La importancia de la estructura en el software

Jordan Colgan
www.codurance.com

¿Es suficiente escribir código "bueno" o "limpio" por sí solo? ¿Qué hace que el código sea bueno? Puedes valorar la capacidad de leer el código y entenderlo sin tener que compilarlo y ejecutarlo. Otros pueden valorar eso, y también que el código demuestre su corrección a través de pruebas.

En la industria del software, se repite tediosamente que los programadores deben valorar las disciplinas y estar orgullosos de que su código sea autoexplicativo y se demuestre a sí mismo [1]. Estos no son descubrimientos o revelaciones recientes. Algunos de los primeros escritos sobre software describen enfoques para demostrar y luego construir programas informáticos, predecesores tempranos de la práctica que hoy conocemos como desarrollo orientado a pruebas (test-driven development).

Durante la Segunda Guerra Mundial, John Mauchly y J. Presper Eckert se propusieron desarrollar la primera computadora programable de Estados Unidos, Electronic Numerical Integrator and Computer (ENIAC), con el objetivo inicial de automatizar cálculos balísticos de artillería [2]. Estas tablas balísticas originalmente eran producidas por humanos, casi 200 mujeres que habían calculado las tablas con calculadoras mecánicas antes del desarrollo de ENIAC. Algunas de estas mujeres fueron seleccionadas para programar la máquina que las reemplazaría a ellas y a sus colegas. Estas empleadas, convertidas en programadoras, utilizaron su práctica de cálculo manual de las tablas para impulsar el desarrollo de ENIAC y también para reducir los errores debido a fallos en sus componentes físicos [3].

Las ideas sobre verificar manualmente la salida esperada de los programas informáticos continuaron, con una mención notable en "Digital Computer Programming" [4], de 1957.

> El primer enfoque para abordar el problema de la verificación se puede hacer antes de comenzar a codificar. Para asegurarse completamente de la precisión de las respuestas, es necesario tener un caso de prueba calculado a mano con el cual comparar las respuestas que más tarde calculará la máquina. Esto significa que las máquinas de programas almacenados nunca se utilizan para un problema verdaderamente de un solo disparo. Siempre debe haber un elemento de iteración para que sea rentable. Los cálculos manuales se pueden realizar en cualquier momento durante la programación. Sin embargo, con frecuencia, las computadoras son operadas por expertos en cálculos para preparar los problemas como un servicio para ingenieros o científicos. En estos casos, es muy deseable que el "cliente" prepare el caso de prueba, en gran parte porque los errores lógicos y los malentendidos entre el programador y el cliente pueden señalarse mediante dicho procedimiento. Si el cliente debe preparar la solución de prueba, es mejor que comience con bastante antelación al momento de la verificación real, ya que para cualquier problema de cierto tamaño, llevará varios días o semanas calcular a mano la prueba.

Esto suena como los inicios del Acceptance Test-Driven Development (ATDD[1]), donde el "cliente" o alguien que representa al cliente (analista de negocios, gerente de productos, etc.) trabaja junto con un ingeniero de aseguramiento de la calidad para producir un conjunto de especificaciones ejecutables de cómo esperan que funcione el programa. Antes de que comience el desarrollo de cualquier característica [5]. Estas especificaciones ejecutables son luego implementadas por los desarrolladores a medida que construyen la característica y utilizan su estado aprobado para marcarla como "terminada". ATDD es, por supuesto, una rama de TDD, una práctica utilizada por los desarrolladores para crear sus propias pruebas, independientemente de las pruebas de aceptación proporcionadas por el negocio y la calidad.

Otra mención notable del desarrollo basado en pruebas proviene del Proyecto Mercury de la NASA en la década de 1960 [6]. Ingenieros de pruebas independientes escribieron procedimientos de prueba, a partir de los requisitos de hardware proporcionados, antes de que los desarrolladores escribieran el software para integrarlo con el hardware. Esto les permitió acortar el tiempo de desarrollo total y aumentar la paralelización de la garantía de calidad. En comparación con otros proyectos de software de la época, que comenzaron a verse afectados por la inminente "Crisis del Software".

La disciplina exacta del Test-Driven Development comenzó a formarse en 1993 [7], como "programación basada en pruebas", que luego se convirtió en el Test-Driven Development más refinado que conocemos hoy. A Kent Beck se le atribuye la invención del Test-Driven Development, pero él se refiere a haber redescubierto la disciplina. Podemos ver por qué pensaría de esta manera, con la valoración de la comprobación antes de la implementación documentada a lo largo de la historia del desarrollo de software.

