Laboratorio de Ingeniería del Software

Diseño dirigido por el dominio (*Domain-driven design*)

Contenidos

- El dominio del problema
- Los modelos de dominio en el desarrollo de software
- Obtener conocimiento de un dominio
- Lenguaje y comunicación
- Documentos y diagramas
- Enlazar modelo e implementación

El dominio del problema

El dominio del problema

- El dominio de un problema es el área de conocimiento que hay que examinar para resolverlo
- En el caso de una aplicación software, es el área de conocimiento a la que se aplica este software
- En un sistema de control de versiones, el dominio del problema incluirá código fuente, texto o ficheros binarios
- En un programa de contabilidad, el dominio incluirá dinero y asientos contables
- En un programa de compra de entradas para conciertos, el dominio incluye fechas, asientos y salas de conciertos

Modelos de dominio

- Un dominio de problema puede incluir una cantidad abrumadora de información
- Un **modelo de este dominio** es una forma de representar una parte de esta información de manera estructurada y simplificada
- Un modelo no es un diagrama
  - Y un diagrama no es un modelo
- Un modelo es una abstracción de cierto conocimiento seleccionado, rigurosamente estructurada
  - Que podemos expresar con uno o más diagramas, explicaciones, fórmulas, código etc.
- Al crear un modelo de un dominio no buscamos el máximo realismo, buscamos representar una parte de la realidad de la manera que mejor se ajuste a nuestras necesidades
  - P.ej. desarrollar cierta aplicación de software

Pregunta

- ¿Cuántos modelos correctos de un dominio de problema podemos crear?

Los modelos de dominio en el desarrollo de sotware

¿Para qué modelos de dominio?

- El modelo de dominio **dicta el diseño** de la parte central de nuestro software
  - La parte más estable, la que menos cambia entre versiones, la que implementa la funcionalidad fundamental para el usuario
- El código es más fácil de mantener y entender porque podemos interpretarlo en base a un modelo
  - Cualquier prototipo del software proporcionará realimentación sobre el modelo
- El modelo de dominio es la espina dorsal de un **lenguaje compartido** entre desarrolladores, expertos en el dominio y usuarios
  - Lo que facilita una comunicación efectiva
- El modelo de dominio es **conocimiento destilado**
  - Es la forma en que el equipo de desarrollo ha decidido razonar sobre el dominio del problema

El papel central del dominio

- Lo fundamental de una aplicación software es que resuelva problemas relacionados con su dominio
  - El resto de características respaldan este propósito
- Si el dominio es complejo, los desarrolladores tendrían que pasar tiempo analizándolo junto a expertos y usuarios
  - Esto no siempre es una prioridad en los proyectos
  - Muchos desarrolladores prefieren los problemas técnicos, normalmente más fáciles de delimitar
  - Tener buen conocimiento de un dominio es una ventaja competitiva fundamental para trabajar en ese campo
- Un buen modelo de dominio es la base para una aplicación que quiera mantenerse durante muchos tiempo
  - La tecnología (GUI, bases de datos, conectividad en red) cambia mucho más deprisa que la mayor parte de los dominios de problema

CNBC

MENU MARKETS BUSINESS INVESTING TECH POLITICS

CYBERSECURITY | ENTERPRISE | INTERNET | MEDIA | MOBILE | SOCIAL MEDIA

Doctors are asking Silicon Valley engineers to spend more time in the hospital before building apps

- Richard Zane, an emergency room physician, developed a program so that engineers can understand the clinician's workflow before they build their products
- RxRevu is one start-up that shadows Zane on the job.
- In the Bay Area, it's become common for doctors to invite technologists from Google and elsewhere to follow them on the job.

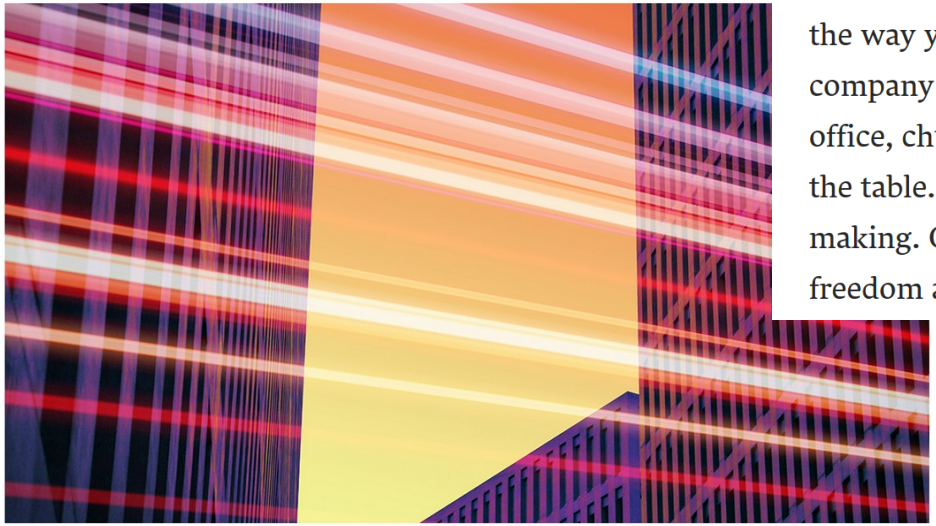
Christina Farr | @chrissyfarr

Published 9:38 AM ET Fri, 28 Dec 2018 | Updated 12:43 PM ET Fri, 28 Dec 2018

In the Digital Economy, Your Software Is Your Competitive Advantage

by Jeff Lawson

January 18, 2021



Arthur Debat/Getty Images

Summary. Many companies respond to digital competition by embracing methodologies like agile, building "innovation centers," acquiring startups, or outsourcing app development to consulting firms. But the true disruptors know that in the digital economy, whoever builds the best software wins. Companies that want to compete need to empower their developers and

That means organizations must build their own software development teams and empower developers to be creative problem-solvers. Companies can start by reskilling existing tech staff. These people are among your most valuable employees, but often are an untapped resource.

