Auge in Auge mit Ihrer Softwarearchitektur

Modellierung, Visualisierung und Prüfung von Softwarearchitekturen in virtueller Realität

Prof. Dr. Rainer Koschke, Universität Bremen

Dieser Beitrag gewährt einen Einblick in eine mögliche baldige Zukunft, wie Softwarearchitekturen mit Hilfe neuer digitaler Medien modelliert und gegen den Code geprüft werden können. Zudem beschreibt er auch eine allgemeine Methode zur Architekturrekonstruktion und -prüfung, die unabhängig von bestimmten Technologien zur Visualisierung und Interaktion funktioniert. Diese Methode lässt sich natürlich genauso mit traditionelleren Medien wie Computer-Bildschirmen, Tastatur und Maus oder Touchscreens durchführen.

Einführung

Eine Softwarearchitektur beschreibt den fundamentalen Aufbau eines Programms auf höherer Abstraktionsebene. Jedes Programm hat eine Softwarearchitektur, unabhängig davon, ob sie nun einem atemberaubenden Meisterwerk gleicht oder doch eher die Gestalt einer Hütte in einem Slum annimmt. Sie ist wichtig für alle Belange in der Software-Entwicklung: für Implementierung und Wartung genauso wie für Planung und Review zentraler Entwurfsentscheidungen sowie für den Test. Oft ist sie jedoch nicht gut dokumentiert. Wenn sie denn dokumentiert wird, dann werden dazu meist UML- oder gar einfache Zeichen-Werkzeuge verwendet, die dann meist ein idealisiertes, unzutreffendes Bild abgeben. In diesem Beitrag werden aktuelle Bestrebungen aus der Forschung vorgestellt, wie Softwarearchitekturen kooperativ in virtueller Realität modelliert, visualisiert und geprüft werden können. Dabei muss nicht auf der grünen Wiese begonnen werden. Man kann hierfür auch vom existierenden Code ausgehen. Der Einsatz virtueller Realität bietet dazu vielfältige Interaktionsmechanismen und erlaubt es, dass mehrere Menschen sich in einem gemeinsamen virtuellen Raum treffen können, um zusammen eine Softwarearchitektur zu erstellen und mit dem Code abzugleichen. Man wünschte sich, dies gerade in den Zeiten von Corona schon zu haben.

Zum Begriff der Softwarearchitektur

Softwarearchitektur ist ein weiter Begriff. Im Folgenden verstehen wir darunter primär den statischen Aufbau eines Programms in Form statischer Komponenten und deren statischen Abhängigkeiten. Eine statische Komponente beschreibt eine konzeptionelle Einheit des Entwurfs, die Daten speichert oder verarbeitet. Im Kontext des Projektmanagements stellt sie eine Arbeitseinheit dar, die von Entwicklern implementiert wird. Wenn in der Programmiersprache Java entwickelt wird, erstellen die Entwickler für die Komponente Klassen und Schnittstellen eingebettet in Pakete. Im Falle einer C-Entwicklung wird eine Komponente durch C-Quelltext-Dateien umgesetzt, die in der Regel in Verzeichnisse eingeordnet werden. Eine Komponente ist immer nur der Teil eines Ganzen. Sie hat eine Schnittstelle, über die sie andere Komponenten verwendet, und eine weitere Schnittstelle, über die andere Komponente ihre Dienste verwenden können. Verwendet eine Komponente die Schnittstelle einer anderen Komponente, ergibt sich daraus eine Abhängigkeit. Die statische

Softwarearchitektur beschreibt, welche Komponenten mit welchen anderen Komponenten eine Abhängigkeit eingehen dürfen beziehungsweise sollen. Bei Architekturen großer Systeme werden die Komponenten in der Regel weiter in kleinere Komponenten verfeinert, so dass sich eine Hierarchie von Komponenten ergibt. Dies dient der Abstraktion und der Möglichkeit, schrittweise ins Detail gehen zu können. Die Hierarchisierung vereinfacht es auch, Regeln aufzustellen, welche Komponenten einander benutzen dürfen. Üblich sind zum Beispiel Architekturen, die in Schichten strukturiert werden, bei denen die in den Schichten enthaltenen Komponenten nur Komponenten derselben Schicht oder der unmittelbar tieferen Schicht verwenden dürfen. Es ist die Aufgabe des Architektur-Designs, schrittweise Details hinzuzufügen und solche Einschränkungen zu formulieren.

Selbstverständlich gibt es auch Laufzeitarchitekturen, die beispielsweise bei verteilten Systemen beschreiben können, welcher Prozess auf welcher Hardware läuft und wie diese Prozesse miteinander interagieren. Sie haben auch einen zeitlichen Aspekt, das heißt, diese Architekturen können sich dynamisch zur Laufzeit verändern. Im Folgenden betrachten wir jedoch nur solche, die sich nicht über die Laufzeit ändern, deshalb also *statische* Architekturen. Ein Programm hat immer beides: eine statische und eine Laufzeit-Architektur.