El Test-Driven Development es solo una parte de la metodología de programación extrema (XP). Concebida con la intención de mejorar la calidad del software y responder a las crecientes necesidades de agilidad empresarial [8]. A pesar de que las prácticas de XP son una herramienta adoptada por muchos programadores modernos, la mayoría aún no lo acepta como propio. Esta actitud contra XP parece casi antagonista, considerando que se dice que estamos en una "Crisis del Software".

El término "Crisis del Software" surgió durante la primera Conferencia de Ingeniería de Software de la OTAN en 1968 en Garmisch, Alemania [9]. Edsger Dijkstra también hizo referencia al mismo problema en su conferencia de entrega del Premio Turing de 1972 [10]. Los principales aspectos relacionados con el software son la baja calidad, la ineficiencia y la falta de cumplimiento de los requisitos deseados. En términos de la entrega general del proyecto, los proyectos de software generalmente presentaban problemas de sobrecosto, demora y baja retención de desarrolladores debido a la calidad del software.

Aunque la crisis del software se identificó hace mucho tiempo, el software construido hoy todavía tiene los mismos problemas. Parece una argumento difícil seguir llamándolo una crisis, considerando que ha estado ocurriendo durante tanto tiempo. ¿Por qué el software

---

[1] ATDD también es Bheaviour Driven Design (BDD). Aunque incorrectamente, a menudo se confunde BDD con el uso de la sintaxis Gherkin (Given,When,Then).

sigue fallando una y otra vez? Para responder a esa pregunta, primero debemos comprender los valores del software.

Los dos valores del software

¿Qué es exactamente el software? "Britannica" ofrece la siguiente definición [11]:

> Software, instrucciones que dicen a una computadora qué hacer. El software comprende el conjunto completo de programas, procedimientos y rutinas asociadas con el funcionamiento de un sistema informático. El término se acuñó para diferenciar estas instrucciones del hardware, es decir, los componentes físicos de un sistema informático. Un conjunto de instrucciones que dirige el hardware de una computadora para realizar una tarea se llama programa o programa de software.

El software fue inventado para cambiar el comportamiento de las máquinas, el hardware. La razón de esto se encuentra en la palabra "hardware", que significa difícil de cambiar. Requiere un esfuerzo y recursos enormes diseñar y construir hardware, por lo tanto, no podemos perder más tiempo intentando cambiarlo por diversas razones. El software prometía la capacidad de cambiar fácilmente la salida para el hardware en el que se ejecuta. A partir de esto, podemos concluir que un valor del software es su "comportamiento".

Entonces, ¿cuál sería el otro valor del software? Eso sería cómo compartimos las instrucciones que el software realiza, es decir, la arquitectura. La arquitectura en el mundo del software es difícil de definir. Con mucha frecuencia se asocia simplemente con la descripción general de alto nivel de cómo interactúan los componentes entre sí, pero es mucho más que eso [12]. Es todo, desde las decisiones de alto nivel hasta los detalles de bajo nivel. La arquitectura es la forma en constante crecimiento del sistema. Por lo tanto, el segundo valor del software es su "estructura".

¿Cuál de los dos es más importante? ¿Es el comportamiento, la estructura o son igualmente importantes? Si le preguntaras a los desarrolladores cuál es más importante para ellos, es muy probable que respondan "el comportamiento". Los programadores son contratados para cambiar la forma en que las máquinas se comportan, con el fin de ganar o ahorrar dinero para sus stakeholders. Por lo tanto, muchos programadores asumen que su prioridad predeterminada es el comportamiento, ya que puede parecer que es la totalidad de su trabajo. Es su responsabilidad implementar los requisitos y corregir cualquier error en la funcionalidad del software.

Si le preguntaras a los stakeholders, a los gerentes y a cualquiera del lado "negocio", por supuesto que responderían "el comportamiento". Para ellos, el proceso parece consistir en proporcionar a los desarrolladores especificaciones funcionales y documentación de requisitos, a partir de la cual los desarrolladores de alguna manera hacen que las máquinas hagan lo que se les pidió, eventualmente. Sin sus solicitudes, los programadores no tendrían trabajo que hacer.

