Cartografía del dominio

Crónica, manual y catálogo del Domain-Driven Design

ANDONI ARROYO

ÍNDICE

Prólogo

Algo iba mal con la programación orientada a objetos

A finales de los noventa, la programación orientada a objetos había ganado. Java se había convertido en el lenguaje por defecto de la empresa, C# llegaba pisándole los talones, y los manuales universitarios habían sustituido los diagramas de flujo por las cuatro letras del paradigma: encapsulación, abstracción, herencia, polimorfismo. UML era el dialecto común. Booch, Rumbaugh y Jacobson habían fusionado sus notaciones en 1997 y publicado el libro azul que decoraba todas las estanterías. La industria parecía haber resuelto el problema del *cómo*.

Y sin embargo, los proyectos seguían fracasando.

Quien hubiera trabajado en un sistema empresarial mediano hacia el año 2000 reconocería el síntoma. Las clases se llamaban `OrderManager`, `CustomerService`, `InventoryHandler`. Tenían cien métodos cada una y dependían de otras quince clases parecidas. Los objetos del negocio —`Order`, `Customer`, `Product`— eran cáscaras: estructuras de datos con un *getter* y un *setter* por cada campo, sin más comportamiento. Toda la lógica vivía afuera, repartida entre los *managers*, los *helpers*, los *facades* y un puñado de *utilities* estáticas que nadie se atrevía a tocar. La sintaxis era orientada a objetos. La arquitectura, no.

Martin Fowler, que llevaba años acompañando estos proyectos como consultor, le pondría nombre en 2003: *anemic domain model*. Un modelo anémico parece haber sido vaciado por dentro. Tiene la forma de un modelo de objetos, pero ninguna de sus virtudes. Las reglas del negocio, que deberían vivir junto a los datos que protegen, se habían deslizado a la capa de servicios. Y desde allí, viajaban a la capa de presentación, a los procedimientos almacenados de la base de datos, a los *event handlers* del front. Cada regla aparecía duplicada en sitios incompatibles. Cuando una cambiaba —y siempre cambiaba— el cambio había que hacerlo cinco veces y a menudo se hacía solo cuatro.

Es el dolor del que nace todo lo que viene en estas páginas, un cuadro que cualquiera con unos años de oficio reconoce sin esfuerzo.

La explicación más cómoda era culpar a los frameworks. Los EJB de la primera generación eran, sin discusión, una atrocidad técnica. Los `BusinessDelegate`, los `SessionFacade`, los `DataAccessObject`: cada patrón del catálogo *Core J2EE Patterns* (Alur, Crupi y Malks, 2001) era una pequeña confesión de que algo no encajaba. Pero la explicación cómoda no era la verdadera. Frederick Brooks ya había avisado en *No Silver Bullet* (1986) de que la complejidad del software se divide en dos clases: la **accidental**, que viene de las herramientas y desaparece cuando estas mejoran, y la **esencial**, que viene del problema mismo y no se va con ningún cambio de tecnología. La industria, durante quince años, había estado luchando con la primera y fingiendo que la segunda no existía.

La complejidad esencial estaba en el negocio. En las reglas tácitas que conocía la gente que llevaba veinte años despachando seguros, en las excepciones que aparecían cuando un envío internacional cruzaba tres aduanas, en los matices que distinguían una baja voluntaria de una baja por inactividad. Esa complejidad no la resolvía ningún *Application Server*. Pero alguien, en algún momento, tendría que escribir código que la respetara. Y para eso, el modelo de objetos —el de verdad, el que Alan Kay había imaginado y Smalltalk había practicado— seguía siendo la mejor herramienta disponible. Solo que la industria, ocupada con los patrones de infraestructura, lo había olvidado.

Algunos no lo habían olvidado. Kent Beck había publicado *Smalltalk Best Practice Patterns* en 1996, recordando que el código tiene que *contar* lo que hace antes que *funcionar*. *Refactoring* (1999), de Fowler con prólogo de Beck, había convertido en disciplina algo que hasta entonces era arte. Y en 2002, en *Patterns of Enterprise Application Architecture*, Fowler había hecho un trabajo curioso y a la vez decisivo: dar nombre a las cosas. *Domain Model*, *Service Layer*, *Repository*, *Data Mapper*, *Identity Map*, *Unit of Work*. Cada patrón era el destilado de prácticas que la comunidad llevaba años usando sin saber que tenían nombre. *PoEAA* era un mapa del territorio que nadie había dibujado antes.

Faltaba alguien que pusiera el mapa en su sitio.

En abril de 2003, en un manuscrito firmado en California, **Eric Evans** terminó un libro de cuatrocientas páginas con un título que no iba a quedar bien en ninguna estantería: *Domain-Driven Design: Tackling Complexity in the Heart of Software*. La portada era un Kandinsky abstracto. El prólogo lo firmaba Fowler. Adentro había agregados, contextos delimitados, lenguaje ubicuo, modelos refactorizados, mapas de contexto. Ninguno de esos términos existía en el vocabulario de la industria.

Quienes lo leyeron entonces no sabían qué hacer con él. Era denso. Repetía. Mezclaba metáforas con UML, anécdotas de proyectos de transporte aéreo con discusiones sobre teoría de tipos, fragmentos de Java hoy anticuados con citas a Christopher Alexander y a Brooks. Mucha gente que lo abrió no llegó al final. Algunos volvieron al cabo de un año y entendieron lo que la primera lectura les había escondido.

La tesis no era nueva, pero estaba dicha con una precisión que hasta entonces nadie había alcanzado. **El corazón del software complejo no está en la tecnología; está en el dominio.** Si el código no captura ese dominio con honestidad, ningún framework, ningún patrón, ninguna metodología compensará la deuda. Y para capturarlo hace falta más que una habilidad técnica: hace falta una conversación sostenida en el tiempo entre quienes conocen el negocio y quienes escriben el código, una conversación que produzca un único lenguaje compartido —Evans lo llamó *ubiquitous language*— y un modelo en código que hable ese lenguaje letra por letra.

A partir de ahí, el libro derivaba consecuencias. Si el modelo importa, hay que protegerlo de la infraestructura: surgió la idea de aislarlo en una capa propia. Si los objetos son la encarnación del modelo, hay que distinguir cuáles tienen identidad y cuáles no: nacieron las **entidades** y los **value objects**. Si la red de objetos puede crecer hasta volverse ingobernable, hay que dibujar fronteras transaccionales: aparecieron los **agregados**. Si el modelo sirve para uno, dos, tres equipos a la vez, hay que aceptar que cada equipo modela una parte distinta y darle nombre a esa parte: nacieron los **bounded contexts** y los **mapas de contexto**.

Evans no inventó casi ninguna de esas ideas en sentido estricto. Las entidades y los objetos valor venían de los patrones de Alexander y de la cultura Smalltalk. Los repositorios circulaban en *PoEAA*. La idea de un lenguaje común con los expertos del negocio era folclore Agile. Lo que Evans hizo —y por lo que el libro pasaría a la historia— fue tejerlas en un solo cuerpo doctrinal y mostrar que, cuando se aplican juntas, dejan de ser patrones aislados y se convierten en una **forma de pensar el software**.

