COMPOSITE COMPONENTS 2.0, TEIL 4

Der Komplexität die Zügel anlegen

Jeder kennt Software, die kaum oder gar nicht wartbar ist. Das sollten Sie von Anfang an vermeiden.

Ein Softwareprojekt hat meist kein definiertes Ende, sondern wächst kontinuierlich, zum einen im Umfang, zum anderen auch in der Komplexität. Dabei sind Projektlaufzeiten von vielen Dekaden heutzutage keine Seltenheit, sondern in vielen Unternehmen ganz normaler Alltag.

Die letzte Episode von Davids Deep Dive [1] hat sich dem Thema Modularisierung gewidmet und gezeigt, wie mithilfe des Single-Responsibility-Prinzips (SRP [2]) als Modularisierungsmethode ein Ergebnis bei der Aufteilung der Anwendung erzielt werden kann, das eine geringe Kopplung und hohe Kohäsion hat. Die Erkenntnis war, dass dies zu langfristig sehr gut wartbaren und erweiterbaren Softwaresystemen führt. Die vorliegende Episode nun betrachtet die Hintergründe und zeigt, warum genau diese Modularisierungsmethode zu eben genau diesen Entwicklungseigenschaften führt.

Der Weg

Wenn ein Softwaresystem kontinuierlich wächst, altert dieses System selbstverständlich auch. Dabei ist entscheidend, wie viel Zeit in Funktionen und wie viel in Softwarequalität investiert wird. Als Faustregel gilt: Je weniger Zeit in die Softwarequalität fließt, umso schneller altert das System. Schreitet dieser Alterungsprozess schneller voran, wird ein System schwieriger zu warten und weiterzuentwickeln.

Ein häufiges Phänomen, das Ihnen sicherlich bekannt ist: Die Implementierung einer Funktion zu Beginn eines Projekts dauert nur wenige Tage, nach einigen Jahren jedoch würde die gleiche Funktion mehrere Wochen an Zeit benötigen. Dies ist in der Regel darauf zurückzuführen, dass ein System immer komplexer wird und somit die Neuimplementierung – und natürlich auch die Bugfixes – immer mehr Zeit in Anspruch nehmen.

Die Problemspirale

Das oben beschriebene Problem ist selbstverständlich nur ein mögliches Phänomen von vielen. Meist handelt es sich um eine Spirale, die durch folgende Faktoren innerhalb des Projekts ausgelöst wird:

- hohe Komplexität
- viel Zeitaufwand für Features und Bugfixes
- mehr Fehler durch Seiteneffekte
- mehr Support-Aufwand
- weniger Zeit für Entwicklungen
- sinkende Kundenzufriedenheit

Vergrößert sich also im Laufe des Projekts seine Komplexität immer weiter, nimmt die Anzahl der Fehler und damit die Support-Arbeit in der Entwicklung ständig zu. Diese Mehraufwände sorgen dafür, dass immer weniger Zeit für die Entwicklung und damit für neue Funktionen zur Verfügung steht. Mit der wachsenden Anzahl von Bugs und der abnehmenden Zahl der neuen Funktionen sinkt meist auch rasch die Kundenzufriedenheit. Diese Spirale dreht sich weiter, bis nur noch eine teure Neuentwicklung die Software retten kann.

Die Domänenkomplexität

Vor einem genaueren Blick auf das Problem ist es wichtig, die hauptsächlichen Komplexitäten in der Softwareentwicklung zu kennen und zu verstehen.

Die Aufgabe eines Softwareentwicklers ist es, ein Problem in einer Domäne (zum Beispiel eine Bank, Versicherung, im Eventmanagement und so weiter) aus fachlicher Sicht zu verstehen, obwohl er normalerweise keinen oder kaum fachlichen Hintergrund in dieser Hinsicht hat. Diese Domäne besteht normalerweise aus einer Menge von Domänenbegriffen und Geschäftsabläufen, die alle jeweils eine gewisse Komplexität aufweisen. Zusammengefasst werden diese Komplexitäten als Domänenkomplexitäten bezeichnet, im Folgenden DK.

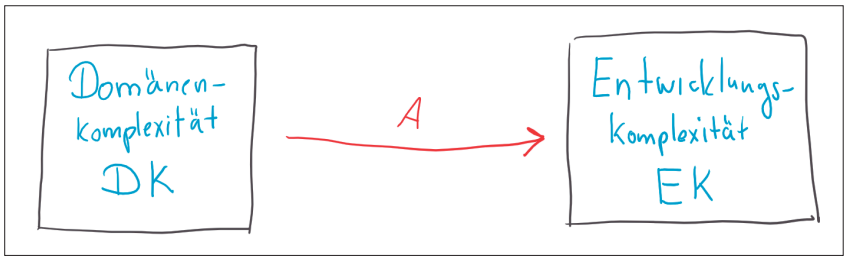
Die Entwicklungskomplexität

Werden die Domänenaufgaben nun in Quellcode umgewandelt, so entsteht dabei auch eine Komplexität, die sogenannte Entwicklungskomplexität, im Folgenden abgekürzt als EK.