Hasta ahora, parece obvio que es más importante que el sistema funcione que tener una estructura sólida. Sin embargo, considere nuevamente que la razón principal por la que se

inventó el software fue para cambiar fácilmente el comportamiento del hardware. El propósito del software es ser maleable, fácil de cambiar. El software que es fácil de cambiar en su estructura puede modificarse fácilmente para que funcione. Es decir, el software que es fácil de cambiar puede mantenerse al día con los requisitos siempre cambiantes del negocio.

El software que funciona, pero sufre de una estructura deficiente que es difícil de cambiar, resulta en una situación no deseada en la que el software tiene dificultades para cumplir con los requisitos. Cuando el negocio desea agregar nuevas funciones o ampliar las existentes, lleva demasiado tiempo modificar el sistema para cumplir con los nuevos requisitos empresariales. Peor aún, el software que no se puede ampliar debido al enredo se considera inútil y pierde su valor para el negocio. El negocio solo ve a los programadores como valiosos debido a su capacidad para hacer que las máquinas hagan lo que se les pide. Cuando los programadores pierden el control del software, también pierden su valor.

Usted, como desarrollador, puede darse cuenta de la importancia de la estructura en este momento, pero ¿cómo convence a su gerente o al negocio de que la estructura es más importante que el comportamiento? Considere el negocio y sus competidores. El software que escribe para el negocio tiene mejores características que sus competidores. Puede implementar estas características más rápidamente, pero solo porque se saltó la calidad del software. Por supuesto, se dice a sí mismo que volverá y lo mejorará. Sin embargo, sus competidores sí se preocupan por la calidad e insisten en mejorar la estructura del software a medida que avanzan, lo que los hace más lentos.

¿Qué sucede cuando se descuida la calidad del software y se promete volver a limpiarlo? Se convence a sí mismo de que algún día lo mejorará, algún día corregirá los parches realizados. Ese día nunca llegará [13]. Incluso si se encuentra con la oportunidad de presentar la idea de corregir los errores, descubrirá que las prioridades han cambiado. Por supuesto, esta es la naturaleza del software y del mundo de los requisitos siempre cambiantes.

Su software que alguna vez tuvo las características que lo convirtieron en la preferencia del mercado ha sido superado por sus competidores. Su estructura les permitió ponerse al día con sus características e incluso los alentó a realizar los cambios necesarios para superarlo.

Los programadores se enfrentan a un dilema. La arquitectura de su sistema es más importante que la urgencia de la funcionalidad que se le requiere. Sin la capacidad de responder fácilmente al cambio, no pueden cumplir y mantener constantemente las necesidades comerciales. El negocio no tiene el conocimiento para entender por qué la arquitectura es importante y por qué se debe dedicar tiempo a cuidarla. Por lo tanto, depende de los programadores aprender y dominar las técnicas requeridas para mantener una estructura sólida.

Acortando la brecha entre código y negocio

La arquitectura de un sistema respalda la intención del sistema. Aunque la arquitectura es más importante que el comportamiento para fines de agilidad, es el comportamiento del

sistema el que influye en la forma del sistema. Una plataforma de comercio electrónico tendrá una arquitectura completamente diferente a la de una red social.

El acto de traducir los requisitos en código se conoce como modelado de dominio [14]. Los modelos de dominio actúan como el enlace entre el lenguaje ubicuo que forma los requisitos en el código y el binario ejecutable que procesa la máquina. Incluso si el dominio es un concepto tangible del mundo real, el modelo sigue siendo nuestra representación artificial en el mundo de los datos. Comprende las abstracciones únicas, el conocimiento y los procesos que la empresa utiliza dentro de la máquina.

A menudo, los modelos de dominio son donde colapsa la estructura del software. Los programadores trabajan en detalles, detalles de bajo nivel. Se preocupan por marcos, interfaces, concurrencia, optimización, por nombrar algunos. Si bien estos son importantes, no son cosas que el negocio entienda o le importe. Cuando los programadores intentan construir modelos de dominio sin principios rectores que aseguren la inversión desde los niveles más bajos[2], se suelen mezclar sus preocupaciones con las del negocio.

El problema no se limita al nivel de los programadores, muchos stakeholders de alto nivel y operaciones pueden influir en decisiones que resultan en un mal modelado de dominio. En organizaciones con grandes bases de código, desde monolitos hasta repositorios dispersos que se conectan entre sí, existirán múltiples modelos. Diferentes modelos tendrán diferentes contextos, pero algunos de estos modelos pueden ser iguales y se trabajarán de manera independiente. La falta de comunicación entre los equipos involucrados resultará en interpretaciones sutiles dentro de los modelos, lo que significa que esos modelos no se pueden compartir. La duplicación es el enemigo del software y es difícil de solucionar.