But companies also must recruit and retain top-tier software engineers. How does a non-tech company lure great developers? You must change the way you view developers. The best engineers won't work for a company that treats them like "code monkeys" — stuck in some back office, churning out code on command. Top developers want a seat at the table. Involve engineers in strategic problem solving and decision-making. Give them a voice in shaping the future of the company, and the freedom and autonomy to be creative.

Obtener conocimiento de un dominio

¿Cómo se modeliza un dominio?

- 1. Te sientas con los expertos del dominio
- 2. Les haces algunas preguntas: qué necesitan del software, lo básico de su trabajo...
  - Se puede hacer en paralelo a la captura de requisitos o después, apoyándote en esos requisitos
- 3. Haces algún diagrama informal que te permita reflejar lo que te cuentan los expertos y discutirlo con ellos
  - Para empezar suelen funcionar los de objetos, alguna interacción, autómatas...
  - Luego se puede ir pasando a diagramas de clases, diagramas de secuencia...
- 4. Discutes el diagrama con los expertos y les haces más preguntas
- 5. Repites 3-4 y refinas los diagramas, que poco a poco se van convirtiendo en un modelo básico del dominio
- 6. Implementas un prototipo a partir de tu modelo básico
  - Te permite comprobar si lo entiendes, si te faltan cosas fundamentales por capturar, te da algo que enseñar a los expertos para confirmar si es lo que tiene que salir y obtener *feedback*...
- 7. Repites los pasos 3-6, evolucionando cada vez tanto el modelo como los prototipos, y conforme tienes cosas más claras vas comenzando el desarrollo real

Extraer conocimiento del dominio

- Relaciona modelo e implementación lo antes posible
  - Un prototipo muy simple puede servir
- Desarrolla un lenguaje compartido (entre desarrolladores y expertas en el dominio) basado en el modelo
  - Llamad todos siempre a los conceptos principales de la misma forma, no uséis sinónimos
- Evita un modelo que sea solo un “esquema de datos”
  - El comportamiento y las reglas son tan importantes o más
- Quitar cosas del modelo conforme lo refinas es tan importante como añadir cosas nuevas
- Cambia, refina, discute, prueba... **iterativamente**
- El trabajo es conjunto entre equipo de desarrolladores y equipo de expertos
- También se obtiene conocimiento de los usuarios de sistemas anteriores y de la experiencia previa del equipo de desarrollo con ese dominio
  - Parte puede venir en documentos (p.ej. análisis de requisitos, manuales de aplicaciones existentes...) pero fundamentalmente hay que hablar

Extraer conocimiento del dominio

- Con metodologías tradicionales (en cascada) las expertas del dominio hablaban con las analistas, que lo digerían y se lo pasaban a los diseñadores/programadores
  - Esto falla, en parte, por falta de *feedback*
    - Los analistas creaban modelos basados solo en los expertos del dominio, sin la posibilidad de experimentar con prototipos de software
- Una metodología iterativa tampoco es la solución si no hay interés en entender el dominio del problema y solo se busca implementar unos requisitos concretos para una aplicación concreta
  - Si no hay un modelo de dominio, la siguiente versión de la aplicación será, en cierta forma, un problema totalmente nuevo
  - Un buen modelo de dominio actúa como hilo conductor entre los requisitos, el diseño y la implementación

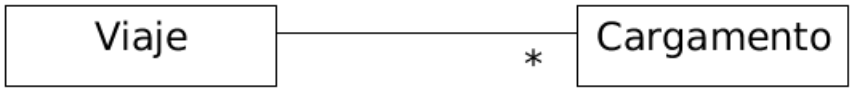
Aprendizaje continuo

- En un proyecto el conocimiento está fragmentado entre personas y documentos, y mezclado con otra información
- El conocimiento se va perdiendo
  - La gente se marcha, los equipos se reorganizan, partes del sistema se subcontratan y el conocimiento de esas partes se queda fuera...
  - ...y el código y los documentos posiblemente no reflejan todo el conocimiento que tanto costó obtener
- Un equipo productivo y consciente de este problema practicará el **aprendizaje continuo**, que incluye un aprendizaje en profundidad del dominio en el que trabajan por parte de todo el equipo

Diseño rico en conocimiento

- Un buen modelo de dominio no es solo un esquema de datos
  - Entidades+relaciones+atributos **no es suficiente**
  - Debe incluir comportamiento (en forma de actividades, procesos, interacciones...) y reglas
    - Muchas veces serán más complicados de explicar para los expertos que los conceptos estáticos
    - Y también más difíciles de expresar y discutir para los diseñadores

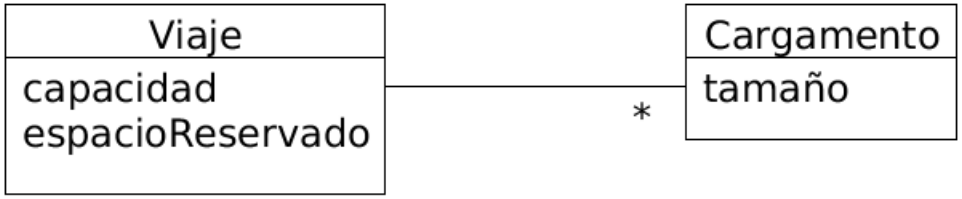
Ejemplo: extraer un concepto oculto



- Un modelo de dominio muy simple para una aplicación que permita reservar espacio para cargamentos en un barco: asociar cada cargamento con un viaje, y grabar y seguir esta relación

