Einschränkung der Größe migrierter Micro-Services

Harry M. Sneed
SoRing Kft. H-1221 Budapest
Harry.Sneed@T-Online.de

Abstrakt: Eine Frage, die im Zusammenhang mit der Migration zu Mikro-Services aufkommt, ist die Frage nach deren Größe. Was heißt Mikro? Wie groß darf ein derartiger Service sein und wie wird die Größe gemessen? Bei der Gewinnung von Services aus bestehenden Code müssen diese Fragen beantwortet werden. Der vorliegende Beitrag zu System Reengineering schlägt vor, die Größe von reengineered Services nach dem Testaufwand zu richten. Er ist ein Versuch den Begriff Mikro-Service zu präzisieren.
Schlüsselwörter: Micro-Services, Modularität, Testaufwand, Reverse und Re-Engineering, Code-Transformation, Codekapselung.

1 Wie groß darf ein Micro-Service sein?

In der Literatur zu Micro-Services heißt es Services haben möglichst klein zu sein, aber was heißt hier möglichst klein. Wir brauchen Zahlen nach denen wir uns richten können. Den Code den wir als Micro-Service verwenden wollen darf eine gewisse Größengrenze – sei es Anweisungen, Data-Points, oder Function-Points - nicht überschreiten. Ausschlaggebend für diese Größe ist der Aufwand, den wir betreiben müssen, die Service zu testen. Da der Servicetest mehrfach wiederholt wird, muss er in einer möglichst kurzer Zeit durchführbar sein, eine Zeit die zum Zyklus einer agilen Entwicklung passt, d.h. maximal 10 Tage.
Zwei Fragen stellen sich in diesem Zusammenhang:
1) wie messen wir die Größe eines Micro-Services und 2) wie sichern wir, dass diese Größe nicht überschritten wird. Diese beiden Fragen stehen in Mittelpunkt dieser kurzen Abhandlung.

2 Eigenschaften eines Micro-Services

Ein Micro-Service ist eine aufrufbare Software Komponente mit einer begrenzten Funktionalität und einer Standard web-basierten Schnittstelle nach außen. Diese Schnittstelle könnte SOAP oder REST sein. Der Service enthält ein Request und gibt eine Response zurück. Das Format der Schnittstelle ist normiert, d.h. der Service ist in jeder Umgebung die diese Art Kommunikation zwischen verteilter Komponente unterstützt, erlaubt. Neben der Verarbeitung von Aufträgen hat er auch eine vorgeschriebene Behandlung von Fehlerzuständen – eine Exception Handling. In einer service-orientierten Architektur werden Services verwendet, um gemeinsame, wiederholte Funktionen zu implementieren. In einem Micro-Service sind die Funktionen so tief gegliedert, dass jede einer Operation entspricht, die unabhängig aufrufbar ist[Newman15]..
Die erste Herausforderung liegt also darin, die Anwendungsfunktionen soweit aufzubrechen, dass jede Grundfunktion in einer Operation gekapselt werden kann. Der Micro-Service umfasst mehrere solcher Operationen die alle die gleichen Daten verwenden. Die zweite Herausforderung ist die vielen elementare Funktionen so miteinander zu verbinden, dass sie ein Ganzes bilden, z.B. die elementaren Funktionen eines Geschäftsprozesses wie Reisebuchung sind so zusammengesetzt, dass sie eine geschlossene Handlung bilden, dennoch können die gleiche elementare Funktionen in einem anderen Prozess wie z.B. einer Buchbestellung, verwendet werden.