¿Es esta la naturaleza de los grandes códigos con muchos equipos o hay formas de corregirlo? Los modelos se aplican en contextos, siendo el contexto un área del código o las características en las que trabaja un equipo. La armonía del modelo puede existir y escalarse con operaciones de desarrolladores o empresas. Requiere una intercomunicación, no solo entre los equipos, sino también en el código. Los alcances del modelo se definen como partes delimitadas del sistema de software, que representan los límites donde se aplicará el modelo y su propiedad.

El producto de definir los límites dentro de la organización y los equipos, y especificar los usos dentro del sistema, se conocen como "contextos delimitados". Los contextos delimitados brindan claridad a los equipos y fomentan una comprensión compartida de lo que debe ser coherente y el posible acoplamiento con otros contextos. Se comprende mejor cuándo usar el mismo modelo y cuándo no. También crea conciencia de los compromisos compartidos, lo que ayuda durante el proceso de colaboración en equipo. La sinergia del equipo probablemente fallará si todos no comprenden dónde están los límites de los contextos del modelo.

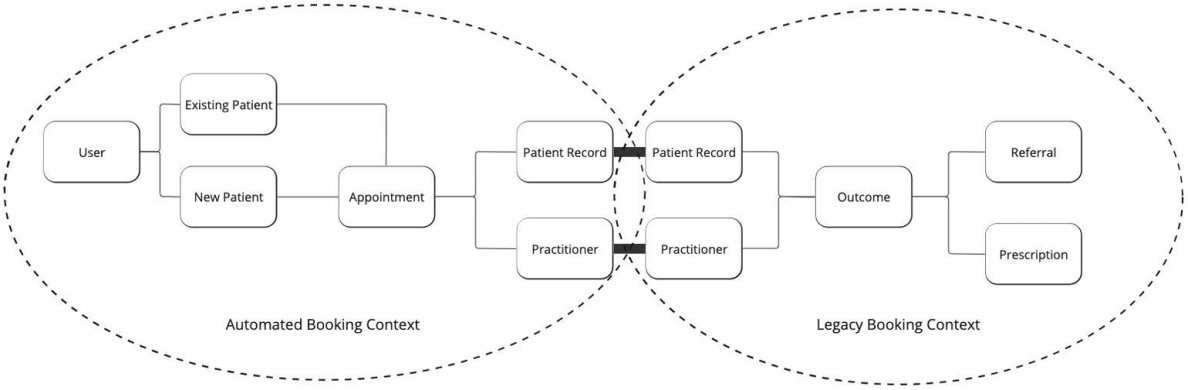
---

[2] También conocido como "Principio de Inversión de Dependencia", se cree erróneamente que se aplica al diseño orientado a objetos, pero representa un principio de diseño de mucho mayor nivel.

¿Cómo se gestionan los contextos delimitados? Hay muchos matices en torno a los diferentes tipos de sistemas de software, pero considere una institución médica que inicia un proyecto interno para la reserva de citas. Este proyecto busca automatizar el proceso de reserva con los médicos, en lugar de que los pacientes tengan que organizarlos manualmente a través de la recepción.

¿Cómo se verán los contextos delimitados para el modelo de esta aplicación? Ya existe un modelo existente con el sistema de reservas heredado utilizado por los recepcionistas. Este sistema es mantenido por un equipo que ha estado influenciando directamente el modelo. Se requiere otro equipo para la nueva aplicación de reserva orientada al usuario. Su modelo esperado tendrá similitudes con el modelo existente, pero hay algunas discrepancias basadas en los casos de uso que han descubierto.

Las reservas de la nueva aplicación deben ser transferidas al sistema de reservas heredado. Se acordó entre los equipos que se formaría un nuevo modelo a partir del modelo heredado. Por lo tanto, el sistema heredado está fuera de los límites. Se requerirán traducciones entre este nuevo modelo y el antiguo. La responsabilidad de estas traducciones recae en el equipo heredado, ya que los contextos atraviesan fuera del límite acordado.



En nuestro ejemplo, los equipos deciden utilizar una base de código monolítica para albergar ambas implementaciones. ¿Significa esto que los contextos delimitados son módulos? Se podría interpretar que los contextos delimitados son lo mismo que los módulos, ya que cada equipo tiene sus propios módulos y submódulos, pero los contextos delimitados y los módulos tienen diferentes motivaciones. Los módulos son paradigmas de lenguaje de programación, y los contextos delimitados deben ser agnósticos de tales paradigmas. El propósito de los módulos es organizar diferentes elementos dentro de un modelo. No siempre comunican claramente la intención de los contextos separados.