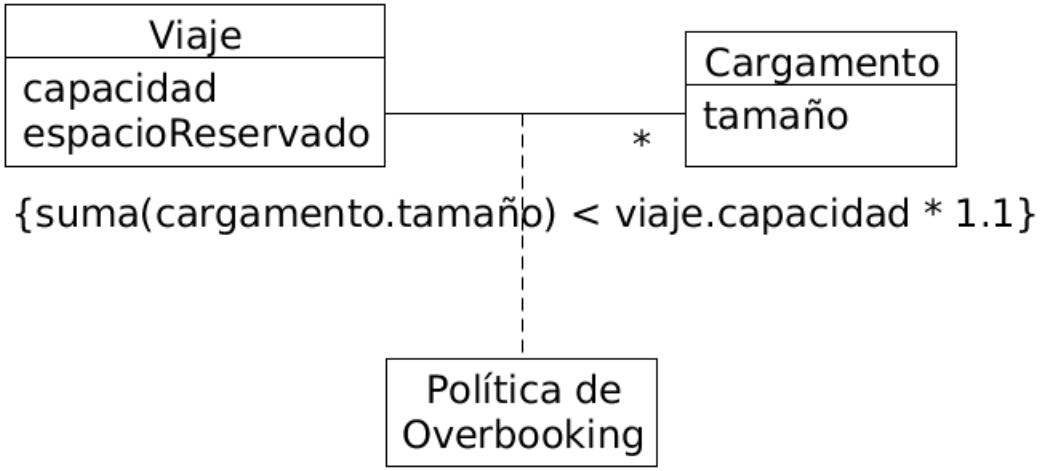
```
public int hacerReserva(Cargamento cargamento, Viaje viaje) {
  int confirmacion = solicitarIdConfirmacion();
  viaje.añadeCargamento(cargamento, confirmacion);
  return confirmacion;
}
```

- Es habitual que se permita hacer *overbooking* de un barco (aceptar más carga de la máxima que cabe). Supongamos que se expresa con un porcentaje y que en el documento de requisitos tenemos:
  - Permitir un 10% de *overbooking*




```
public int hacerReserva(Cargamento cargamento, Viaje viaje) {
  double maxReserva = viaje.capacidad() * 1.1; // 10% OVERBOOKING
  if (viaje.espacioReservado()+cargamento.tamaño()) > maxReserva)
    return -1;
  int confirmacion = solicitarIdConfirmacion();
  viaje.añadeCargamento(cargamento, confirmacion);
  return confirmacion;
}
```

- Ahora tenemos una regla de negocios **oculta** en el código de un método de la aplicación
- Deberíamos hacerla más visible (y con más razón si fuera más compleja)
  - Para poder compartirla con los expertos del dominio sin tener que enseñarles el código (que no van a entender)
  - Pero también para que un perfil técnico (desarrollador) tenga más claro que ese fragmento de código está ahí por una política concreta (permitir un 10% de *overbooking*)



```
public int hacerReserva(Cargamento cargamento, Viaje viaje) {
  if (!politicaOverbooking.sePermite(cargamento, viaje))
    return -1;
  int confirmacion = solicitarIdConfirmacion();
  viaje.añadeCargamento(cargamento, confirmacion);
  return confirmacion;
}
```

Y en la nueva clase `Política de Overbooking`:

```
public boolean sePermite(Cargamento cargamento, Viaje viaje) {
  return (cargamento.tamaño() + viaje.espacioReservado()) <=
(viaje.capacidad() * 1.1);
}
```

- La política de *overbooking* es ahora explícita y la implementación está separada (y por tanto podría usarse fácilmente en distintas partes de la aplicación)
- No tendremos que ir a este nivel de detalle para cada elemento del dominio, pero si para las cosas que sean más importantes o que estén más sujetas a cambios

Lenguaje y comunicación

Lenguaje compartido

- Cada dominio de problema tiene su propia jerga y los/as desarrolladores/as tenemos la nuestra
  - Un proyecto necesita un lenguaje compartido que sea mejor que el mínimo denominador común entre jergas
- El modelo del dominio puede ser el núcleo de un lenguaje común para un proyecto software
  - Por ejemplo, los conceptos principales de ese lenguaje corresponderán con clases del modelo del dominio y se llamarán igual
- Las metodologías tradicionales enfatizan los documentos de texto y los diagramas detallados para la especificación y el control del proyecto
- Las metodologías ágiles enfatizan la conversación, los diagramas informales y el código fuente para la comunicación
- Todos estos medios pueden ser valiosos en ocasiones diferentes
  - Pero para que cualquiera funcione, **los conceptos que recogen y el lenguaje en que se expresen debe ser compartido** por todo el mundo en el proyecto
    - En conversaciones, documentos, diagramas y código