Diese Komplexität bildet sich – für die Entwickler am einfachsten ersichtlich – in der Projektmappe der IDE ab, bei .NET also in Form der Visual-Studio-Projektmappe. Jeder Entwickler kennt sicherlich das Problem, dass eine Projektmappe irgendwann so verworren, unaufgeräumt und somit komplex ist, dass einzelne Elemente kaum noch aufzufinden sind und zusammen mit der komplexen Struktur im Code selbst die Entwicklung zu einem Albtraum werden kann.

Die Abbildung

Wenn nun ein Geschäftsvorfall aus der DK in der EK umgesetzt werden soll, muss ein Entwickler sich selbstverständlich Gedanken dazu machen, wie die Datenobjekte und Prozesse (aus der DK) richtig mit der vorhandenen EK und den dort verfügbaren Technologien umzusetzen sind. Man könnte also sagen, es werden die Oberflächen gestaltet, Controller, Models, Verwaltungsklassen, Datenklassen, Mapping-Klassen zur Datenbank und so weiter entworfen. Abstrakt gesehen, handelt es sich um eine Abbildung (im Folgenden nur als A bezeichnet) von einer Komplexität (DK) in eine andere Komplexität (EK). Diese Abbildung A gilt selbstverständlich nicht nur für Neuentwicklungen, sondern auch für Fehlerbehebungen. Dort ist sie zwar wesentlich komplexer, aber für die Betrachtung hier reicht diese vereinfachte Darstellung. Der Zusammenhang von DK, EK und A ist in Bild 1 dargestellt.



Zusammenhang zwischen den Komplexitäten und ihrer Abbildung (Bild 1)

Steigende Komplexitäten

Wie bereits erwähnt, werden Projekte ja allgemeinhin kontinuierlich weiterentwickelt: Es werden neue Domänenbereiche hinzugefügt, neue Geschäftsprozesse angelegt, geändert und so weiter. Mit dieser steigenden Anzahl an Elementen aus der Domäne steigt natürlich auch die Domänenkomplexität DK kontinuierlich an. Gehen wir von der unrealistischen Annahme aus, dass ein Projekt unendlich lange entwickelt wird, so würde sie gegen unendlich steigen.

Das Wachstum der Entwicklungskomplexität hängt nun sehr stark von der verwendeten Modularisierung ab. Angenommen, es gibt immer einen Geschäftsprozess aus einem Domänenbereich.

Dazu sei zunächst das Beispiel einer chaotischen Modularisierung betrachtet: Ein Entwickler speichert alle erzeugten Quellcodes im Ordner *A* in Projekt *X*; ein anderer packt den gesamten Quellcode in die Eventhandler der Oberfläche, und wieder ein anderer legt die Funktionalität in einer gespeicherten Prozedur in der Datenbank ab. Es würde also jeder Entwickler eine unterschiedliche Struktur und damit Komplexität für seine Implementierung wählen.

Im Kontrast dazu würde eine ordentliche Modularisierung etwa wie folgt aussehen. Zuerst wird eine Richtlinie für die Entwickler erstellt: Jeder Geschäftsprozess wird in nur einer Methode codiert, die den Namen des Geschäftsvorfalls trägt und in einer Klasse platziert wird, die nach dem Domänenbereich benannt ist. Neu angelegt würde ein Kunde beispielsweise in *Kunde.Neuanlage()*, der Monatsabschluss eines Kontos würde über *Konto.Abschluss()* berechnet.

Wie Sie leicht erkennen können, erzeugt die chaotische Modularisierung wesentlich mehr Komplexität als die ordentliche Modularisierung. Die zuvor vorgeschlagene Implementierungsvorschrift ist stark vereinfacht, selbstverständlich ist dies in der Realität komplexer: Tabellen, Frameworks, Mappings, Entwurfsmuster, Workflows, Controller, Microservices, Single-Page-Anwendungen – das sind nur ein paar Themen, die während der Implementierung zu berücksichtigen sind. Aber behalten Sie den Kern der Aussage im Kopf: Wird eine Richtlinie für die Modularisierung vorgegeben, erzeugen alle Entwickler dieselbe Komplexität in der Anwendung, während ohne eine solche Richtlinie das Chaos ausbricht.

Komplexität der Abbildung

In der Entwicklungskomplexität sowohl der chaotischen als auch der ordentlichen Modularisierung gibt es eine Abbildung, wie sie Bild 1 zeigt; wie bereits erwähnt, gilt die Annahme, dass die Domänenkomplexität gegen unendlich wächst.