Esa distinción —entre conocer DDD como catálogo de patrones y tenerlo enraizado como forma de pensar— acabaría siendo la línea divisoria entre dos lecturas posibles del libro. Una se quedaría en el *cómo*. La otra empezaría a entender el *porqué*.

Lo que pasó después es la historia de este libro. Tardó años en suceder. Durante una década, DDD fue un movimiento minoritario, un secreto que se contaban los buenos arquitectos en los pasillos. Hubo una resistencia fuerte: los frameworks ORM intentaron resolver

el dominio inyectando anotaciones, los *application servers* prometieron escalabilidad sin pedir nada a cambio, las metodologías predicaron que con buen *agile* y buenos tests bastaba. Mucha gente llegó a la conclusión de que DDD era un lujo académico para *enterprise developers* con demasiado tiempo libre.

Y entonces el ecosistema empezó a moverse.

En 2010, Greg Young, un desarrollador canadiense con un talento singular para la provocación útil, publicó un puñado de papers en abierto que separaban la lectura de la escritura en sistemas de dominio rico. Lo llamó **CQRS**: *Command Query Responsibility Segregation*. La idea era pequeña en apariencia y enorme en consecuencias. Si el modelo de escritura tiene que proteger invariantes y el de lectura solo tiene que servir vistas, ¿por qué obligarlos a vivir en la misma estructura? Sumado a eso, Young recuperó una idea más antigua, formulada por Fowler en 2005: que el estado de un sistema podía guardarse no como una foto, sino como una secuencia de cambios. **Event Sourcing.** Juntas, CQRS y Event Sourcing reabrieron la conversación sobre qué significa "persistir un objeto" y le dieron a DDD la maquinaria que necesitaba para escalar más allá del monolito clásico.

En 2013, **Vaughn Vernon** publicó *Implementing Domain-Driven Design*, el libro rojo, una obra de más de seiscientas páginas que hacía algo que Evans no había hecho: bajar al detalle. Las cuatro reglas para diseñar agregados, los patrones de integración entre bounded contexts, la convivencia con sistemas legacy, el código real para repositorios y application services. Si Evans había escrito el manifiesto, Vernon escribió el manual. La comunidad pasó de tener una filosofía a tener una práctica.

En 2014, en talleres de Italia y Polonia, un consultor llamado **Alberto Brandolini** empezó a pegar post-its naranjas en paredes para reconstruir el flujo temporal de los eventos del negocio con expertos del dominio. Lo llamó **Event Storming**, lo difundió en charlas, y dos años después la mitad del mundo DDD había hecho al menos un taller. Había ocurrido algo importante: DDD dejaba de ser solo una manera de escribir código y pasaba a ser también una forma de descubrir qué código hay que escribir.

En 2020, la comunidad recopiló su propia historia en un libro coral, *Domain-Driven Design: The First 15 Years*, publicado en Leanpub. Era una retrospectiva. Vernon, Brandolini, Verraes, Tune, Avram, Cockburn —los nombres que habían empujado el movimiento durante quince años— escribían capítulos cortos sobre lo aprendido. Una de las líneas que más se repetía: *casi todo lo que llamamos microservicios bien hechos, en realidad son bounded contexts*. La industria, sin saberlo, llevaba años haciendo DDD por la puerta de atrás.

En 2021, **Vlad Khononov** publicó *Learning Domain-Driven Design*, el libro que le faltaba a la comunidad para enseñar DDD a quien empezaba ahora. Khononov reordenó el material de Evans y Vernon con vocabulario contemporáneo: heuristic, complexity, alignment. Por primera vez, el lector novel tenía un manual moderno que no necesitaba traducir mentalmente desde el dialecto de los 2000.

Y mientras tanto, en paralelo, **Scott Wlaschin** demostraba en *Domain Modeling Made Functional* (2018) que el núcleo de DDD —proteger invariantes, hacer que los estados ilegales sean *imposibles de representar*, expresar el dominio en el código— florece igual de bien (o mejor) en lenguajes funcionales. **Khalil Stemmler** llevaba todas estas ideas al frontend, donde la industria había decidido por inercia que "el front es solo presentación". Y nuevas voces —Mathias Verraes, Cyrille Martraire, Nick Tune— iban añadiendo capas: *living documentation*, *team topologies*, *strategic monoliths*.

Al cierre de 2025, DDD ya no era un secreto. Se había convertido en una de las pocas disciplinas de diseño que sobrevive a tres olas de tecnología sin envejecer. La razón es la que Evans había escrito en la primera página del Blue Book: **el corazón del software complejo no está en la tecnología, está en el dominio**. Las tecnologías cambian; los dominios solo se vuelven más complejos.

Este libro cuenta esa historia y la convierte en herramienta.

Cuenta la historia porque entender DDD sin saber **de qué dolor nació, qué problema vino a resolver cada idea y por qué la comunidad fue añadiendo capas** es quedarse en la superficie. Es la diferencia entre saber recitar las cuatro reglas de Vernon para los agregados y saber elegir las fronteras de un agregado nuevo en un dominio que nadie ha modelado todavía. Esa elección no se aprende en una lista. Se aprende sintiendo en el cuerpo qué incomodidad estaba intentando resolver Vernon cuando formuló la regla.

La convierte en herramienta porque, una vez asentada la historia, cada concepto está fichado: definición precisa, dónde vive en la arquitectura, de qué puede depender, cuándo brilla, cuándo es la respuesta equivocada, con qué se confunde, qué señales indican que se está aplicando mal. Las fichas son la otra mitad del libro. Una vez completada la primera lectura, el catálogo queda como referencia consultable durante años. No están escritas para entretener; están escritas para resolver discusiones de pasillo.

El precio de un modelo anémico, de un bounded context que nunca se dibujó, de un microservicio que se montó antes de modelar, de una clase llamada `Manager` cuyo origen nadie recuerda — todos esos son síntomas que en estas páginas dejan de ser una nube

y empiezan a separarse, uno a uno, en piezas con nombre. No es un libro que prometa resolverlo todo. Es un libro que ofrece un vocabulario, un mapa y un instinto donde antes había ruido.

El libro alterna dos voces, y conviene reconocerlas desde aquí. Cuando un patrón entra en escena, lo hace dentro del relato histórico, situado en el momento y el problema que lo trajeron al mundo. Esa es la voz de la crónica. Cerca, al final del capítulo correspondiente, aparece la otra: una **ficha** seca, sin metáforas, con definición precisa, reglas operativas, fronteras, dependencias, antipatrones y fuentes. Las fichas reunidas forman el catálogo final, indexado por nombre y por problema que resuelve, pensado para sostenerse sin necesidad de volver al capítulo.

Empezamos.

. . .