Lenguaje compartido

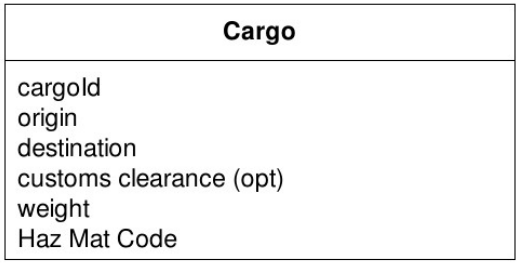
- Un modelo puede verse como un lenguaje
  - Los nombres de clases y operaciones principales constituyen el vocabulario básico
  - El lenguaje proporciona los medios para discutir sobre las reglas que se han hecho explícitas en el modelo
- Este lenguaje, basado en el modelo, debería ser la herramienta principal de comunicación en el equipo de desarrollo para describir los artefactos del sistema, sus tareas y su funcionalidad
  - Y para que los/as expertos/as se comuniquen con el equipo de desarrollo, e incluso entre ellos cuando hablen de requisitos o planes para el proyecto
- El modelo puede no ser lo bastante bueno para cumplir con todos estos roles
  - Pero el compromiso a usarlo ayudará a ver sus debilidades y a tener que buscar soluciones para ellas, que luego se trasladan a los diagramas del modelo y al código

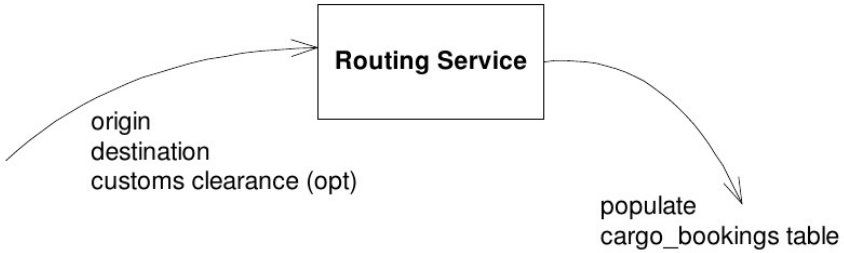
Lenguaje compartido

- Usad el modelo como base del lenguaje compartido
- Usad el mismo lenguaje en diagramas, en documentos, en las discusiones, las conversaciones y en el código fuente
- Si hay dificultades, probad expresiones alternativas que reflejen modelos alternativos
  - Luego refactorizad el código cambiando nombres etc. para ser conformes al nuevo modelo
- Las expertas en el dominio señalarán cosas que les resulten extrañas, forzadas o inadecuadas
- El equipo de desarrollo vigilará que no haya ambigüedades o inconsistencias

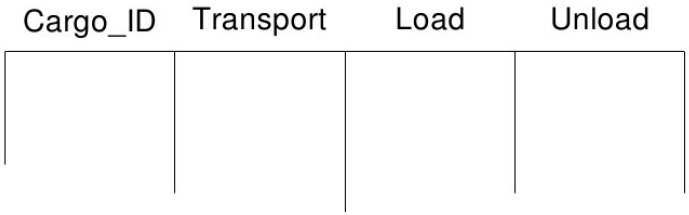
¿Quién habla mejor lenguaje compartido?

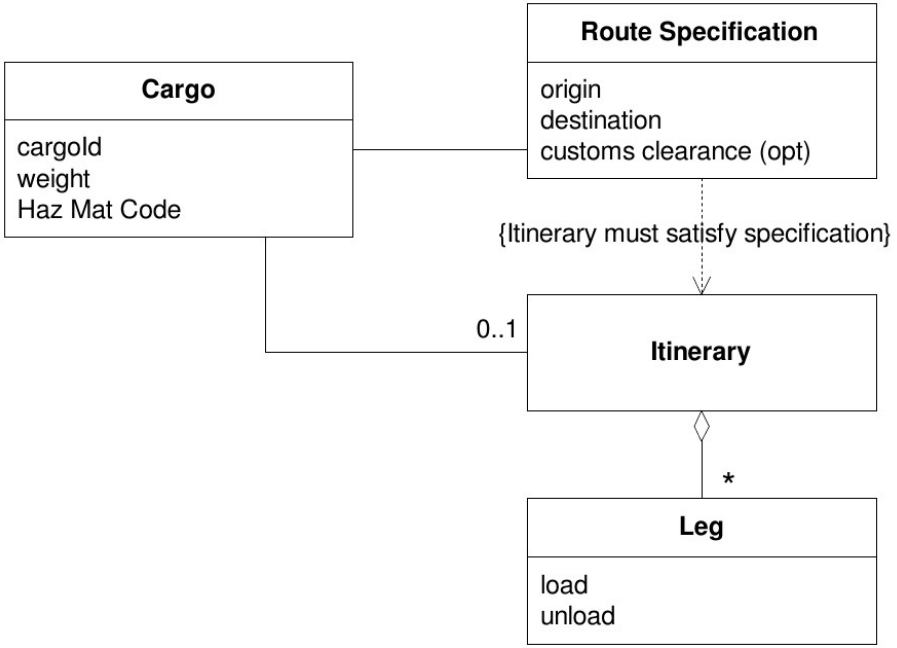
- Experto: cuando cambiamos el punto de control aduanero, tenemos que rehacer toda la ruta
- Desarrollador 1: De acuerdo. Borraremos todas las filas de la **tabla de entregas** que tengan los **identificadores** de los **cargamentos** actuales, calcularemos nueva **ruta** con el **servicio de cálculo de rutas** y repoblaremos la **tabla de entregas**
- Desarrolladora 2: De acuerdo. Cuando haya algún cambio en **la especificación de la ruta** para los **cargamentos** actuales, borraremos el **itinerario** anterior y calcularemos nueva **ruta** con el **servicio de cálculo de rutas** basándonos en la **especificación actualizada**

