pensaba que si tuviera ya un ayudante le diría lo que quería y que él se pegase con la máquina.

Por todos estos motivos, y otros más que veremos, creo que el sector de construcción del software se va a transformar mucho, pero tampoco tanto: **vamos a seguir queriendo que sea otro el que se tire muchas horas delante de la pantalla luchando con los agentes.**

Aunque pensemos que la IA va a acabar con gran parte de la comunidad de desarrollo, creo que seguirán interviniendo muchos perfiles: construcción, aunque con menos personas, pruebas, administración de sistemas, expertos en normativa, encargados de formación y soporte. Es posible que sus funciones cambien, pero sigo pensando que harán falta. **La oportunidad siempre se desplaza con el avance tecnológico.**

Esta dificultad para mantener la atención con los agentes no va a desaparecer con la próxima versión del modelo, es algo estructural, no coyuntural. Cualquier persona puede perder el hilo, cambiar de prioridad, dedicar su tiempo a otros proyectos o dejar de estar disponible, temporal o permanentemente. En este sector, además, la rotación es alta.

Una organización no puede depender de que una sola persona conserve todo el histórico y el contexto. Hace falta que otros perfiles puedan revisar, continuar y operar el sistema.

Es más, hace falta diferenciar lo que se quiere conseguir de lo que el agente te crea, el código. Necesitamos ficheros que den continuidad entre sesiones (la **especificación**, las decisiones y las **pruebas**), porque al cambiar de sesión o de ordenador no siempre recuperamos las decisiones con el contexto en que las tomamos.

En el caso de mi proyecto de fin de Máster (y en todos los proyectos posteriores construidos con IA) **empecé a pedirle al sistema que formalizase todo lo que pedía con especificaciones** (los agentes ya son muy listos y con esta instrucción tan vaga pueden hacer mucho). En ese momento empezó el proyecto a ganar control. Quería conservar esa información fuera del chat para poder mantener un ritmo alto sin perder el control sobre lo construido.

Peter Naur defendió en 1985 que **programar es construir una teoría**: el conocimiento decisivo de un sistema no vive en el código ni en la documentación, sino en las personas que lo han pensado, y por eso se degrada cuando el equipo cambia (Naur, 1985). Cuando trabajo con agentes, parte de las razones que discutimos queda repartida entre conversaciones. Quiero conservar esas decisiones y los ejemplos que las explican para que quien tome el relevo pueda reconstruir el razonamiento y sepa qué necesita preguntar.



Figura 2. Del entusiasmo inicial a conservar el trabajo fuera de la conversación

Lo que me pasó con Solap también me recordó una conversación con un amigo que trabaja en una administración pública. Ya tenían problemas cuando algunos departamentos resolvían sus necesidades con hojas Excel. Ahora, me decía, con la IA «les crecen setas por todos lados». Una persona voluntariosa, a veces con poca preparación técnica, encuentra una solución que le sirve. Otros empiezan a usarla y, al cabo de un tiempo, solo la sabe mantener quien la hizo. Mientras tanto, ¿quién se ocupa de los datos que maneja?

También pasa lo contrario, gente amante de su trabajo puede ser todavía ajena al potencial de la construcción agéntica de software. Yo le digo a mis antiguos compañeros anclados en el pasado que si todavía tocan código a mano creo que no se han dado cuenta del momento del tiempo que estamos viviendo.

Con esta afirmación invitaría a técnicos, al menos, a revisar la situación y hacerse esta pregunta: si están aprovechando el momento, si entregan el máximo valor a sus empleadores y si se preparan para el siguiente salto. Cada equipo tendrá que encontrar su propia respuesta.

Si eres muy técnico, debes recordar, aunque te haya gustado mucho programar, que no debemos confundir los fines con los medios. Cualquier técnica de construcción de software que genere un producto funcional y de calidad es encomiable, pero el software nunca ha sido un fin en sí mismo: sirve para resolver un problema. El código fuente es un instrumento y no tiene más mérito porque lo hayamos construido línea a línea. La propia **especificación** también es un medio para conseguir entregar valor. Lo que realmente importa es el comportamiento útil, comprensible y gobernable que necesita una organización.

