Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Neubrandenburg, 13. Februar 2015

Niels Gundermann

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Motivation

In größeren Software-Systemen ist es üblich, dass mehrere Komponenten miteinander über Schnittstellen kommunizieren. In der Regel werden diese Schnittstellen so konzipiert, dass sie Informationen oder Services anbieten, die von anderen Komponenten abgefragt bzw. benutzt werden können. Dabei wird zwischen der Komponente, welche die Schnittstelle implementiert, als angebotene Komponente und der Komponente, welche die Schnittstelle nutzen soll, als nachfragende Komponente unterschieden (siehe Abb. 1).

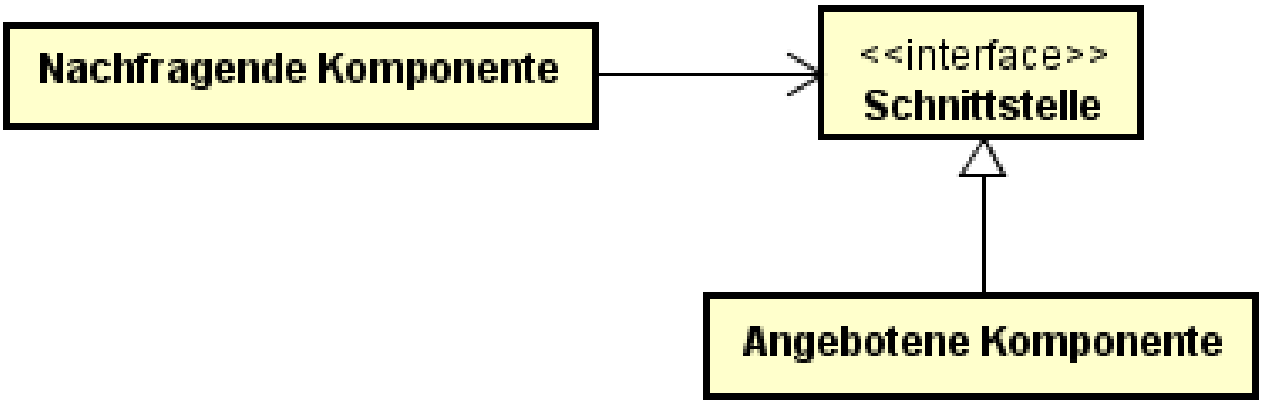

Abbildung 1: Abhängigkeiten von nachfragenden und angebotenen Komponenten

Wird von einer nachfragenden Komponente eine Information benötigt, die in dieser Form noch nicht angeboten wird, so wird häufig ein neues Interface für diese benötigte Information erstellt, welches dann passend dazu implementiert wird. Dabei muss neben der Anpassung der nachfragenden Komponente auch eine Anpassung oder Erzeugung der anbietenden Komponente erfolgen und zusätzlich das neue Interface deklariert werden. Zudem bedingt eine nachträgliche Änderung der neuen Schnittstelle ebenfalls eine Anpassung der drei genannten Artefakte.

In einem großen Software-System mit einer Vielzahl von bestehenden Schnittstellen ist eine gewisse Wahrscheinlichkeit gegeben, dass die Informationen oder Services, die von einer neuen nachfragenden Komponente benötigt werden, in einer ähnlichen Form bereits existieren. Das Problem ist jedoch, dass die manuelle Evaluation der Schnittstellen mitunter sehr aufwendig bis, aufgrund von unzureichender Dokumentation und Kenntnis über die bestehenden Schnittstellen, unmöglich ist.

Weiterhin ist es denkbar, dass ein Software-System auf unterschiedlichen Maschinen verteilt wurde und dadurch Teile des Systems ausfallen können. Das hat zur Folge, dass die Implementierung bestimmter Schnittstellen nicht erreichbar ist. Dadurch, dass eine Schnittstelle durch eine nachfragende Komponente explizit referenziert wird, kann eine solche Komponente nicht korrekt arbeiten, wenn die Implementierung der Schnittstelle nicht erreichbar ist, obwohl die benötigten Informationen und Services vielleicht durch andere Schnittstellen, deren Implementierung durchaus zur Verfügung stehen, bereitgestellt werden könnten.

Dies führt zu der Überlegung, ob es nicht möglich ist, dass eine nachfragende Komponente einfach selbst spezifizieren kann, welche Informationen oder Services sie erwartet, wodurch auf der Basis dieser Spezifikation eine passende anbietende Komponente gefunden werden kann.