ACTO I — La complejidad como enemigo

Capítulo 1 — El suelo en el que nació DDD

1986–2002

I. La promesa rota de los objetos

En 1972, en el Xerox Palo Alto Research Center, un grupo dirigido por Alan Kay diseñaba un lenguaje pensado para ser hablado por niños. Lo llamaron Smalltalk. Su tesis era sencilla y radical: el software debía construirse con objetos que se enviaban mensajes, igual que las personas se hablan; cada objeto era una pequeña máquina autónoma con su estado y su comportamiento, indistinguible para quien la usaba de un programa entero. La belleza de la idea sedujo a varias generaciones de programadores. Su consecuencia industrial, sin embargo, tardó dos décadas en llegar y, cuando llegó, ya casi no se reconocía.

A finales de los noventa, "orientación a objetos" significaba otra cosa. Significaba Java —que en 1995 había irrumpido con la promesa de "write once, run anywhere"— y, poco después, C# y la plataforma .NET. Significaba UML, que en 1997 había unificado las notaciones rivales de Booch, Rumbaugh y Jacobson en un solo lenguaje gráfico que pronto adornó cada presentación de arquitecto en cada empresa. Significaba *application servers*, *containers de servicios*, *frameworks de inyección de dependencias*. Lo que Alan Kay había imaginado como un lenguaje para que los niños construyeran sus propios mundos se había convertido en la herramienta industrial dominante para escribir sistemas empresariales que costaban millones y tardaban años.

Y los proyectos seguían fracasando. Standish Group publicaba cada poco su *CHAOS Report* con porcentajes deprimentes: la mayoría de proyectos de software empresarial salían tarde, salían mal o no salían. Los frameworks crecían más rápido que los problemas que pretendían solucionar. Cada conferencia técnica anunciaba una nueva versión de un nuevo *pattern* que prometía, esta vez sí, escalar a los problemas reales del negocio.

Quien hubiera abierto el código fuente de un sistema empresarial mediano hacia el año 2000 reconocería el cuadro: clases con nombres terminados en `Manager`, `Service`, `Helper`,

`Handler`; métodos con doce parámetros y trescientas líneas; herencia de cuatro o cinco niveles de profundidad para reutilizar dos métodos; objetos del negocio que eran solo cáscaras —`Customer`, `Order`, `Invoice` con un *getter* y un *setter* por cada campo y nada más—; toda la lógica esparcida entre los *managers* y los procedimientos almacenados de la base de datos. La sintaxis era orientada a objetos. La arquitectura, no.

II. Brooks revisitado

Para entender lo que ocurría hacía falta volver a un texto de catorce años antes. En 1986, Frederick Brooks Jr. —el autor de *The Mythical Man-Month* y veterano de IBM y de la Universidad de Carolina del Norte— había presentado en el congreso IFIP un ensayo titulado *No Silver Bullet: Essence and Accidents of Software Engineering*, reimpreso al año siguiente en *IEEE Computer*. Era un texto corto, de unas quince páginas, y tenía la forma de una predicción y una advertencia. La predicción: ninguna técnica, lenguaje o herramienta produciría en la siguiente década un aumento de productividad de un orden de magnitud. La advertencia: ninguna lo haría tampoco después, porque el problema no estaba donde la industria lo buscaba.

Brooks distinguió entre dos tipos de complejidad en el software. La primera, **accidental**, viene de las herramientas, los lenguajes, las plataformas. Es la dificultad que añade tener que pelear con sistemas operativos, con compiladores, con bases de datos, con redes. Esa complejidad es real, pero es contingente: mejora a medida que las herramientas mejoran. La segunda, **esencial**, viene del problema que el software intenta resolver. Una compañía de seguros, una red logística, un banco, un hospital: cada uno tiene una complejidad propia, irreducible, que ningún cambio de tecnología puede eliminar. El software, decía Brooks, es difícil porque es la única disciplina de ingeniería que se enfrenta directamente a la complejidad esencial: no hay leyes físicas que la simplifiquen; las reglas son las que el negocio quiere, y casi siempre son más complejas de lo que cualquiera admitirá.

Su tesis era incómoda. Implicaba que el progreso de las últimas décadas —compiladores mejores, lenguajes más expresivos, IDE, depuradores, *frameworks*— había sido en su mayor parte progreso sobre la complejidad accidental. La esencial seguía intacta.

A finales de los noventa, casi nadie en las salas de máquinas se acordaba de Brooks. Los EJB de la primera generación parecían el siguiente nivel de abstracción que cambiaría todo. *Enterprise JavaBeans 1.0*, lanzado en 1998, prometía persistencia, transacciones distribuidas, seguridad, *load balancing*: todo gestionado por el contenedor, todo configurable mediante descriptores XML. La frase “el contenedor se encarga” se había convertido en

mantra. En el siguiente lustro, los desarrolladores aprenderían qué significaba en la práctica: descriptores XML de mil líneas, *deployment* de quince minutos, *stack traces* incomprensibles, problemas de concurrencia que solo aparecían en producción.

III. Los rebeldes de Smalltalk

Mientras la industria enterprise se atascaba con sus contenedores, algunos programadores habían pasado los noventa cultivando una tradición distinta. Habían aprendido a programar en Smalltalk en los años en que aquello era posible, y luego habían pasado a Java o a C++ con la sospecha de que algo importante se había perdido en la traducción.

Kent Beck era una de las figuras centrales. En 1996 publicó *Smalltalk Best Practice Patterns*, un libro pequeño y engañosamente sencillo que recogía sesenta y dos patrones a nivel de método y clase. No eran los *Design Patterns* de la *Gang of Four*, publicados dos años antes. Eran reglas más finas: cómo nombrar los métodos privados, cuándo crear una variable temporal, cuándo extraer un método, cuándo usar un *parametric collaboration* en lugar de pasar argumentos. La tesis subyacente era que el código no era solo algo que tenía que funcionar; era algo que tenía que **comunicar**. Cada nombre, cada decisión, cada estructura era una elección sobre cómo el siguiente programador iba a entender el sistema.

Esa idea —el código como medio de comunicación entre personas— se había perdido en gran parte de la cultura enterprise. Los EJB, con sus interfaces *remote* y *home*, sus descriptores y sus *callbacks*, no comunicaban nada del negocio: solo su propia infraestructura. Beck representaba el contrapeso: un programador interesado en escribir clases pequeñas, métodos con nombres expresivos, código que pudiera leerse en voz alta.

Cerca de Beck estaba Ward Cunningham, que había inventado en 1995 el *wiki* —una idea que entonces parecía extraña: una web que cualquiera podía editar—, había sido pionero en CRC cards (*Class-Responsibility-Collaborator*) para el modelado de objetos en sesiones de pizarra, y había acuñado el término “deuda técnica” para hablar de la consecuencia financiera de las decisiones de diseño tomadas a la ligera. Cunningham era una figura de transición: pertenecía a la generación de los pioneros del Smalltalk pero estaba más interesado en la sociología del software que en su sintaxis.