3 Aufwandsbegrenzung in agilen Projekten

Der Aufwand für die Entwicklung neuer Services wird weitgehend von dem Aufwand für den Test geprägt. Das Verhältnis ist nach allen Erfahrungen mindestens 40:60 %, d.h. 40% für den Entwurf und die Kodierung und 60% für den Test. Durch die Wiederverwendung alter Klassen als Services, bzw. Wrapping, lässt sich der Aufwand reduzieren. Da der Entwurf und ein Großteil des Coding wegfallen, ist das Verhältnis eher 20:80, d.h. 20% für die Codeumwandlung und 80% für den Test. Der Aufwand für die Wiederverwendung von Legacy Code als Services wird vom Test der Services getrieben, auch dann, wenn der Servicetest weitestgehend automatisiert ist. [Sned09].
Die Entwickler kommen nicht darum herum die Service Schnittstellen mit verschiedenen Parameter-kombinationen zu testen und alle relevanten Pfade durch das Service zu durchlaufen. Sämtliche Anweisungen müssen mindestens einmal ausgeführt werden. Das erfordert Zeit. Je mehr Parameter, je mehr Pfade und je mehr Anweisungen es gibt je höher der Testaufwand. Ergo wächst der Testaufwand proportional zur Servicegröße- und -komplexität. Um den Aufwand zu begrenzen und dadurch die Projektlaufzeit zu kürzen muss die Größe und Komplexität der Services begrenzt bleiben. Zwar lassen sich viele Testaufgaben automatisieren, aber die Auswahl der Testeingaben und die Voraussage der Testausgaben können zurzeit nur von Menschen bewältigt werden. Je mehr Testfälle zu konzipieren sind, desto höher der Testaufwand. Der Test von 10 Service-Testfällen kann bis zu einer Stunde dauern. Der Test von 100 Testfällen kostet einen Arbeitstag.
Es stellt sich die Frage – wie viel Testaufwand kann sich ein agiles Projekt leisten, ohne den Zeitrahmen von maximal zwei Wochen pro Zyklus zu sprengen. Zu bedenken ist, dass der Test nur 50% des Gesamtaufwandes beträgt. Daraus folgt, dass der Test allenfalls eine Personenwoche in Anspruch nehmen darf. Dieses Ziel ist nur zu erreichen, wenn das Testobjekt nicht mehr als 200 Zweige und 50 Parameter hat. Denn die Erfahrung mit dem Test von Java Services

zeigt, dass der Tester für den Test eines einzelnen Requests mit Selenium eine Viertelstunde braucht, wenn er auch noch das Response sorgfältig prüft. Ein Request ist gleich einem Testfall. Wenn der Testet nur die Hälfte der Zeit produktiv ist, wie von Linda Crispin behauptet, sind das 16 Testfälle pro Tag [Crisp09].
Fazit ist: Der Testumfang muss sich nach der verfügbaren Zeit richten, nicht umgekehrt. Wenn 10 Tage pro Service vorgesehen sind und davon 6 Tage für den Test können in der Zeit nicht mehr als 100 Testfälle pro Service getestet werden. Die Testzeit bestimmt wie groß ein Service sein darf.

4 Kriterien für die Servicegröße

Die Größe eines Micro-Service lässt sich auf zweierlei Weise messen:

- zum ersten, extern an der Menge der Parameter, bzw. ausgetauschter Daten und
- zum zweiten intern an der Länge des Codes.

Die Menge der übergebenen Daten bestimmt die Breite einer Schnittstelle. Sie ist die Anzahl einzelner Parameter in der Request sowie die Anzahl einzelner Ergebnisse in dem Response. Die Summe der Parameterdaten ergibt die Schnittstellenbreite. Je breiter die Schnittstellen, desto größer den Aufwand sie zu generieren und zu validieren. Der Testaufwand wird durch die Menge der Ein- und Ausgaben bestimmt. Demzufolge darf ein Service nur so viele Parameter haben wie das Entwicklungsteam Zeit hat sie zu testen. Schon bei der Transformation der Klassen in Services werden die Parameter gezählt. Wenn sie eine bestimmte Anzahl überschreitet wird dies gemeldet und der zuständige Entwickler aufgefordert die Anzahl der Parameter in den alten Operationen oder die Anzahl der Operationen in der Klasse zu reduzieren.