A raíz de lo que me pasó con Solap empecé a construir SDDApp, mi propia aplicación para desarrollar software a partir de especificaciones. Ya tengo una primera versión. En este libro cuento por qué elegí guardar las decisiones del equipo junto al código y qué pruebas le pediría a un agente antes de aceptar sus cambios.

Soy consciente de que puede resultar una obra pesada para quien busque algo ligero y al grano. Comparto el recorrido con la esperanza de que resulte útil a quienes quieran adquirir un conocimiento profundo, aunque intento escribirlo en un lenguaje asequible.

Tengo **intención** de hacer un doctorado sobre la construcción agéntica de software. Me gustaría partir del trabajo que recojo en este libro para investigar con más rigor cómo se combinan las **especificaciones** y los agentes al desarrollar un programa.

Sobre este documento

Los asistentes de programación basados en **grandes modelos de lenguaje** han cambiado la economía de escribir software. Una persona puede explorar una idea, generar código y corregirlo en ciclos que antes exigían más gente, más tiempo y más coordinación. Esa posibilidad está al alcance incluso de quienes tienen poca formación en ingeniería.

El conocimiento técnico sigue haciendo falta para reconocer cuándo una solución es insegura, difícil de mantener o cara de operar. En este libro estudio cómo puede ayudarnos el **desarrollo basado en especificaciones** (**SDD**, del inglés *Specification-Driven Development*) a construir y revisar ese software.

En este libro, llamamos **agente** a una herramienta de IA generativa que interpreta instrucciones y puede proponer o modificar código dentro del **entorno** autorizado, como **Claude Code** o Codex.

A QUIÉN SE DIRIGE ESTE LIBRO

A toda persona interesada en construir software de calidad. Es un libro técnico, pensado para quien quiera conocer con profundidad y detalle las técnicas de construcción de software. Seguir el razonamiento no exige escribir una sola línea de código, pero sí detenerse a entender cómo funciona cada pieza.

Se presuponen unas nociones básicas de **repositorios**, código y pruebas, aunque las repasaremos al revisar los antecedentes. Los términos más específicos se explican cuando los necesitamos y se recogen también en el glosario, para que sea fácil volver a consultarlos sin interrumpir la lectura.

Sea cual sea el modo de construcción, seguimos necesitando saber si el sistema hace lo que pretendíamos. Y no solo mientras lo construimos: si dentro de un año alguien nos audita, tendremos que poder contestar a esa misma pregunta.

Para poder comprobarlo, su comportamiento debe quedar por escrito en una especificación que podamos revisar. Las pruebas ejecutadas en cada integración deberán contrastar el código con esa referencia, y el registro de decisiones permite recuperar por qué acordamos cada regla.

Necesitamos, además, que los resultados sean reproducibles y que ningún cambio se integre sin superar las comprobaciones obligatorias. También debemos poder recuperar el trabajo tras una interrupción. Para ello hace falta conservar su estado fuera de la sesión y disponer de una opción de vuelta atrás.

Una petición ambigua puede convertirse en cientos de líneas de código plausibles. Al probar la aplicación quizá parezca que el agente ha entendido el encargo, aunque haya dejado una condición pendiente o haya elegido por su cuenta una regla importante. Para averiguar quién pidió esa regla y por qué se aceptó, hace falta conservar las decisiones junto al proyecto. Un registro separado puede perderse o quedarse atrás a medida que cambia el sistema.

La figura recoge los criterios que usaré para comparar las herramientas. Me interesa saber cómo conservan la relación entre lo que pedimos, las decisiones que tomamos y el código que recibimos, incluso después de cerrar la conversación. En algunos módulos estudiaré también qué haría falta para volver a generar el código a partir de esas fuentes.



Figura 3. Qué distingue a un sistema fiable de uno frágil.