Detrás de los dos, como un antepasado común que muchos citaban sin leerlo, estaba Christopher Alexander. Alexander era arquitecto, no programador. Su libro *A Pattern Language* (1977) era un catálogo de patrones de diseño urbano y arquitectónico, organizados desde la escala de la región hasta la de la habitación, con la idea de que la arquitectura era

un lenguaje compuesto por elementos recurrentes que cada constructor podía componer. Beck y Cunningham habían descubierto a Alexander en la década anterior y habían tomado de él dos cosas: la noción de **patrón** como unidad de diseño nombrable, y la convicción de que los patrones se descubrían en el uso, no se inventaban en el escritorio. Esa lectura cruzada —arquitectura urbana aplicada a software— sería, sin que nadie lo notara aún, una de las raíces más profundas de lo que iba a venir.

IV. La crisis de los EJB

A inicios del 2000, los Enterprise JavaBeans 1.0 y 1.1 estaban desplegados en buena parte del mundo enterprise. Las quejas eran constantes. Los desarrolladores que habían intentado usarlos habían descubierto que cada *Bean* requería un puñado de clases ceremoniales: la interfaz remota, la interfaz local, la interfaz *home*, la implementación, los *value objects* para no enviar entidades por la red, los *data access objects* para hablar con la base de datos sin pasar por la persistencia automática del contenedor (que era lenta y poco predecible). Una entidad simple del negocio, un *Customer*, requería entre seis y diez clases solo para empezar. La regla de los seis archivos por entidad se había convertido en folclore.

En 2001 apareció un libro que recogió ese folclore como si fuera doctrina: *Core J2EE Patterns*, de Deepak Alur, John Crupi y Dan Malks, los tres ingenieros de Sun Microsystems. El libro catalogaba veintiún patrones que la comunidad había desarrollado para sobrevivir a los EJB: *Business Delegate*, *Session Facade*, *Data Access Object*, *Value Object*, *Service Locator*, *Composite Entity*, *Transfer Object Assembler*. Cada uno tenía su problema documentado, su solución, sus consecuencias. Leído hoy, el libro tiene la cualidad melancólica de un manual de supervivencia. Leído entonces, fue tomado como evidencia de la madurez de la plataforma.

Lo que casi ninguno de aquellos patrones describía era el dominio. Eran patrones de **infraestructura**: cómo navegar la complejidad accidental que el contenedor introducía, cómo aislarse de sus caprichos, cómo transferir datos sin que la red explotara. La complejidad esencial —la del negocio— no aparecía en ninguno. Los modelos del negocio dentro de aquellos sistemas eran, casi sin excepción, anémicos: bolsas de campos sin comportamiento, navegadas por *services* y *managers* que orquestaban operaciones sobre ellas. Las reglas del negocio se reproducían en *services*, en *triggers* de base de datos, en validaciones del lado cliente, en cálculos en hojas de Excel que nadie había integrado todavía.

Martin Fowler había estado mirando esos sistemas durante años. Llevaba la primera mitad de su carrera consultando para grandes empresas y la segunda escribiendo libros que

destilaban lo que veía. En noviembre de 2003 publicaría en su blog un texto corto titulado *Anemic Domain Model* donde le pondría nombre al fenómeno. Pero el patrón —en su versión sin nombrar— ya era universal en 2001.

V. Refactoring y la disciplina de Fowler

Fowler había publicado en 1999 *Refactoring: Improving the Design of Existing Code*, un libro que parecía técnico y resultó ser cultural. Su tesis: el código se mejora **mientras se modifica**, no en grandes reescrituras. Cada cambio funcional puede acompañarse de pequeñas transformaciones —extraer un método, renombrar una variable, mover un campo— que dejan el código en mejor estado. El libro catalogaba setenta y dos refactorizaciones con su mecánica precisa, sus *bad smells* y sus consecuencias.

Refactoring hizo dos cosas. La primera, dar nombre y mecánica a algo que algunos buenos programadores hacían intuitivamente: ahora se podía hablar de "extraer método" o "introducir parámetro" como operaciones legítimas con criterios objetivos. La segunda, y más sutil, instalar la idea de que el código es **mantenible** o **deteriorable** según las decisiones cotidianas, y que la disciplina importaba más que la genialidad puntual.

Tres años después, en 2002, Fowler publicó el libro que más influencia tendría sobre lo que estaba por venir: *Patterns of Enterprise Application Architecture*. *PoEAA*, como pasaría a llamarse, no inventó casi nada. Lo que hizo fue **dar nombre** a lo que la comunidad ya hacía. *Domain Model* —el patrón opuesto al *Transaction Script*, el modelo donde los objetos del dominio tienen comportamiento, no solo datos. *Service Layer* —una capa de aplicación que orquesta los casos de uso. *Repository* —una abstracción que oculta la persistencia detrás de una colección de objetos. *Data Mapper* —el contrapunto de *Active Record*, donde los objetos del dominio no saben de la persistencia. *Identity Map*, *Unit of Work*, *Lazy Load*, *Optimistic Offline Lock*.

El libro no era prescriptivo. No decía "haz esto". Trazaba un mapa de territorios donde cada patrón vivía y advertía de sus consecuencias. Al hacerlo, instaló un vocabulario. A partir de *PoEAA*, dos arquitectos podían sentarse y discutir si una capa concreta debía ser *Service Layer* o *Domain Model*, si conviene un *Repository* o un *DAO*, si la persistencia debía ser *Active Record* o *Data Mapper*. Antes de *PoEAA*, esas conversaciones se daban con palabras cambiantes y sin referencias compartidas. Después, todos los participantes leían el mismo libro y citaban las mismas páginas.

PoEAA fue, en cierto sentido, el último libro de la era pre-DDD. Era todavía un libro de **infraestructura para el dominio**: catalogaba las piezas que rodeaban al modelo, no el modelo en sí. Pero al nombrarlas, dejó preparado el terreno para que alguien viniera a poner el modelo en el centro.

VI. Los movimientos paralelos

Mientras Fowler escribía sobre patrones de aplicación, otra conversación distinta estaba ocurriendo en la comunidad de programadores. En febrero de 2001, diecisiete personas se reunieron en una estación de esquí de Utah —el complejo Snowbird— para discutir qué tenían en común sus diferentes formas de programar. Algunos venían de Smalltalk, otros de XP, otros de Scrum, otros de FDD o DSDM. Salieron de la reunión con un documento de cuatro frases —el *Manifesto for Agile Software Development*— y doce principios. Lo firmaron Kent Beck, Robert C. Martin, Martin Fowler, Ward Cunningham, Ron Jeffries, Alistair Cockburn, Jim Highsmith y los demás. Era un documento minimalista, casi un poema. Decía que valoraban más a los individuos y sus interacciones que a los procesos y herramientas, más al software funcionando que a la documentación exhaustiva, más a la colaboración con el cliente que a la negociación contractual, más a responder al cambio que a seguir un plan.

El Manifiesto Ágil se interpretaría durante años de mil maneras —y se prostituiría en otras tantas certificaciones de Scrum vendidas a corporaciones que querían "agilidad" sin tener que cambiar nada—, pero algunas de sus consecuencias fueron permanentes. La idea de iteraciones cortas, de retroalimentación constante con el negocio, de pruebas automatizadas (*test-driven development*, también de Beck) se asentaron en cada vez más equipos. La cultura de la programación dejó de presentarse como disciplina solitaria de genios y se reconoció a sí misma como práctica social.