Liegt eine chaotische Modularisierung vor, so wächst die Entwicklungskomplexität EK ebenfalls an und steigt gegen unendlich – und damit wächst auch die Komplexität der Abbildung gegen unendlich. Dieser Satz könnte auch einem der theoretischen Bücher entstammen, die Sie schon im Studium oder während der Ausbildung nicht besonders gemocht haben, daher noch mal praktisch formuliert: Wenn während des Entwickelns die Komplexität des Quellcodes, der Architektur und der Projektmappe immer weiter wächst (EK), dann dauert es auch immer länger, eine Anforderung aus der Domäne (DK) zu implementieren (A).

Aber warum machen eine ordentliche Modularisierung und eine damit verbundene Richtlinie wie das Single-Responsibility-Prinzip (SRP) das Ganze besser? Das vereinfachte Implementierungsbeispiel von oben hat gezeigt, dass ein Geschäftsprozess bei der Implementierung immer auf eine Standardstruktur abgebildet wird. Diese Struktur hat natürlich auch eine Komplexität, aber die nächste Anforderung wird wieder nach derselben Struktur implementiert, und somit wächst die Komplexität nicht, sondern sie ist konstant. Egal wie viele Geschäftsvorfälle zu implementieren sind: Die Domänenkomplexität wächst zwar, die Entwicklungskomplexität bleibt jedoch gleich, und somit wächst auch die für die Abbildung benötigte Leistung nicht, sie ist also ebenfalls konstant. Auf die praktische Arbeit bezogen: Egal wie lange ein

Projekt schon läuft, die Implementierung einer Funktion dauert am Anfang genauso lange wie nach fünf, zehn oder fünfzehn Jahren Projektlaufzeit.

Daher sollten Sie immer eine ordentliche Modularisierung wählen, die durch eine Richtlinie allen Entwicklern vorgeschrieben wird.

SRP macht es noch besser

Die letzte Episode hat als ordentliches Modularisierungsverfahren das SRP vorgestellt, das die Aufteilung eines Softwareprojekts an der Struktur der Domäne ausrichtet. Damit gibt es – das sei schon mal verraten – ein ordentliches Modularisierungsverfahren.

Nun hat die Modularisierung mit SRP aber noch eine zweite, zweifelsohne geniale Eigenschaft. Die Domänenkomplexität besteht streng genommen aus Aufgaben in der Domäne und den darin enthaltenen Unteraufgaben und so weiter. Das ist die Domäne!

Nun nutzen wir aber genau dieselbe Struktur der Domäne auch in der Entwicklung, das heißt, die Domänenkomplexität und die Entwicklungskomplexität sind identisch. Dadurch wird die Abbildungsleistung A sehr klein und bei Kenntnis der entsprechenden Richtlinie ist sie weitestgehend zu vernachlässigen.

Praktisch heißt dies, dass die Struktur der Projektmappe fast identisch ist mit der Struktur der Domäne. Hat man also die Domäne verstanden, ist auch die Navigation in der Entwicklungsstruktur kein Problem mehr. Somit ist langfristig nur noch ein minimaler Aufwand in der Entwicklung nötig und es ergibt sich langfristig eine schnelle Entwicklungsgeschwindigkeit in einem Projekt.

Fazit

In der Praxis lässt sich die Abbildungsleistung natürlich nicht auf null bringen, sondern es gibt noch andere Komplexitäten: Es müssen Programmiersprachen, Frameworks, Patterns und vieles mehr beherrscht werden, um eine Implementierung zu ermöglichen. Das sind aber allesamt konstante Komplexitäten. Das heißt: Wenn sie einmal erlernt sind, können sie in einem „unendlichen" Projekt immer wieder verwendet werden. Egal wie komplex eine Domäne wird, diese Komplexitäten bleiben gleich und wachsen nicht ins Unendliche.

Daher gilt die getroffene Aussage auch für die Praxis: Zum einen können konstante Komplexitäten durch Schulungen oder Richtlinien bewältigt werden; zum anderen kann eine ordentliche Modularisierung wie die nach SRP die Entwicklungskomplexität EK selbst bei steigender Domänenkomplexität DK weitestgehend konstant halten bei einer ebenfalls konstanten Abbildungsleistung A. So funktioniert professionelle Softwareentwicklung.

Wie zuvor gibt es auch dieses Mal bei YouTube ein Video zum Thema [3]. Viel Spaß bei der ordentlichen Modularisierung Ihrer Anwendungen! ■

[1] David Tielke, Sauber geteilt, Composite Components 2.0, Teil 4, dotnetpro 2/2020, Seite 38ff., www.dotnetpro.de/A2002DDD

[2] Wikipedia, Single-Responsibility-Prinzip, www.dotnetpro.de/SL2003DDD1

[3] David Tielke, Warum Software unwartbar wird, www.dotnetpro.de/SL2003DDD2

David Tielke
ist freiberuflicher Berater und von Microsoft zertifizierter Trainer für die Anwendungsentwicklung auf der .NET-Plattform. Darüber hinaus hat er sich auf die Bereiche Softwarearchitektur, Softwarequalität und ALM spezialisiert.
mail@david-tielke.de

dnpCode A2003DDD