Hay varias formas de trabajar con especificaciones, que veremos con detalle. En **spec-first**, se escribe la especificación antes de programar y sirve para guiar el desarrollo, aunque después puede quedarse desactualizada. En **spec-anchored**, la especificación se mantiene junto al código y las pruebas avisan cuando ambos dejan de coincidir. La tercera modalidad es **spec-as-source**, que podemos entender como “la especificación como fuente”. En ella, el código se trata como una salida que puede volver a generarse a partir de la especificación y que no se edita a mano.

En este libro propongo mantener el código del proyecto junto a la especificación **ratificada** y comprobar cada cambio con pruebas que el equipo haya acordado. Para algunas piezas

concretas, como el cálculo de cuántas bolas caben en juego, propongo generar el código a partir de reglas precisas. En el capítulo 7 llamaré a esa combinación **spec-anchored con núcleo determinista**. El libro parte por ello de la construcción agéntica de un juego sencillo y se apoya después en prácticas de ingeniería que permiten describir, comprobar y conservar su comportamiento.

Con este texto me propongo:

- **Fundamentar.** Explicar qué problema resuelve SDD, qué técnicas hereda y qué mecanismos sostienen sus garantías.
- **Contrastar herramientas representativas.** Examinar **GitHub Spec Kit, AWS Kiro**, **OpenSpec**, **Tessl y Structura** por sus mecanismos: **artefactos**, responsabilidades, controles y **verjas**.
- **Explicar el diseño de SDDApp y especificar qué debería poder hacer una aplicación de este tipo.** Definir una aplicación web personal, complementada con una **CLI** o interfaz de **comandos**, que cualquier persona pueda usar para construir software basado en especificaciones. Su núcleo de conocimiento estará en **Markdown**, un formato de texto plano con marcas sencillas para dar formato, muy utilizado en documentación técnica. Se podrá recorrer con Obsidian y estará gobernado por pruebas y por la **integración continua** o CI. La aplicación combinará mecanismos seleccionados de las herramientas analizadas, sin depender de una sola y conservando la **trazabilidad** y el rigor. Con esa especificación, cualquier lector podrá construir la suya o usar directamente las peticiones al agente que vamos reuniendo, sin construir nada.

Términos para leer la comparación

En este libro, un artefacto es cualquier resultado versionable del proceso: una intención, una especificación, un **ADR** (registro de una decisión de arquitectura), una lista de **tareas**, una prueba o un cambio de código.

Llamaremos **control asistido** a una revisión hecha por una persona o por un modelo; puede descubrir errores, pero su veredicto no es una prueba reproducible.

Llamaremos **verja** a una comprobación ejecutable que, al conectarse a una **política** obligatoria de integración, puede impedir que un cambio avance. Por ejemplo, las pruebas pueden ejecutarse automáticamente antes de integrar una entrega de código. Si una prueba figura como obligatoria en la política de la **rama** y falla, la plataforma bloquea el botón de integrar y el cambio no puede entrar en la **rama** principal hasta que se corrija el problema. Esta capacidad de bloquear distingue a la verja del control asistido y será central al analizar las herramientas.

Llamaremos **oráculo**, como es habitual en las pruebas de software (Barr et al., 2015), a la regla que fija qué resultado se considera correcto. Es el criterio con el que comparamos lo que produce el código. Si una función calcula un precio con IVA, el oráculo puede ser el resultado esperado que escribimos en la prueba: para 100 euros al 21 % debe devolver 121 euros. Si el código devuelve otra cifra, la prueba falla.

Si esperamos 120 euros en la prueba anterior, una implementación equivocada podría pasarla y la correcta fallaría. La verja solo resulta útil cuando el criterio de comparación, su oráculo, expresa bien el resultado que queremos proteger.

En adelante, **aprobar** significará permitir que una herramienta continúe; **ratificar** significará aceptar el significado de una versión exacta y convertirla en referencia autorizada. Una propuesta puede estar aprobada para seguir siendo examinada sin estar ratificada como regla vigente.