2 Verwandte Arbeiten

Ein solcher Ansatz wurde bereits in [BNL+06] von Bajaracharya et al. verfolgt. Bajaracharya et al. entwickelten eine Source Engine namens Sourcerer, welche Suche von Open Source Code im Internet ermöglichte. Darauf aufbauend wurde von derselben Gruppe in [LLBO07] ein Tool namens Code-Genie entwickelt, welches einem Softwareentwickler die Code Suche über ein Eclipse-Plugin ermöglicht. In diesem Zusammenhang wurde offenbar erstmals der Begriff der Test-Driven Code Search (TDCS) etabliert. Parallel dazu wurde in Verbindung mit der Dissertation Oliver Hummel [Hum08] ebenfalls eine Weiterentwicklung von Sourcerer veröffentlicht, welche unter dem Namen Merobase bekannt ist. Das Verfahren, auf dem beide Search Engines grundlegend aufbauen wird, wie bereits angedeutet, als TDCS bezeichnet. Dieses Verfahren beruht grundlegend darauf, dass der Entwickler Testfälle spezifiziert, die im Anschluss verwendet werden, um relevanten Source Code aus einem Repository hinsichtlich dieser Testfälle zu evaluieren. Damit kann das jeweilige Tool dem Entwickler Vorschläge für die Wiederverwendung bestehenden Codes unterbreiten.

Bezogen auf die am Ende des vorherigen Abschnitts formulierte Überlegung ermöglichen die genannten Source Engines, das Internet nach bestehendem Source Code zu durchsuchen und damit bereits bestehende Implementierungen für eine nachfragende Komponente zu ermitteln.

3 Gegenstand dieser Arbeit

In dieser Arbeit soll jedoch nicht das gesamte Internet als Quelle oder Repository für die Codesuche dienen. Vielmehr wird der Suchbereich weiter eingeschränkt.

Es wird von einem System ausgegangen, in dem ein EJB-Container zur Verfügung steht. Die Suche soll sich auf die Menge der angemeldeten Bean-Implementierungen beschränken. Die angemeldeten Bean-Implementierungen stellen damit die Menge der angebotenen Komponenten dar. Dabei wird eine angebotenen Komponente als Kombination eines Interfaces, welches die Schnittstelle für die Aufrufer definiert, und einer Implementierung des Interfaces. Das Interfaces einer angebotenen Komponenten wird im Folgenden auch als angebotenes Interfaces bezeichnet. Die Beans werden bspw. als Provider für Informationen oder im weitesten Sinne auch als Services verwenden, die von unterschiedlichen Komponenten des Systems verwendet werden. Bei der Entwicklung bzw. Weiterentwicklung einer Komponente kann es zu folgenden Szenario kommen, welches durch die Erfüllung der unten aufgeführten Annahmen charakterisiert wird:

- Es werden Informationen und Services benötigt, bei denen der Entwickler davon ausgehen kann, dass es innerhalb des Systems angebotene Komponenten gibt, die diese Informationen liefern können bzw. die Services erfüllen.
- Der Entwickler weiß nicht, über welche konkreten angebotenen Komponenten er die Informationen abfragen bzw. die Services in Anspruch nehmen kann.

3.1 Funktionale Anforderungen

In dieser Arbeit soll ein Konzept entwickelt werden, welches dem entwickler ermöglicht die Erwartungen an die angebotenen Komponenten zu spezifizieren. Darauf aufbauend soll ein Algorithmus vorgeschlagen werden, welcher die angebotenen Komponenten zur Laufzeit hinsichtlich der spezifizierten Erwartungen des Entwicklers evaluiert und eine Auswahl derer trifft, die diese Erwartungen erfüllen. Da die Evaluation zur Laufzeit durchgeführt wird, kann der Entwickler anders als bei den oben genannten Arbeiten nicht aus einer Liste von Vorschlägen auswählen, welche der evaluierten Komponenten letztendlich verwendet werden soll. Diese Entscheidung ist durch den Algorithmus zu treffen.

3.2 Nichtfunktionale Anforderungen

Aufgrund bestimmter Konfigurationen des Gesamtsystem gibt es folgende weitere nichtfunktionale Anforderungen:

- Die Suche muss innerhalb des Transaktionstimeouts von 5 Minuten zu einem Ergebnis führen.
- Die Suche soll hinsichtlich der Besonderheiten des System, in dem sie verwendet wird, angepasst werden können. (Bspw. bei der Verwendung bestimmter Typen, deren Fachlogik bei der Suche nicht untergraben werden darf.)
- Bei einem Fehlschlag der Suche, sollen dem Entwickler Informationen zur Verfügung gestellt werden, die eine zielgerichtete Anpassung seiner spezifizierten Erwartungen erlauben.

4 Entwurf des Explorationsalgorithmus

4.1 Voraussetzungen

Die erste Voraussetzung bezieht sich auf die Spezifikation der Erwartungen einer nachfragenden Komponente. Diese soll aus zwei Teilen bestehen.