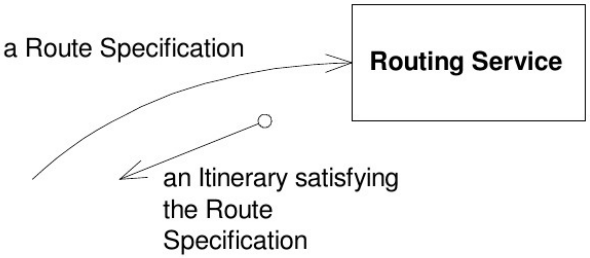




Database table: cargo_bookings



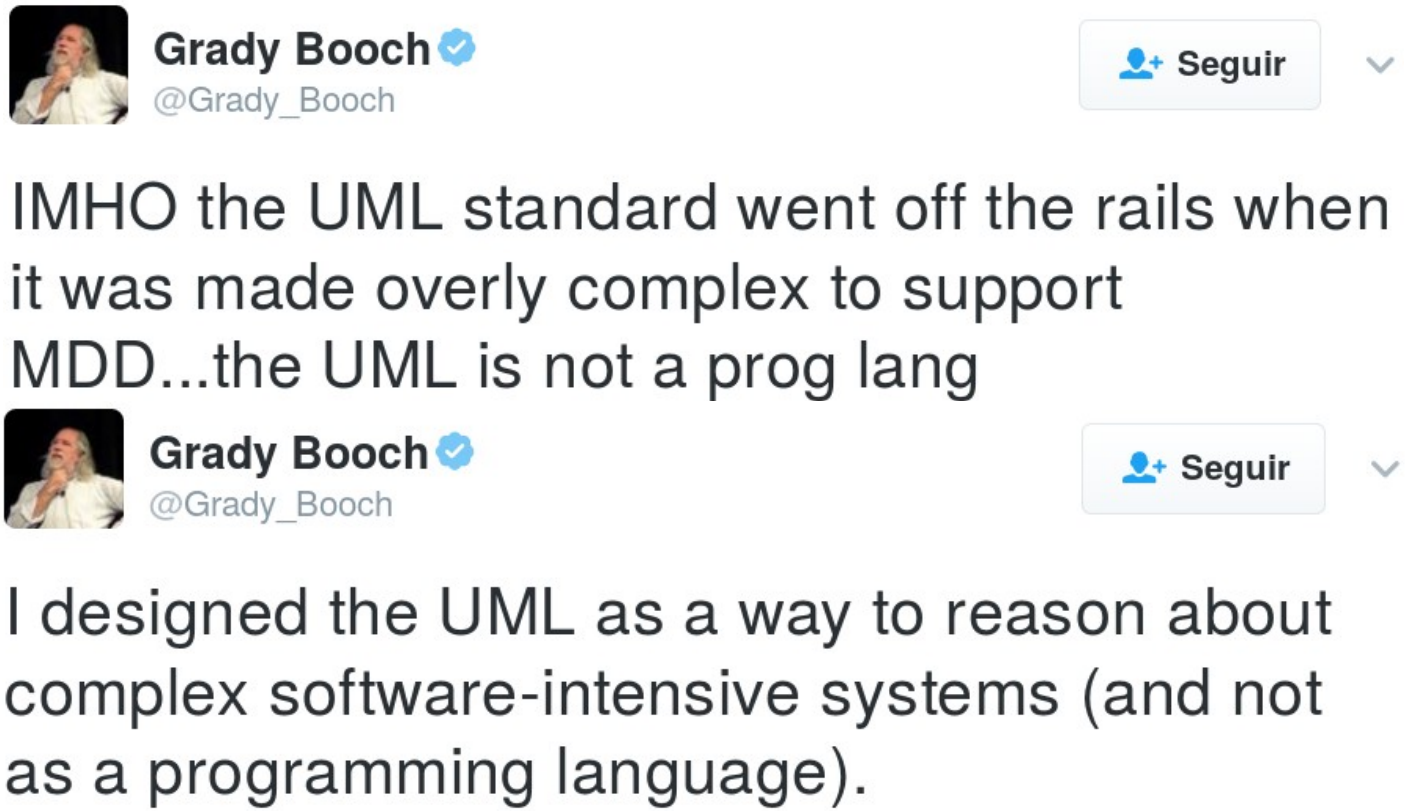




Documentos y diagramas

Diagramas

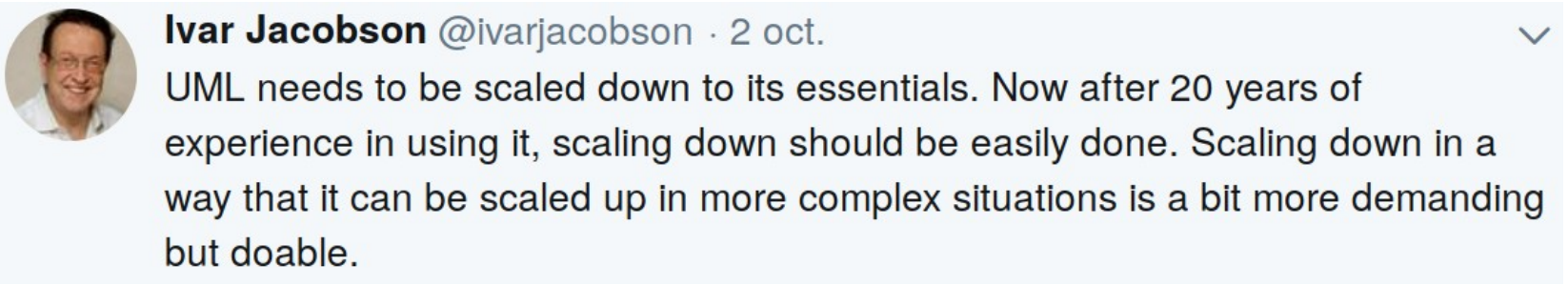
- Los diagramas son una forma natural de representar conceptos y sus relaciones e interacciones
  - Algunos de los diagramas del UML (no necesariamente muy formales o sintácticamente estrictos) son buena opción
- Un diagrama es una buena forma de anclar una discusión
  - Probad a discutir sobre cualquier tema mínimamente complicado con una docena de personas de países diversos con distintos niveles de inglés sin dibujar nada
- Los problemas aparecen al intentar representar **todo** el modelo en UML
  - A partir de cierto nivel de detalle, el UML (como todas las notaciones gráficas) pasa rápidamente de ser una ayuda a ser un estorbo