Tampoco necesitamos revisar con la misma profundidad un cambio de color y una modificación de las reglas de puntuación. El riesgo, el alcance y la posibilidad de deshacer cada cambio determinan qué pruebas, revisión y presupuesto exigiremos. Llamaremos **perfil de aseguramiento** al conjunto de esas exigencias. Es una decisión distinta de elegir un modo automático o guiado, que describe cómo interactúa la persona con la herramienta.

Cómo está organizado

- La **Parte I** establece el problema, las técnicas heredadas y el espectro de adopción.
- La **Parte II** compara GitHub Spec Kit, Kiro, OpenSpec, Tessl y **Structura** con los mismos criterios y utiliza el cambio de **multibola** para seguir sus **recorridos**.
- La **Parte III** explica por qué un **repositorio** Markdown y Obsidian sirven como memoria navegable.
- La **Parte IV** ordena las peticiones al agente en un proyecto preparado para Claude Code y termina en los **requisitos** de una aplicación de SDD.
- El estudio desemboca en un recorrido de uso: preparar el proyecto, construir una primera versión de **Arkanoid**, comprobarla y continuar con cambios posteriores. En ese recorrido situamos la recuperación, la colaboración y la memoria, junto con los criterios que permitirán evaluar una aplicación de SDD.
- El glosario y las referencias cierran esta copia. El glosario sirve para consultar los términos; las explicaciones necesarias aparecen junto a cada concepto en el cuerpo del libro.

  Aquí distinguimos el análisis de herramientas y los ejemplos comprobados de las capacidades que propongo para una aplicación de SDD.

El libro incluye numerosas infografías, con la intención de que quien busque una lectura rápida pueda recorrer primero las figuras y detenerse después en los apartados que le aporten más valor.

Es un libro voluntariamente amplio porque hacemos un estudio profundo de antecedentes y de herramientas para explicar las decisiones que tomé al diseñar SDDApp. **¡La ingeniería de software es amplia y tenemos que ponerla en valor!**

Los ejemplos utilizan Claude Code para que podamos seguir el recorrido en un mismo entorno de trabajo. Cuando procede, comparamos sus mecanismos con los equivalentes de otros asistentes.

Convenciones de lectura

Explicamos el juego en castellano. En el Arkanoid usamos nombres en inglés para los identificadores del código y de los **contratos**. Por ejemplo, cuando se destruye un ladrillo, el juego emite el **evento** `BrickDestroyed`. Los nombres oficiales, comandos, rutas y ficheros se reproducen tal como aparecen en la herramienta o en el proyecto.

Las listas que se citan desde otros capítulos llevan un identificador con un prefijo propio, de modo que un número no pueda confundirse con el de otra lista. Las peticiones al agente empiezan por P, los mecanismos que nos llevamos de cada herramienta por M, los elementos del desarrollo agéntico por E, los huecos del proyecto preparado para Claude Code por H y los pasos del recorrido de uso por USO. Los requisitos del Arkanoid se escriben FR-001, y los de la aplicación añaden el área, como FR-RAT-001.

Disponibilidad de la última versión del libro

El libro tendrá numerosas modificaciones y revisiones a lo largo del tiempo. El proceso de construcción de SDDApp y lo que he aprendido con ella dan para otro libro.

Las herramientas del mercado también seguirán evolucionando, por lo que habrá que actualizar las partes que las describen.

Cuando esta edición esté disponible en Leanpub, quienes adquieran el libro allí podrán descargar las revisiones desde su biblioteca. La compra también contribuye a financiar el trabajo de mantenerlo al día.