5 Einschränkung der Parameteranzahl

Die erste Vorbedingung für die Generierung von Micro-Services aus bestehenden Klassen ist die Einschränkung deren Parameteranzahl. Jede Eingangsparameter muss beim Testen der Service gezielt gesetzt werden. Da dies bisher nicht automatisierbar war, muss das ein menschlicher Tester besorgen. Ein Eingangsparameter zu versorgen und ein entsprechendes Ausgangs-parameter zu prüfen kostet bis zu einer halben Stunde. In einer Operation mit vier Parameter sind das 4 x 4 = 16 Parameterkombinationen Wenn ein Service fünf solcher Operationen hat sind das 5 x 16 = 80 Testfälle aus der Sicht der Daten. Das bedeutet 5 Testtage..

6 Einschränkung der Anweisungsanzahl

Die zweite Vorbedingung für die Generierung von Micro-Services aus bestehenden Code ist die Einschränkung der Anweisungsanzahl. Die Länge des Codes wird anhand der Anzahl Anweisungen bestimmt. In einem Java System für die Abrechnung von Arztkosten für Rentner gab es neben 2745224 Codezeilen, 1094684 Anweisungen, 262389 Zweige und 124364 Pfade. In einem zweiten Java System für die elektronische Gesundheitsakte gab es 1808598 Codezeilen, 510185 Anweisungen, 161077 Zweige und 77295 Pfade. In dem zweiten System waren es 9 Anweisungen pro Pfad. Demnach wäre das Verhältnis von Anweisungen zu Testfällen 8:1. 256 Testfälle würden 2048 Anweisungen bedeuten.
Die zweite Vorbedingung für die Generierung von Micro-Services ist die Beschränkung der Codegröße. Die Höhe des Service-Testaufwandes hängt also in zweiter Linie von der Anzahl der Anweisungen ab. Wer sorgfältig testen will, muss dafür sorgen, dass jede Anweisung mindestens einmal durchlaufen wird, d.h. Tester müssen Argumente zuweisen, die jeden Pfad durch den Service ansteuern. Die Anzahl Pfade durch den Service-Code hängt von der Anzahl Bedingungen ab. Jeder Bedingungsausgang ist eine neue Verzweigung und jede Verzweigung führt zu einem weiteren Pfad. Wie bei den Parametern werden hier die Anzahl Zweige gezählt. Ein Java Service dürfte nicht mehr als 2000 Anweisungen groß sein. Diese Grenze wird vom Wrap Tool kontrolliert. Wenn die maximal zulässige Zahl überschritten wird, wird der Reengineer aufgefordert die alte Klasse zu refaktorieren oder mit einem anderen Tool refaktorieren zu lassen. Jedenfalls wird die Kapselung der alten Klasse an dieser Stelle unterbrochen.

7 Kapselung der Legacy Services

Für die Kapselung der aufgeteilten Klassen wird das Tool SoftWrap eingesetzt. In SoftWrap werden zunächst die public Methoden ausgesucht. Die Java Schnittstelle zu jeder öffentlichen Methode wird in eine WSDL bzw. in eine REST Schnittstelle umgewandelt in dem die Parameter Types transformiert und die Methoden in Operationen umgewandelt werden. Jede Operation bekommt eine Input Schnittstelle mit dem Request und eine Output Schnittstelle mit dem Response. Dazu kommt eine Operation für Exception Handling. Der Benutzer braucht lediglich die Operationen über deren Schnittstelle aufzurufen. Der Code der ursprünglich in den Methoden war, ist jetzt in den Operationen eingebettet. Die Klassendaten sind jetzt Service-Daten und können von allen Operationen verwendet werden. Lokale Methodendaten sind jetzt in den Operationen eingebettet und dürfen nur von der jeweiligen Operation verwendet werden. Das Resultat ist eine Menge wiederverwendbarer und testbarer Micro-Services in einer service-orientierter Architektur.

8 Referenzen

(Newman15] Newman, S. Building Microservices, Oreilly Pub., Boston, 2015
[CrGr09] Crispin, L. / Gregory, J.: Agile Testing – A practical Guide for Testers and agile Teams, Addison-Wesley-Longman, Amsterdam, 2009
[Sned09] Sneed, H.: „A Pilot Project for migrating COBOL Code to Web Services", International Journal of SW-Tools, Nr. 11, 2009, S. 441