Eso importaba para lo que venía. Si DDD iba a ser una propuesta sobre **conversaciones entre programadores y expertos del dominio**, necesitaba un caldo de cultivo donde esa conversación fuera concebible. En 1995 era utópica; en 2003 era plausible.

A los movimientos del manifiesto se sumaron otras corrientes paralelas. Alistair Cockburn estaba pensando ya en lo que en 2005 llamaría *Hexagonal Architecture* —la arquitectura de puertos y adaptadores que invertía la dependencia entre dominio e infraestructura—, una idea que solo cuajaría más tarde pero cuyas raíces estaban en sus charlas de finales de los noventa. Bertrand Meyer había publicado en 1988 *Object-Oriented Software Construction*, donde formulaba el principio de **Command Query Separation**: una operación o cambia el estado o devuelve un valor, nunca las dos cosas. Greg Young, que en 2001 era todavía

un desarrollador anónimo, leería ese principio y lo elevaría una década después a nivel arquitectónico bajo el nombre de **CQRS**. Bertrand Meyer también había nombrado el *Design by Contract* y los invariantes; ambos conceptos quedarían anclados en el vocabulario que DDD heredaría.

Mientras tanto, en Bélgica y Países Bajos, en una pequeña comunidad de consultores de proyectos de seguros y bancos, había gente trabajando con sesiones de modelado en pizarra que ya empezaban a parecer lo que años después se llamaría *Event Storming*. No tenían el nombre, no tenían el método sistematizado, pero la práctica —pedir a los expertos que contaran el flujo del negocio, dibujarlo en orden temporal, descubrir el lenguaje en el proceso— empezaba a circular entre quienes habían descubierto que las reuniones de requisitos tradicionales producían más documentación que entendimiento.

VII. Eric Evans, el consultor

En el centro de esta confluencia de corrientes, casi en silencio, estaba Eric Evans. En 2002 era un consultor independiente que había pasado los noventa trabajando en proyectos complejos: sistemas de transporte aéreo, plataformas de trading, software hospitalario, integraciones bancarias. Tenía algo de programador, algo de modelador y mucho de lingüista. Su empresa unipersonal se llamaba *Domain Language*, un nombre que prefiguraba lo que vendría.

Evans había llegado al software por un camino oblicuo. Había estudiado matemáticas, había trabajado como tester, había pasado por varias consultoras antes de independizarse. Sus mentores fueron, en mayor o menor medida, los mismos que habían dado forma a la cultura: Fowler, Cunningham, Beck. De ellos había aprendido a tomar la modelación en serio: no como una tarea preliminar al desarrollo sino como **el desarrollo mismo**.

Lo que distinguía a Evans no era una técnica nueva. Era una pregunta que se hacía constantemente: ¿qué dice realmente el experto del dominio cuando explica un proceso? Y luego: ¿el código que estamos escribiendo dice lo mismo? Aplicada con disciplina durante años de proyectos, esa pregunta había llevado a Evans a una serie de descubrimientos que en privado llamaba con nombres provisionales: *aggregates* para los grupos de objetos que cambiaban juntos, *bounded contexts* para los límites donde un mismo término significaba cosas distintas, *ubiquitous language* para el vocabulario compartido entre equipo y experto.

Hacia 2001, Evans empezó a escribir un libro. La motivación era pragmática: cada cliente nuevo pasaba por las mismas explicaciones. Documentar el método ahorraría tiempo. El

libro creció. Acumuló capítulos. Llegó a más de cuatrocientas páginas. La editorial Addison-Wesley aceptó publicarlo, y Martin Fowler se ofreció a escribir el prólogo.

Mientras escribía, Evans trabajaba en un proyecto de transporte naval que se convertiría en el caso de estudio principal del libro. El sistema gestionaba contenedores, rutas, puertos, escalas, transferencias de carga. Evans aprovechó el proyecto para probar cada una de las ideas que pretendía publicar. El experto del dominio era un veterano del sector con treinta años de experiencia, capaz de explicar las reglas tácitas que ningún manual recogía. Las sesiones de modelado entre Evans y aquel hombre serían, sin que ninguno lo supiera, uno de los momentos fundacionales de una disciplina entera.

En abril de 2003, en un manuscrito firmado en California, Evans terminó el libro. Lo subió a su servidor con la palabra “draft” en la portada. Las copias preliminares circularon entre amigos y colegas. La portada definitiva, cuando Addison-Wesley la imprimió a finales de año, sería un Kandinsky abstracto sobre fondo azul. El título, deliberadamente largo: *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

VIII. El terreno preparado

Visto en retrospectiva, todas las piezas estaban en su sitio cuando el libro salió. La cultura Smalltalk de Beck y Cunningham había cultivado la noción de código como comunicación. Brooks había advertido sobre dónde estaba la complejidad real. Fowler había puesto el vocabulario común con *PoEAA*. El Manifiesto Ágil había instalado la idea de la conversación constante con el cliente. Cockburn estaba pensando en hexágonos. Meyer había nombrado el *Command Query Separation*. El fracaso visible de los EJB había convencido a buena parte de la industria de que la abstracción técnica no era la respuesta.

Lo que faltaba era alguien que **tejiera todo aquello en un solo cuerpo doctrinal**. Que dijera, con todas las palabras: el corazón del software complejo no está en la tecnología, está en el dominio; y para llegar al dominio no hay atajo, hay que sentarse con los expertos, descubrir su lenguaje, modelarlo en código que hable ese lenguaje letra por letra. Que diera nombres a las piezas: entidades y *value objects*, agregados, repositorios, *bounded contexts*, mapas de contexto. Que mostrara cómo encajaban todas. Que escribiera, en definitiva, un libro que la industria no estaba esperando pero al que llevaba quince años caminando.

El libro de Evans tenía tres apartados grandes. La primera parte, *Putting the Domain Model to Work*, defendía la centralidad del modelo. La segunda, *Building Blocks of a Model-Driven Design*, presentaba los patrones tácticos. La tercera, *Refactoring Toward Deeper Insight*,

mostraba cómo el modelo evoluciona con el entendimiento. Una cuarta parte, *Strategic Design*, casi tan larga como las otras tres juntas, introducía los *bounded contexts*, los mapas de contexto, los subdominios, y cerraba el libro con la idea más ambiciosa de todas: que la arquitectura del software no se decidía por el diseño técnico sino por la organización del equipo y por la claridad con la que ese equipo modelaba la realidad del negocio.

Pocos lectores llegaron al final en la primera lectura. El libro era denso, repetía conceptos, mezclaba metáforas con UML y anécdotas de proyectos con citas a Christopher Alexander. La gente que lo abrió en 2003 dejó la mayoría de copias a medias. Algunos volvieron al cabo de un año y entendieron lo que la primera lectura les había escondido. Otros volvieron a los cinco años, después de haber sufrido un par de proyectos enterprise mal modelados, y entonces el libro se les apareció como evidencia de algo que llevaban tiempo intuyendo sin saber nombrar.