Los contextos delimitados son solo parte de los pasos hacia una mejor topología de equipos y cohesión de código. Existen otros patrones de diseño estratégico junto con los contextos delimitados, formando lo que conocemos como Domain-Driven Design (DDD) [15].

La búsqueda constante de calidad

Una buena estructura permite un buen comportamiento, tanto en el sottware como en las personas. Una mala estructura obstruye el buen comportamiento. La flexibilidad para facilitar el cambio debe estar presente desde el principio de la implementación de sistema. Sin mantener limpia la estructura. nos estamos preparando para no sequir nunca el camino de desarrollo sostenible.

¿Que tan bueno es suticientemente bueno? Hay muchas metricas que podemos utilizar para medir la calidad, la cohesión y la agilidad del código. Así como el software tiene la propiedad de cambiar. también la tienen nuestras tácticas y enfoques para su desarrollo y mantenimiento. La mejor actitud que puedes tener es estar siempre en la búsqueda constante de la calidad. Del Juramento del Programador [161. la segunda promesa es:

> El código que produzco siempre será mi mejor trabaio. No permitiré intencionadamente que se acumule codigo detectuoso en comportamiento o estructura.

Mejorar la estructura del software es difícil. Requiere muchos años de conocimientos complejos, que abarcan muchos campos diferentes de la informática. No sólo es esencial un conocimiento teórico profundo. sino que también es crucial una vasta experiencia práctica. Asi como uno debe buscar una estructura de software constante de alta calidad. también debe buscar una alta calidad en sus propias habilidades v conocimientos

Los desarrolladores profesionales deben dar gran importancia a la estructura del código sobre el comportamiento. El software ha sido, y probablemente seguirá siendo, un ámbito rápidamente acelerado en nuestro mundo. Las demandas de los desarrolladores son allas y las exigencias de que los desarrolladores sean rápidos son mavores. No todos los desarrolladores tienen e privilegio de aprender que la única manera de ir rapido es hacer o bien. or eso. es nuestro deber como quienes saben quiar a guienes no saben.

> Los fuertes deben avudar y proteger a los débiles. Entonces. los débiles se volverán fuertes y ellos. a su vez. avudarán y protegerán a los más débiles que ellos. Esa es la lev de la naturaleza
>
> -Tanjiro Kamado

Referencias:

1. Martin, Robert C (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Pearson. ISBN 9780136083238.
2. Light, Jennifer S (1999)."When Computers Were Women". ISSN 0040-165X.
3. Fritz, Barkley W (1996)."The Women of ENIAC". IEEE Anna ls of the History of Computing. doi:10.1109/85.511940.
4. Mc Cracken DD (1957). Digital Computer Programming. John Wiley & Sons.ISBN 9780471582458.
5. Pugh, Ken (2011). Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration. Addison-Wesley. ISBN 978-0321714084.
6. Larman, Cand Basili, VR(2003)."Iterative and incremental developments. A brief history" (PDF). Computer.doi: 10.1109/MC.2003.1204375.
7. Beck, Kent (2002).Test-Driven Development by Example. Vaseem: Addison Wesley. ISBN 9780321146533.
8. Beck, Kent (1999). Extreme Programming Explained. Addison-Wesley Professional. ISBN 9780201616415.
9. Naur, Pand Randell, B (1969).“Software Engineering: Reportona Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968”. Scientific Affairs Division, NATO.
10. Dijkstra, Edsger W (1972).“The humble programmer”. Communications of the ACM 15.10.
11. Britannica,The Editors of Encyclopaedia (2023). "software". Encyclopedia Britannica.
12. Martin, Robert C (2017). Clean Architecture: A Craftsman’s Guide to Software Structureand Design. Pearson. ISBN 9780134494272.
13. Kondo, Marie (2014). The Life-Changing Magic of Tidying Up:The Japanese Art of Decluttering and Organizing. Clarkson Potter/Ten Speed.ISBN9 781607747307.
14. Evans, Eric (2003). Domain-Driven Design:Tackling Complexity in the Heart of Software. Addison-Wesley Professional. ISBN 0321125215.
15. Fowler, Martin (2014). “Bounded Context”. Martin Fowler.
16. Martin, Robert C (2015). “The Programmer's Oath”. The Clean Code Blog.