En mi humilde opinión, el estándar UML descarriló cuando lo hicieron demasiado complejo para soportar el diseño dirigido por modelos... el UML no es un lenguaje de programación.

Yo diseñé el UML como una forma de razonar sobre sistemas complejos intensivos en software (y no como un lenguaje de programación).

(Grady Booch es uno de los tres padres del UML)



UML necesita ser reducido a lo esencial. Ahora, tras 20 años de experiencia con su uso, reducirlo debería ser fácil de hacer. Reducirlo de manera que pueda aumentarse en situaciones más complejas es un poco más complicado, pero factible.

Exacto. En 2010 escribí un artículo con Steve Cook titulado “El camino adelante para UML”, sugiriendo un UML Esencial, un pequeño pero bien elegido subconjunto de UML. Menos del 20% de UML cubriría más del 80% de las necesidades del modelizado. Una pena, todavía no está implementado.

(Ivar Jacobson es uno de los tres padres del UML)

Diagramas

- Tampoco el UML es la solución perfecta para expresar el comportamiento del sistema
  - Los diagramas de interacción y secuencia son útiles para representar algunos comportamientos especialmente importantes o complicados, pero tratar de representar la mayoría de las interacciones así solo es factible en problemas de juguete
- Ni con las restricciones o aserciones
  - En UML las puedes representar como texto o como fórmulas en OCL, pero esto no es muy visual (hablamos de diagramas)
- Tampoco el UML ayuda nada a expresar las responsabilidades de los objetos
  - El lenguaje natural sin embargo, funciona bien para esto

Documentos de diseño

- Una vez se crea un documento, a menudo queda desconectado del proyecto
  - P.ej. porque la evolución del código lo acaba por dejar atrás
- El código no miente y su comportamiento no es ambiguo
  - Pero puede abrumar con detalles al lector y ser difícil de comprender
  - Además, los/as programadores/as no son los únicos que tienen que entender el modelo
- La comunicación verbal ayuda a entender el código, pero cualquier grupo un poco grande necesitará la estabilidad y "compartibilidad" de algunos documentos escritos
  - Cuanto más grande y distribuido está un equipo, más importante es dejar más cosas por escrito
- Los documentos de diseño tienen que aportar significado, sacar a la luz estructuras grandes y centrarse en los elementos principales del programa
- Deben explicar las razones que han llevado al sistema a ser así, las alternativas que se barajaron y las restricciones que se aplicaban
- Deben estar vivos y escritos en el lenguaje compartido
  - Un documento que no se actualiza es porque no se usa, y normalmente eso es porque el equipo lo encuentra inútil
  - Los documentos muy largos y/o complejos tienden a ser difíciles de actualizar y por tanto acaban por ser desechados

En resumen

- Usar el UML casi como lenguaje de programación, incluyendo todos los detalles necesarios sobre el sistema, no es una buena opción
  - Si aún así te empeñaras en usarlo entonces tendrías que buscar una alternativa para representar el modelo del sistema sin tanto detalle
- **Los diagramas y documentos de diseño son un medio de comunicación y explicación y facilitan razonar**
  - Si son muy detallados, pierden esta utilidad: sobrecargan al lector con detalles y se hacen difíciles de entender
- **Los detalles del diseño están en el código**
  - Una implementación bien escrita debe revelar el modelo subyacente
- **El modelo no es el diagrama**
  - El diagrama comunica y explica el modelo y el código contiene los detalles del mismo