Der erste Teil soll die Struktur der erwarteten Informationen bzw. Services beschreiben. Der Entwickler soll hierzu die Schnittstelle, die von ihm erwartet wird in der Form beschreiben, wie es in dem vorliegenden System üblich ist. In dem konkreten System, von dem in dieser Arbeit ausgegangen wird, handelt es sich dabei um Java-Interfaces. Die Struktur der erwarteten Informationen bzw. Services wird demnach innerhalb der nachfragenden Komponente durch ein Interface dargestellt, in dem die erwarteten Methoden, die innerhalb der nachfragenden Komponente verwendet werden sollen, deklariert wurden. Ein solches Interface wird im Folgenden auch als erwartetes Interface bezeichnet.

Der zweite Teil besteht aus einer Menge von Testfällen, durch die ein Teil der erwarteten Semantik spezifiziert wird. Hierzu können Testfälle in Methoden mehrerer Klassen implementiert werden. Zur Referenzierung der Testklassen wird eine Annotation @QueryTypeTestReference im erwarteten Interface verwendet. Dort können über den Parameter testClasses mehrere Testklassen angegeben werden. Die Testklassen müssen über den Default-Konstruktor verfügen. Innerhalb der Testklassen werden die Testmethoden mit der Annotation @QueryTest markiert. Weiterhin ist es notwendig, die zu testende Instanz zur Laufzeit in ein Objekt einer Testklasse zu injizieren. Dies erfolgt durch Setter-Injection. Aus diesem Grund muss in jeder Testklasse ein Setter für ein Objekt vom Typ des erwarteten Interfaces implementiert werden und mit der Annotation @QueryTypeInstanceSetter markiert werden.

Listing 1 zeigt ein Beispiel für ein erwartetes Interface, welches eine Testklasse referenziert. Listing 2 hingegen zeigt diese referenzierte Testklasse mit den bereits erwähnten Annotationen für den Setter des zu testenden Objektes und den Testmethoden.

Listing 1: Erwartetes Interface IntubatingFireFighter

```
1 @QueryTypeTestReference( testClasses = IntubatingFireFighterTest.class )
2 public interface IntubatingFireFighter {
3
4   public void intubate( AccidentParticipant injured );
5
6   public void extinguishFire( Fire fire );
7 }
```

Listing 2: Testklasse des erwarteten Interfaces IntubatingFireFighter

```
1 public class IntubatingFireFighterTest {
2
3   private IntubatingFireFighter intubatingFireFighter;
4
5   @QueryTypeInstanceSetter
6   public void setProvider( IntubatingFireFighter intubatingFireFighter ) {
7     this.intubatingFireFighter = intubatingFireFighter;
8   }
9
10  @QueryTypeTest
11  public void free() {
12    Fire fire = new Fire();
13    intubatingFireFighter.extinguishFire( fire );
14    assertFalse( fire.isActive() );
15  }
16
17  @QueryTypeTest
18  public void intubate() {
19    Collection<Suffer> suffer = Arrays.asList( Suffer.BREATH_PROBLEMS );
20    AccidentParticipant patient = new AccidentParticipant( suffer );
21    intubatingFireFighter.intubate( patient );
22    assertTrue( patient.isStabilized() );
23  }
24 }
```

Die zweite Voraussetzung betrifft den Zugang zu den bestehenden angebotenen Schnittstellen und deren Implementierungen in dem bestehenden System. Um in der Menge aller angebotenen Komponenten eine passende Komponente finden zu können, muss diese Menge bekannt sein oder ermittelt werden können. Wie oben beschrieben, wird in dieser Arbeit von einem System ausgegangen, in dem die angebotenen Komponenten als Java Enterprise Beans umgesetzt wurden. So wird dementsprechend eine Möglichkeit geschaffen, sämtliche der angemeldeten JNDI-Namen und die dazugehörigen Bean-Interfaces abzufragen. Die Abfrage der angemeldeten Bean-Implementierungen zu einem JNDI-Namen ist durch den EJB-Container bei Vorliegen des entsprechenden Interfaces und des JNDI-Namens bereits gegeben.

4.2 Explorationskomponente

4.3 Heuristiken

Literaturverzeichnis

[BNL+06] BAJRACHARYA, SUSHIL, TRUNG NGO, ERIK LINSTEAD, YIMENG DOU, PAUL RIGOR, PIERRE BALDI und CRISTINA LOPES: *Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search*. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, Seiten 681–682, New York, NY, USA, 2006. Association for Computing Machinery.

[Hum08] HUMMEL, OLIVER: *Semantic Component Retrieval in Software Engineering*. Doktorarbeit, April 2008.

[LLBO07] LAZZARINI LEMOS, OTAVIO AUGUSTO, SUSHIL KRISHNA BAJRACHARYA und JOEL OSSHER: *CodeGenie: A Tool for Test-Driven Source Code Search*. In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, Seite 917?918, New York, NY, USA, 2007. Association for Computing Machinery.