Lo que pasaría después tardaría en suceder. Durante una década entera, *Domain-Driven Design* sería un libro de culto, un secreto compartido entre arquitectos en pasillos, un nombre que aparecía en charlas pequeñas y conferencias regionales. Greg Young lo leería en 2005 y empezaría a darle vueltas a la separación entre lectura y escritura que, cinco años después, llamaría CQRS. Vaughn Vernon lo leería hacia 2004 o 2005 y empezaría a documentar cómo aplicar las ideas en sistemas reales, lo que en 2013 cristalizaría en *Implementing DDD*. Alberto Brandolini lo leería en torno a 2007 y empezaría a inventar la dinámica de pegar post-its naranja en paredes que, en 2014, la comunidad bautizaría como *Event Storming*. Cada uno de ellos, sin coordinación, descubriría que el libro de Evans no era un manual sino el inicio de una conversación.

Pero todo eso pasaría más tarde. En 2003, cuando las primeras copias salieron de la imprenta, había un libro incómodo, mal vendido al principio, recibido con perplejidad por una industria que esperaba el siguiente *framework* y se encontraba con un tratado sobre conversaciones con expertos del dominio. Ese contraste entre lo que la industria esperaba y lo que el libro ofrecía es el motor de toda la historia que viene a continuación.

Capítulo 2 — Lo que se hacía sin nombre

La práctica del modelado antes de que se llamara DDD

I. La pregunta que precede al método

La impresión común al leer *Domain-Driven Design* por primera vez es que Eric Evans inventó un método. Es una impresión equivocada. Lo que Evans hizo en 2003 fue **darle nombre y forma sistematizada a un conjunto de prácticas que ya circulaban**, en pequeños círculos, desde hacía al menos diez años. Para entender por qué el libro encontró audiencia tan rápidamente entre los pocos que lo leyeron en serio, hace falta mirar qué se estaba haciendo —sin nombre, sin tratado, sin certificación— en los proyectos que durante los noventa habían empezado a tomar la complejidad del dominio en serio.

Este capítulo no es sobre teorías. Es sobre el trabajo concreto de un puñado de consultores y arquitectos que se sentaban con expertos del negocio, dibujaban en pizarras, escribían tarjetas con nombres en lápiz, y descubrían que había una forma de modelar el software que no salía de los manuales de UML. Cuando Evans publicó su libro, esos consultores reconocieron en él **lo que llevaban años haciendo**. Reconocer lo propio formulado por otro convierte una intuición personal en disciplina compartida.

II. La cultura de los patrones (1993–1996)

En agosto de 1993, en una casa rural en el norte de Illinois, se reunió por primera vez un grupo de programadores que empezaba a leer obsesivamente a Christopher Alexander. Se llamaron a sí mismos *Hillside Group*. Entre los asistentes estaban Kent Beck, Ward Cunningham, Ralph Johnson (uno de los autores de *Design Patterns*, publicado el año siguiente), Grady Booch y Jim Coplien. La pregunta que los convocaba era directa: si Alexander había encontrado patrones recurrentes en la arquitectura urbana, ¿podían encontrarse patrones recurrentes en el software que pudieran nombrarse, catalogarse y enseñarse?

De aquella reunión salió la primera conferencia *PLoP —Pattern Languages of Programs —*, que se ha celebrado anualmente desde 1994. La idea original de Alexander era que los patrones formaran un **lenguaje**: cada patrón era una palabra; combinados, contaban historias coherentes sobre cómo construir mejor. Los programadores se quedaron con la primera mitad de la idea —los patrones individuales— y casi olvidaron la segunda. *Design Patterns* (Gamma, Helm, Johnson, Vlissides, 1994), el libro de los *Gang of Four*, fue el resultado más visible de aquella primera ola: veintitrés patrones técnicos catalogados con su forma canónica.

Pero Beck y Cunningham, los más cercanos a Alexander en su lectura, persistieron en la idea del **lenguaje completo**. Sus patrones eran de otra naturaleza: no resolvían problemas técnicos puntuales (cómo gestionar la creación de objetos, cómo separar interfaz de implementación), sino que describían **decisiones de modelado** del dominio. Cunningham había publicado en 1989, junto con Beck, un paper en OOPSLA titulado *A Laboratory for Teaching Object-Oriented Thinking*. En él introducían las **CRC cards**: tarjetas de cartón donde se escribía el nombre de una clase, sus responsabilidades y sus colaboradores. Eran tarjetas físicas, manipulables, baratas. Se usaban en sesiones grupales: el equipo se sentaba en torno a una mesa, repartía las tarjetas, simulaba el flujo de un caso de uso pasándose preguntas y respuestas entre los objetos representados por cada tarjeta.

CRC cards parecía un juguete, y a algunos arquitectos formales les resultaba ofensivamente simple. Pero en la práctica resolvían algo que UML no: **forzaban a las personas a hablar el dominio en voz alta**. La conversación era la unidad de modelado, no el diagrama. Las tarjetas se rehacían constantemente —escribir en lápiz era parte de la disciplina— porque cada conversación añadía o eliminaba responsabilidades. Una sesión de CRC cards de tres horas producía más entendimiento que una semana de diagramas UML elaborados en solitario.

Esa convicción —**que el modelo emerge en la conversación**— es una de las raíces directas de lo que DDD formalizaría una década después. El *Event Storming* de Brandolini, el *Domain Storytelling* de Hofer y Schwentner, el *Example Mapping* de Wynne: todos heredan de la cultura CRC card su confianza en que la pizarra y la conversación son herramientas de modelado serias.

III. Cockburn y el modelado por casos de uso

Mientras Beck y Cunningham trabajaban con CRC cards, Alistair Cockburn estaba sistematizando otra técnica complementaria. Cockburn había trabajado para IBM en el Reino Unido

durante los años ochenta, había estudiado lingüística y filosofía además de ingeniería, y traía una sensibilidad particular para la **narrativa**. Su pregunta era distinta a la de Beck: no "qué clases necesitamos" sino "qué hace el actor cuando interactúa con el sistema".

Su libro *Writing Effective Use Cases* (2000) fue la culminación de una década de práctica. Cockburn proponía describir el sistema como una colección de **casos de uso**: secuencias de pasos que un actor (humano o sistema) ejecuta para conseguir un objetivo. Cada caso de uso tenía un nivel de detalle (objetivo de usuario, sumario, subfunción), un actor primario, precondiciones, postcondiciones, un escenario principal y escenarios alternativos.

Lo que distinguía el enfoque de Cockburn era su insistencia en escribir los casos de uso **en lenguaje del negocio**. No "el sistema valida el formulario" sino "el cajero contrasta la firma con la registrada en el reverso de la tarjeta". La diferencia parece estética; en la práctica era operativa. Casos de uso escritos en lenguaje técnico pierden a los expertos del dominio; escritos en su lenguaje permiten que los expertos los lean, corrijan, completen.