Enlazar modelo e implementación

Modelo e implementación

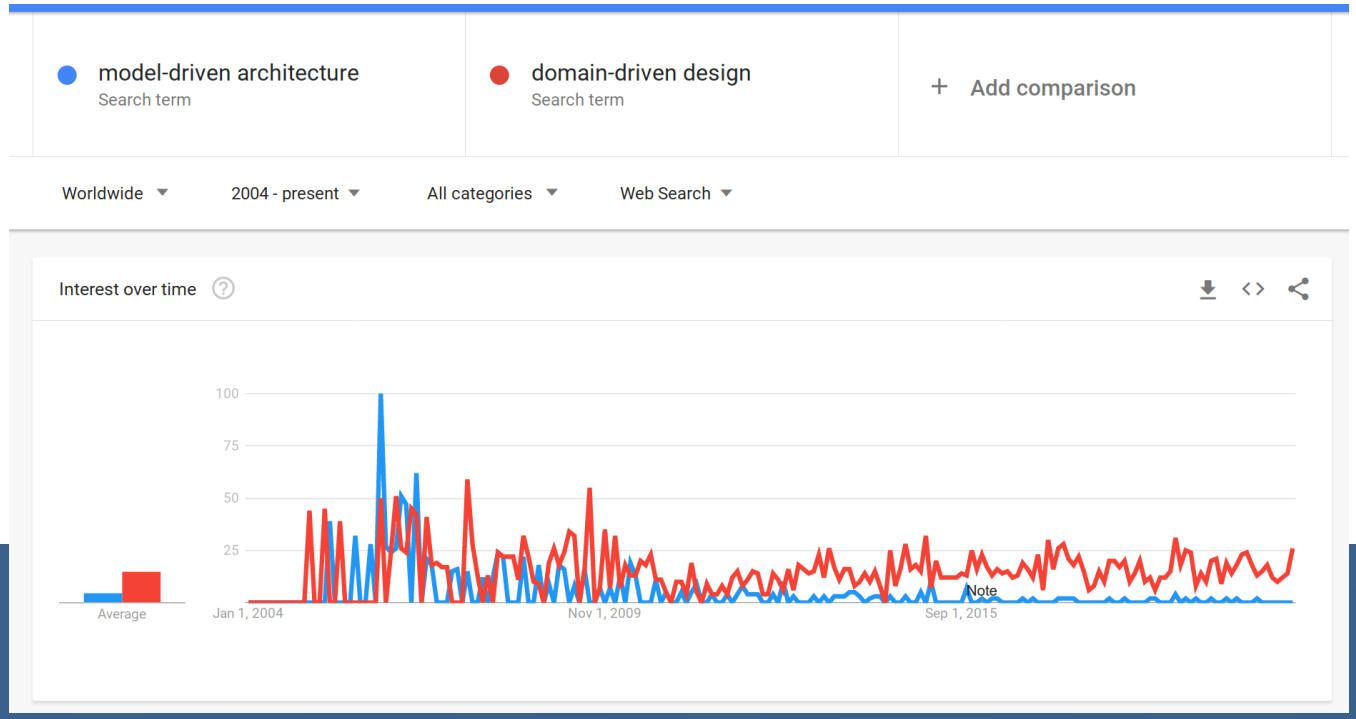
- La base del diseño dirigido por el dominio es que **el mismo modelo debería ser la base de la implementación, el diseño, el análisis y las comunicaciones en el equipo** (incluyendo también a los/as expertos/as en el dominio)
  - Y también una forma de explicar ese dominio
- El diseño dirigido por el dominio busca un modelo que no sea solo una forma de analizar el problema, sino que sea la auténtica base para el diseño
  - Esto tiene implicaciones para el código, pero también requiere una aproximación diferente al modelizado

Modelo e implementación

- Las metodologías más tradicionales crean un “modelo de análisis” para comprender (analizar) el dominio del problema
  - Este modelo no considera en absoluto aspectos de diseño o implementación del software
- Después se crean modelos de diseño que tienen cierta correspondencia con el modelo de análisis, pero que sí tienen en cuenta algunos aspectos de implementación
  - Típicamente ambos modelos acaban por diferir bastante, porque abordan problemas diferentes
  - En el paso del modelo de análisis al de diseño se pierde conocimiento sobre el dominio del problema
- Mantener la relación entre ambos modelos es costoso, así que esta se va perdiendo
- Cualquier cosa que surja en el diseño/implementación de interés sobre el dominio nunca llegará al modelo de análisis
  - Que en estas metodologías suele considerarse casi inmutable a pesar de haberse creado cuando menos se sabía del problema
- Como resultado, el modelo de análisis se abandona poco tiempo después de empezar a escribir código, y hay que ir reinventándolo (mal) sobre la marcha, y con menos acceso a los expertos del dominio

Model-Driven Architecture

- Para tratar de arreglar algunos problemas de esta visión tradicional (pero sin cambiarla en su esencia) el OMG propuso en 2001 la arquitectura dirigida por modelos (*Model-Driven Architecture*)
  - Basada en UML y en complejas herramientas de tipo CASE
- Casi 20 años después, esta práctica no es de uso común y el interés por la misma así lo refleja



Grady Booch ✓
@Grady_Booch

Seguir

IMHO the UML standard went off the rails when it was made overly complex to support MDD...the UML is not a prog lang

Grady Booch ✓
@Grady_Booch

Seguir

I designed the UML as a way to reason about complex software-intensive systems (and not as a programming language).

Grady Booch, padre del UML, insiste ;-)