Índice

Prefacio 3
Sobre este documento 8
Convenciones de lectura 12
Disponibilidad de la última versión del libro 12
Índice 13
Parte I — Fundamentos y motivación 19
1. El problema: del vibe coding al desarrollo basado en especificaciones 20
1.1. El caso de estudio: un clon de Arkanoid 24
2. Vibe coding en la práctica: construir Arkanoid sin especificación 27
2.1. El primer intento: una sola petición 27
2.2. Contexto persistente y capacidades reutilizables 30
2.3. El límite: contexto no es especificación 42
2.4. Reingeniería de especificaciones 43
2.5. La inconsistencia que justifica el enfoque 47
3. Las piezas que SDD hereda 53
3.1. Cómo encajan las piezas 54
3.2. Requisitos escritos y estructurados 60
3.2.1. La dificultad y el precio de abstraer 72
3.3. Historias de usuario 74
3.4. TDD y la infraestructura que convierte pruebas en una red de seguridad 79
3.4.1. Límites de la prueba y técnicas de mocking 84
3.4.2. La red de seguridad y el coste acumulado 87
3.5. BDD y Gherkin: comportamiento en lenguaje compartido 91
3.5.1. DDD: del lenguaje compartido al modelo del juego 96
3.6. Especificación mediante ejemplos 98
3.7. API-first y pruebas de contrato 101
3.8. Pruebas basadas en propiedades 110
3.9. ADR: conservar el porqué 116
3.10. EARS: patrones para reducir vaguedad 119
3.11. Un ejemplo integrado con diez técnicas 122
3.12. Qué se lleva SDD 125
3.13. Lo que ya podemos montar con Claude Code 127
4. Antecedentes: de los modelos al ciclo de vida con IA 132

4.1. MDA: el desarrollo dirigido por modelos 132
4.1.1. MDA cuando construimos con agentes 138
4.1.2. Aprovechar ideas de MDA con skills 140
4.2. AI-SDLC: la IA en el ciclo de vida del software 143
4.2.1. De dónde viene el término 143
4.2.2. Los elementos del desarrollo agéntico 144
4.2.3. Dónde encajan las especificaciones 148
5. Control de versiones, integración y entrega 152
5.1. Git: memoria y colaboración 152
5.2. Integración, entrega y despliegue continuos 159
5.2.1. Dónde se ejecuta la CI: Jenkins y GitHub Actions 161
5.2.2. Linters y puertas de calidad 163
5.2.3. Entrega continua y despliegue continuo 165
5.2.4. Vigilar, deshacer y conocer la procedencia 167
5.3. Inventario de componentes 168
5.4. Integración y entrega en local 171
5.5. Deshacer cambios cuando hay datos: esquemas y migraciones 176
6. Estudios recientes sobre SDD con agentes 181
6.1. Del punto de partida a la especificación 181
6.2. Requisitos no funcionales, constitución y seguridad 185
6.3. Diseño, contexto, planificación y memoria 188
6.4. Construcción, comprensión del código e integración 195
6.5. Entrega, operación y evolución del proyecto 199
7. El espectro: spec-first, spec-anchored y spec-as-source 206
7.1. spec-first 207
7.2. spec-anchored 207
7.3. spec-as-source 208
7.3.1. spec-as-source por reglas 208
7.3.2. Regeneración gobernada por especificaciones 211
7.4. Coste, mantenibilidad y soberanía tecnológica 214
7.5. Mi opinión sobre el espectro 215
8. La jaula determinista y el trinquete 218
8.1. Cuándo un test se convierte en diente 222