Cockburn era además uno de los firmantes del Manifiesto Ágil. Sus *use cases* convivieron sin contradicción con la cultura ágil: **eran herramientas de conversación**, no contratos de especificación. Una iteración podía empezar con un caso de uso escrito a lápiz y terminar con código funcionando. El caso de uso evolucionaba con el código, no era una promesa rígida sobre lo que el código tenía que hacer.

Cockburn llevaba además años pensando en algo que solo formalizaría en 2005: el dominio del software debía estar **aislado** del exterior, comunicándose a través de **puertos** que adaptadores específicos implementarían. La arquitectura hexagonal todavía no tenía nombre, pero ya estaba presente en sus charlas y en su práctica de consultoría. Cuando, dos años después del libro de Evans, Cockburn formalizara el patrón en un breve artículo en su sitio personal, encontraría que las ideas de Evans y las suyas encajaban como dos piezas de un mismo puzzle.

IV. La escuela europea: Coad, Yourdon, Jacobson

Mientras la cultura de los patrones florecía en Estados Unidos, en Europa una tradición distinta estaba madurando. Edward Yourdon y Peter Coad publicaron en 1990 *Object-Oriented Analysis*, uno de los primeros libros que aplicaba sistemáticamente la orientación a objetos al modelado del análisis (no solo del diseño). La obra era didáctica, profundamente sistemática, y propuso una notación gráfica precursora de UML.

Más influyente fue el trabajo de Ivar Jacobson, ingeniero sueco que había trabajado durante los setenta en Ericsson en sistemas de telefonía. Jacobson había publicado en 1992 *Object-Oriented Software Engineering: A Use Case Driven Approach*, donde introducía la noción de **caso de uso** como artefacto central del proceso de desarrollo. Jacobson insistía en algo que Cockburn refinaría después: el sistema se entiende mejor mirándolo desde fuera, desde la perspectiva de quien lo usa, que desde dentro, desde la estructura interna de sus clases.

Las ideas de Jacobson, fusionadas con las de Booch y Rumbaugh, darían lugar a UML y al *Rational Unified Process* (RUP). Para 1999, RUP era la metodología enterprise dominante: un marco de trabajo extenso, documentado en cientos de páginas, con cuatro fases (concepción, elaboración, construcción, transición) y nueve disciplinas. RUP prometía industrializar el desarrollo de software mediante el modelado disciplinado.

La realidad fue distinta. Aplicado por consultorías como guía contractual, RUP producía toneladas de documentos UML que nadie volvía a leer una vez firmado el contrato. Los desarrolladores aprendieron a llamar al fenómeno *analysis paralysis*: equipos atrapados durante meses elaborando diagramas de casos de uso, de actividad, de secuencia, de despliegue, sin haber escrito una línea de código.

La reacción fue lo que ya sabemos: el Manifiesto Ágil de 2001. Pero entre los críticos del UML pesado había una distinción que la cultura ágil no siempre supo articular: **modelar es valioso; producir documentación de modelado para entregar a otro no lo es**. UML era útil cuando dos personas dibujaban en una pizarra durante una conversación; inútil cuando se convertía en archivo entregable. Algunos arquitectos —Cockburn entre ellos— mantuvieron viva la primera lectura: usar UML como herramienta de pensamiento, no como artefacto contractual. La distinción entre **modelado vivo** y **documentación muerta** sería otra de las raíces de DDD.

V. Las sesiones de pizarra que nadie publicaba

Durante los años noventa, en oficinas de consultoras grandes y en proyectos puntuales de empresas medianas, una pequeña tribu de consultores había empezado a desarrollar una forma de trabajo que no encajaba bien en ningún manual. La práctica, vista desde fuera, parecía improvisación: el consultor llegaba con post-its, marcadores y café, se sentaba con el experto del dominio, y la conversación duraba horas. La pared se llenaba de garabatos. Al final del día había un modelo dibujado a lápiz, una lista de preguntas abiertas, y a veces el primer esquema de las clases que se programarían.

Lo que esa tribu sabía y no había sistematizado era una serie de descubrimientos que hoy reconocemos:

- Los expertos del dominio **conocen reglas tácitas** que ningún manual recoge. Esas reglas solo aparecen cuando se discuten casos concretos, no cuando se piden requisitos en abstracto.
- Distintas áreas del negocio **usan los mismos términos con significados distintos**. "Cliente" en el departamento comercial no es el mismo "Cliente" en facturación; un consultor experimentado aprendía a no asumir que la palabra significaba lo mismo.
- El modelado **no termina nunca**. Cada conversación con un experto nuevo añadía matices, contradicciones o casos especiales que obligaban a revisar lo dibujado. La práctica era iterar, no completar.
- El código que se escribía después tenía que **hablar el lenguaje del modelo**. Si el modelo decía "el envío se asigna a una ruta" pero el código decía `routeAssigner.process(shipment)`, había una grieta que tarde o temprano produciría bugs por interpretación incorrecta.

Esa lista parece obvia hoy. En 1998 era folclore tribal. No había libros, no había certificaciones, no había conferencias —al menos ninguna popular— que lo enseñara. Los consultores que sabían trabajar así lo habían aprendido de mentores, de proyectos que habían sufrido, de leer a Beck y Cunningham entre líneas. Era un oficio, no una disciplina.

Eric Evans pertenecía a esa tribu. La intuición que le llevaría a escribir el libro fue la de que aquello podía **sistematizarse**. No para industrializarlo —el modelado vivo no se industrializa—, sino para que los profesionales jóvenes pudieran aprenderlo sin tener que pasar por los diez años de proyectos fallidos que los veteranos habían atravesado.

VI. Test-Driven Development como compañero de viaje

Mientras la cultura de modelado maduraba en pocas oficinas, otra disciplina avanzaba en paralelo: el desarrollo dirigido por tests. Kent Beck había estado practicando lo que hoy llamamos TDD desde los noventa, en proyectos Smalltalk y luego en Java. En 2002 publicó *Test-Driven Development: By Example*, que codificó la práctica para una audiencia amplia. La mecánica era simple: escribir un test que fallara, escribir el código mínimo para que pasara, refactorizar para limpiar el diseño. Repetir.

TDD parecía, a primera vista, una técnica de aseguramiento de calidad. Beck insistía en que era otra cosa: una **técnica de diseño**. Escribir el test antes que el código forzaba a tomar

decisiones de uso —cómo se invoca esta función, qué devuelve, cuándo falla— antes que decisiones de implementación. El resultado era código diseñado **desde fuera hacia dentro**, en un orden que evitaba jerarquías técnicas innecesarias.

TDD encajaba con la cultura del modelado en pizarra. La conversación con el experto producía un modelo conceptual; los tests traducían ese modelo a especificaciones ejecutables; el código emergía como consecuencia. La diferencia con RUP era radical: no había documentación intermedia. El modelo vivía en la pizarra; los tests vivían en el código; cualquier cambio en uno se propagaba inmediatamente a los otros.

Cuando en 2006 Dan North publicara *Introducing BDD* y diera nombre al *Behavior-Driven Development*, y luego Gojko Adzic publicara *Specification by Example* (2011), el ciclo se cerraría: los ejemplos descubiertos en sesiones con el experto se convertirían en escenarios ejecutables que verificarían el comportamiento del sistema. Pero esa formalización vendría una década después. En 2002 lo que había era TDD y conversaciones de pizarra coexistiendo en pocas oficinas, sin nombre común para el conjunto.