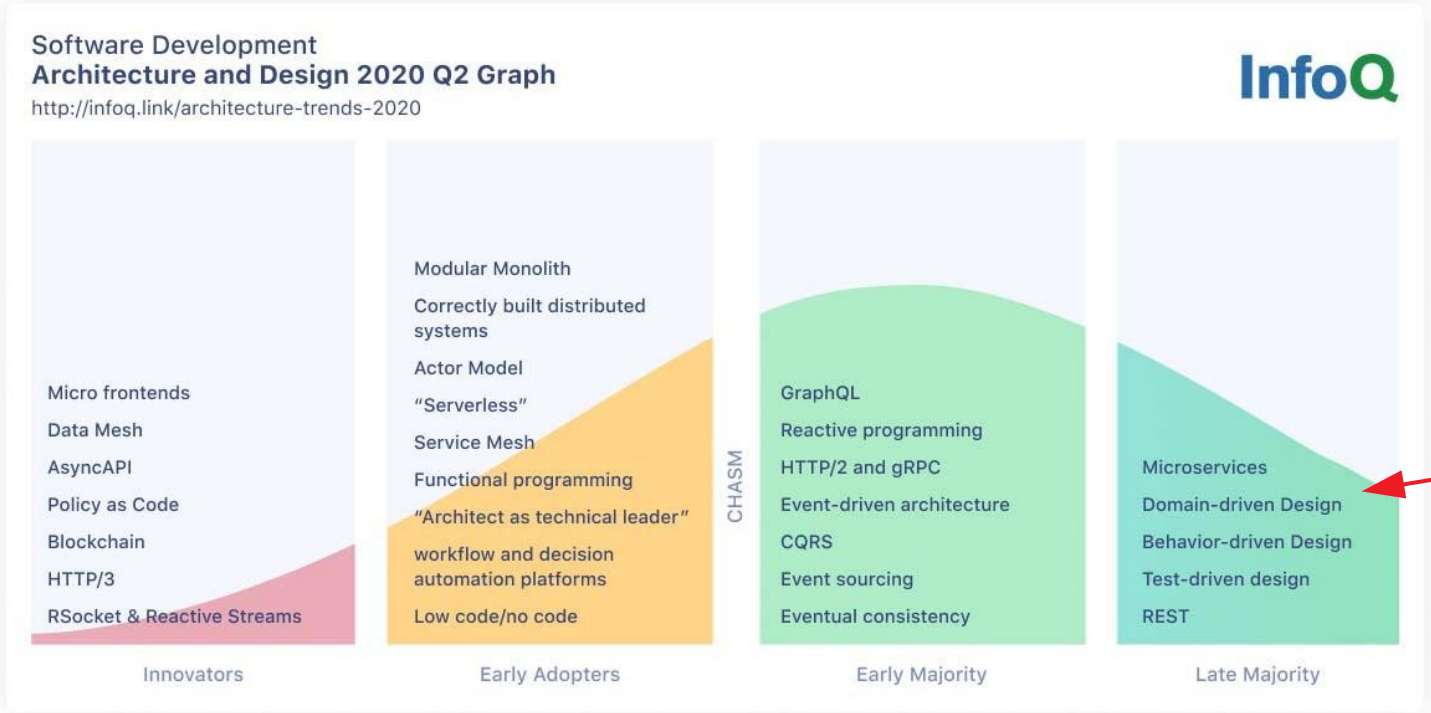
Ivar Jacobson @ivarjacobson · 2 oct.

UML needs to be scaled down to its essentials. Now after 20 years of experience in using it, scaling down should be easily done. Scaling down in a way that it can be scaled up in more complex situations is a bit more demanding but doable.

Ivar Jacobson @ivarjacobson · 19 sept.

Exactly, in 2010 I wrote a paper with Steve Cook titled "The Road ahead for UML" ubm.io/2Nn3NFt, suggesting a UML Essentials, a small but well selected subset of UML. Less than 20% of UML would cover more than 80% of the need of modelling. Too bad, not yet implemented.

Ivar Jacobson, padre del UML, insiste ;-)



La tendencia (2020) es que el DDD se ha convertido en una práctica mayoritaria en el diseño y arquitectura de software

Modelo e implementación

- Si el diseño no se relaciona claramente con el modelo del dominio (antes conocido como modelo de análisis), ese modelo tiene poco valor, y la corrección del software estará bajo sospecha
- Por otra parte, los mapeos complejos entre modelos y elementos del diseño son difíciles de expresar y, sobre todo, difíciles de mantener conforme evoluciona el diseño
- Una división entre el análisis y el diseño que impida que esas dos actividades se alimenten mutuamente es una buena forma de conseguir un mal software

Modelo e implementación

- Visión tradicional
  - El análisis debe capturar los conceptos del dominio de manera lo más expresiva y completa posible
  - El diseño debe especificar los componentes que se desarrollarán/integrarán con las tecnologías de las que dispongamos para implementar un sistema eficiente y correcto
- **El diseño dirigido por el dominio descarta la necesidad de tener modelos de análisis y diseño distintos**, y propone un único modelo que cumpla ambos propósitos
  - Crear este modelo será más complejo, pero nos permitirá crear un software más coherente, mejor documentado y más fácil de mantener
  - Expresaremos este modelo único tanto con diversos diagramas UML como con código

Paradigmas de modelizado y herramientas de soporte

- Para que esto funcione, es necesaria una correspondencia fiel entre modelo e implementación
  - El lenguaje de programación tiene que soportar conceptos directamente análogos a los que usemos en las otras expresiones (p.ej. diagramas) del modelo
- La orientación a objetos cumple este requisito
  - El avance crucial de la orientación a objetos es que permite utilizar los mismos conceptos para analizar un problema, y para implementar una solución al mismo
- La orientación a objetos es el paradigma de análisis/diseño/implementación mejor establecido y se adapta bien a dominios de problema muy diversos
  - Aunque si tu dominio de problema se modeliza fielmente con reglas lógicas, quizás tu implementación natural sea PROLOG

Modelizadores e implementadores

- En la visión tradicional/ingenua, el desarrollo de software se consideraba como un tipo de fabricación
  - Se suponía que ingenieros cualificados diseñaban y trabajadores menos cualificados producían
- Esta metáfora es catastróficamente incorrecta: **el desarrollo de software es diseño, no es fabricación, y requiere una alta cualificación**
- Los responsables de modelizar el dominio de problema no pueden estar separados totalmente del código
  - Eso les impediría conocer de primera mano los problemas que surgen ahí, y ese *feedback* es fundamental para crear un buen modelo del dominio
- Los que escriben el código deben sentirse corresponsables del modelo y entenderlo bien, o el software acabará creándose sin contar con este modelo
  - Perderíamos las ventajas del diseño dirigido por el dominio
- Todos los desarrolladores tienen que involucrarse (aunque sea a niveles distintos) en las discusiones sobre el modelo y el contacto con los expertos del dominio

Bibliografía

- Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*
  - Capítulos 1,2,3