8.2. Lo que ya podemos montar para proteger lo ratificado 226
Parte II — Herramientas representativas y herencia técnica 231
9. Panorama y método de análisis 232
9.1. Ingeniería del entorno del agente: harness engineering 235
9.2. Construyendo software con modelos locales 252
9.3. Mecanismos disponibles en el entorno de trabajo 263
9.3.1. Construir Arkanoid paso a paso con Claude Code 267
10. GitHub Spec Kit 275
10.1. Ficheros, carpetas y comandos 275
10.2. Recorrido: multibola con Spec Kit 281
10.3. Matriz operativa 286
10.4. Capacidades y límites de Spec Kit 287
11. AWS Kiro 294
11.1. Especificaciones, steering, hooks y corrección 294
11.2. Recorrido: multibola sin salir del IDE 299
11.3. Capacidades y límites de Kiro 305
12. OpenSpec 310
12.1. Dos estados, una configuración y un grafo de artefactos 311
12.2. Recorrido: multibola como cambio aislado 315
12.3. Matriz operativa 319
12.4. Capacidades y límites de OpenSpec 320
13. Tessl: contexto versionado para los agentes de programación 324
13.1. El producto actual: Registry, plugins y skills 325
13.1.1. Un paquete de metodología para SDD 326
13.1.2. Las especificaciones y su trazabilidad ligera 326
13.1.3. Loops, revisión y riesgo del cambio 327
13.2. Recorrido: multibola con Tessl y Claude Code 328
13.3. Matriz operativa 332
13.4. Capacidades y límites de Tessl 332
14. Structura: modelado de dominio y compilación por reglas 338
14.1. Matriz operativa 342
14.2. Capacidades y límites de Structura 343
15. Comparativa y posicionamiento 348
15.1. Posición en el espectro 348
15.2. Las herramientas en los elementos del ciclo con agentes 351

15.3. Estrategia para construir una aplicación de SDD 357
15.4. Relación con el harness y con el conocimiento portable 361
Parte III — El núcleo de conocimiento: Markdown, enlaces y Obsidian 365
16. Del conocimiento disperso al conocimiento enlazado 366
16.1. El método Zettelkasten 367
16.2. El segundo cerebro 368
16.3. La unidad adecuada 373
17. Por qué Obsidian 381
17.1. Condiciones para un equipo 385
17.2. Directorio de trabajo y enlaces simbólicos 386
18. Un repositorio, cuatro representaciones 389
18.1. Intención y razonamiento — por qué 391
18.2. Especificación normativa — qué debería ocurrir 392
18.3. Comprobaciones y evidencia — qué comprobamos y qué resultado obtuvimos 392
18.4. Implementación y ejecución — qué ocurre hoy 393
18.5. La bitácora — cómo llegamos hasta aquí 393
18.6. Cuando los artefactos discrepan 396
Parte IV — Aplicar lo aprendido 398
19. Un proyecto preparado para trabajar con Claude Code 399
19.1. Qué cambia respecto a chatear 399
19.2. Los ficheros del proyecto 399
19.3. CLAUDE.md como mapa 400
19.4. Las skills del proyecto 401
19.5. Las plantillas que comparten las skills 407
19.6. Las skills, una por una 410
19.6.1. Partir de un proyecto que ya existe 410
19.6.2. Definir lo que se quiere 410
19.6.3. Diseñar el cambio 412
19.6.4. Construir y probar 413
19.6.5. Revisar y cerrar 414
19.6.6. Mantener el proyecto 415
19.7. Hooks y permisos 415
19.8. Subagentes y revisor independiente 419

19.9. Estado y relevo 420
19.10. La CI y la rama protegida 422
19.11. Un cambio completo: la multibola 430
19.12. Dónde ha quedado cada petición 432
19.13. Lo que esta estructura no resuelve 432
20. Requisitos de una aplicación de SDD 435
20.1. El recorrido mínimo de uso 435
20.2. Requisitos funcionales 446
20.3. Requisitos no funcionales 455
20.4. Prioridades y piloto 458
21. Cierre 461
Apéndice. Peticiones y fichas del proyecto 463
Índice de las 120 peticiones 463
Instrucciones de las 24 skills del proyecto 466
casos-limite 466
extraer-especificacion 467
caracterizar 468
proponer-cambio 468
aclarar 469
requisitos 470
calidad 471
escenarios 472
contratos-verificables 472
decisiones 473
modelo-dominio 474
planificar 475
escribir-pruebas 476
diente 477
corregir-fallo 477
regenerar 478
contrastar 479

cierre 480
incorporar-cambio 480
cambiar-regla 481
mantener-skills 482
portabilidad 483
versiones-coordinadas 484
resumen-cambios 485
Glosario 486
Referencias 505