VII. El laboratorio de Evans: el caso de carga marítima

Entre los proyectos en los que Evans había trabajado durante los años anteriores al libro, uno se convertiría en el caso de estudio canónico: un sistema de gestión de carga para una compañía naviera. Los detalles concretos del cliente Evans nunca los hizo públicos —los consultores no suelen poder—, pero la naturaleza del dominio sí. Era un sistema que tenía que gestionar contenedores, rutas, escalas, transferencias entre buques, aduanas, documentación legal, asignaciones a clientes finales. Cada uno de esos conceptos tenía sus propias reglas, sus propios estados, sus propias excepciones.

El experto del dominio era un veterano del sector con treinta años de experiencia. Sabía, por ejemplo, que un contenedor podía estar “asignado” a una ruta en un sentido y a una expedición en otro, y que esos dos significados de “asignación” eran completamente distintos: el primero era una decisión logística que podía cambiar; el segundo era un compromiso comercial con el cliente que no se modificaba sin coste contractual. El equipo, en sus primeras sesiones, había usado la palabra “asignación” para los dos conceptos. La consecuencia fue que el modelo confundía decisiones logísticas con compromisos comerciales, y los bugs que aparecieron en producción tardaron semanas en diagnosticarse: las reglas que aplicaban dependían de cuál de los dos significados estaba en juego, pero el código no lo distinguía.

El descubrimiento, contado en el libro como un episodio entre otros, era el momento exacto en el que una de las ideas centrales de DDD se cristalizaba. Si el lenguaje del experto y el lenguaje del código no coinciden, se producen bugs por traducción. La solución no era pedirle al experto que aprendiera el lenguaje del código; era pedirle al código que hablara el lenguaje del experto. Evans empezó a llamar a este principio **ubiquitous language**: un único vocabulario, compartido por todos los participantes, materializado en el código.

El proyecto de carga le proporcionó también la primera experiencia práctica con los **bounded contexts**. El sistema tenía partes que servían a operaciones logísticas y partes que servían a operaciones comerciales. Los dos tenían modelos parecidos —ambos hablaban de contenedores, rutas, expediciones— pero las reglas que aplicaban eran distintas. Forzar un único modelo unificado era posible técnicamente, pero producía un modelo gigante con cien atributos por entidad y reglas condicionales por todas partes. Evans descubrió, por necesidad, que era más limpio mantener **dos modelos distintos**, conectados por traducciones explícitas. Cada parte del sistema vivía en su propio contexto; los modelos no eran iguales pero sí compatibles.

Esos descubrimientos no eran originales en sí. Otros consultores habían llegado a conclusiones similares en proyectos parecidos. Lo nuevo era **el nombre**. Evans empezó a usar “ubiquitous language”, “bounded context”, “aggregate” en sus charlas internas. Los términos cuajaron en su práctica y fueron viajando a otros proyectos donde Evans entraba como consultor o donde colaboradores suyos llevaban la metodología. Para 2001, antes de empezar a escribir el libro, ya tenía un vocabulario interno consistente. Para 2003, el manuscrito estaba listo.

VIII. La diferencia entre tener el método y nombrar el método

Lo que se hacía sin nombre durante los noventa funcionaba. Los consultores que sabían trabajar así entregaban sistemas que duraban más, que evolucionaban sin colapsar, que sus equipos podían mantener. La diferencia entre esos sistemas y los que producía el resto de la industria era visible para quienes los vivían, pero no había manera de **enseñar** la diferencia. Cada consultor tenía que reinventar el oficio. Cada equipo nuevo cometía los mismos errores. Cada proyecto que fracasaba lo hacía por razones que ya estaban catalogadas en la cabeza de los veteranos pero no en ninguna parte transferible.

Esa imposibilidad de transferir es lo que un libro resuelve. No porque contenga magia, sino porque convierte el folclore tribal en disciplina compartida. Cuando *Domain-Driven Design* salió en 2003, los consultores que llevaban años trabajando así se reconocieron en

él. Lo importante no era que las técnicas fueran nuevas —no lo eran, casi todas—, sino que tenían **nombres** y formaban un **cuerpo coherente**. Por primera vez se podía decir "esto que hago es DDD" en lugar de "esto que hago es difícil de explicar". Y por primera vez un junior podía leer cuatrocientas páginas y empezar a aproximarse, sin diez años de cicatrices, a la forma de pensar que los veteranos habían destilado.

El propio Evans era consciente de que el libro no era un manual completo. En el prefacio reconocía que muchas de las ideas no eran originales, que los patrones tácticos venían de la cultura del modelado de objetos de dos décadas atrás, que la idea del lenguaje compartido tenía precedentes claros en la cultura ágil. Lo que el libro aportaba —decía— era **la articulación de un cuerpo de prácticas como disciplina coherente**.

Esta distinción importa para entender por qué DDD se difundiría como lo hizo. Las disciplinas que se imponen como revelación suelen producir sectas. Las que se reconocen como articulación de algo conocido producen comunidades. DDD pertenece al segundo grupo. La gente que lo adoptó en los primeros años no sentía que aprendiera algo nuevo; sentía que por fin tenía nombres para lo que ya creía. Esa sensación de reconocimiento es la que mantiene a las comunidades unidas mucho tiempo. Es también la razón por la que veintitrés años después, en pleno 2026, DDD sigue teniendo tracción cuando otros movimientos contemporáneos —RUP, los EJB, el OO formal— se han convertido en piezas de museo.

. . .

Fin de la muestra

Aquí termina la muestra gratuita. El libro completo continúa con cinco actos más (Evans y el Blue Book, CQRS y Event Sourcing, Vernon y el manual operativo, diseño colaborativo, era moderna) y un anexo extenso con catálogo de patrones, árboles de decisión, glosario, índice analítico y bibliografía comentada.

Cartografía del dominio

> *“La complejidad esencial está en el dominio, no en la tecnología. Una disciplina que se ocupa de la complejidad esencial envejece lentamente.”*

Domain Driven Design cumplió veintitrés años. En ese tiempo dejó de ser un libro azul de culto entre arquitectos enterprise y se convirtió en disciplina madura: vocabulario común, conjunto de técnicas reconocibles, comunidad internacional.

Este libro cuenta esa historia y, a la vez, ofrece un manual de referencia para aplicarla. Una crónica que va de Eric Evans a Vlad Khononov pasando por Greg Young, Vaughn Vernon y Alberto Brandolini. Y un catálogo de patrones tácticos, estratégicos, arquitectónicos y de descubrimiento, con criterios de decisión articulados.

Para programadores que quieren entender de dónde vienen las técnicas que aplican; para arquitectos que necesitan vocabulario para justificar decisiones; para equipos que buscan empezar por el dominio sin caer en DDD-lite.

Trazar el mapa antes de construir: la disciplina más simple y más contracultural del software moderno.