Softwarearchitektur als Graph

Statische Architekturen lassen sich in Form eines Graphen beschreiben, dessen Knoten die Komponenten und dessen Kanten die Abhängigkeiten modellieren. Im Falle einer statischen Softwarearchitektur ändern sich dabei die Knoten und Kanten nicht über die Zeit. In gleicher Weise kann man auch die statische Struktur und die Abhängigkeiten des Codes beschreiben. Beispielsweise kann ein Java-Programm in seine Pakete und Klassen zerlegt werden, die sich gegenseitig benutzen. In C existieren Code-Dateien, die oft zur besseren Übersicht in verschiedene Verzeichnissen geschachtelt werden und wechselseitige Bezüge durch Aufrufe und andere Formen statischer Abhängigkeiten aufweisen.

Wenn man sowohl die Architektur als auch die Struktur des Codes als Graph auffasst, lassen sich beide uniform in Form von Diagrammen visualisieren, in denen Knoten durch graphische Elemente wie Rechtecke oder Kreise und die gerichteten Kanten durch Pfeile abgebildet werden, die die Knoten miteinander verbinden. Die Grafik im linken Teil der Abbildung 1 ist ein Beispiel für die Visualisierung der Architektur eines Compilers in Form eines solchen Diagramms. Letztlich sind Paket- und Klassendiagramme der UML auch nichts anderes als eine Verbildlichung von Graphen.

Die Modellierung als Graph hat noch einen weiteren Vorteil: Sie schafft die Möglichkeit eines Abgleichs der Graphen. Eine Architektur ist während der Implementierung eine Vorgabe, wie der Code strukturiert werden soll. Im weiteren Verlauf des Projekts dient sie auch der Dokumentation der Implementierung auf höherem Abstraktionsniveau. Schon während der Implementierung und erst recht in der Wartungsphase eines Programms kommt es aber oft dazu, dass Entwickler Abhängigkeiten im Code einbauen, die in der Architektur nicht vorgesehen waren. Und selbst wenn solche neuen Abhängigkeiten berechtigt sind, dann werden sie nur

selten in der korrespondierenden Architekturbeschreibung ergänzt. Zunehmend laufen Architekturbeschreibung und Implementierung somit auseinander. Wenn die Diskrepanz zu groß wird, verliert die Architekturbeschreibung jedoch ihren Nutzen sowohl als deskriptive als auch präskriptive Dokumentation.

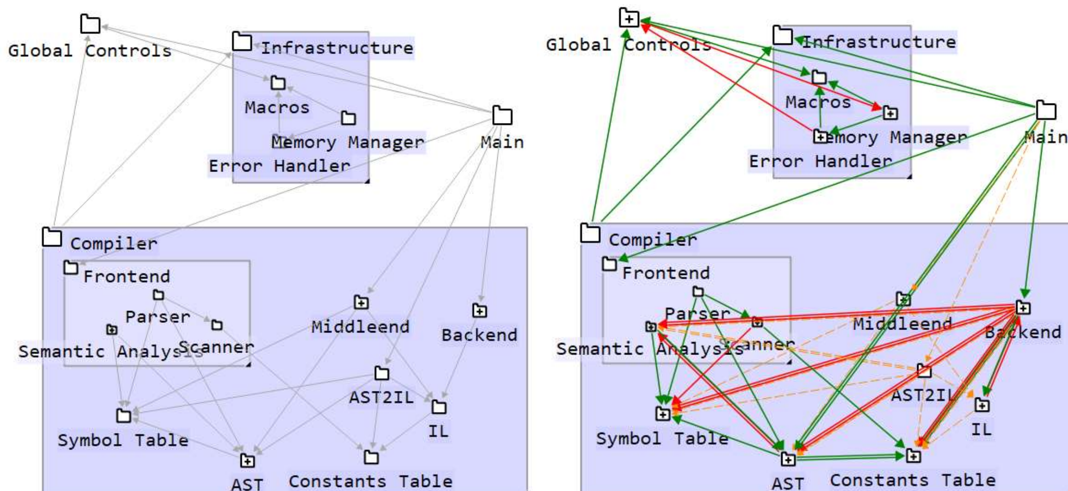

Abbildung 1: Architekturbeschreibung (links) und das Ergebnis ihrer Prüfung (rechts)

Weil aber die Architekturbeschreibungen einen großen Mehrwert auch in der weiteren Evolution eines Programms hat, sollten Anstrengungen unternommen werden, Architektur und Implementierung im Einklang zu halten. Die Modellierung beider als Graphen ermöglicht die automatisierte Prüfung, inwieweit das der Fall ist. Dazu müssen beide Graphen zunächst in Beziehung zueinander gebracht werden, indem beschrieben wird, welche Implementierungskomponenten (z.B. C-Dateien oder Java-Klassen) welche Architekturkomponenten implementieren. In der Terminologie der Graphen heißt das, dass wir die Knoten aus dem Implementierungsgraphen auf jene des Architekturgraphen abbilden müssen. Diese Abbildung wird in der Regel manuell von den Entwicklern beigesteuert. Unter Umständen lässt sie sich automatisieren, zum Beispiel, wenn Namenskonventionen angewandt werden, aus denen der Bezug der beiden zuverlässig hervorgeht.

Wenn diese Abbildung vorhanden ist, kann der Rest vollständig automatisiert werden. Im Folgenden sei $m : I \rightarrow A$ die Abbildung von Knoten eines Implementierungsgraphen I auf Knoten eines Implementierungsgraphen A. Ein Algorithmus prüft dann, ob jede Kante, die zwischen zwei Implementierungskomponenten i_1 und i_2 existiert, auch eine Entsprechung im Architekturgraphen hat; das heißt, es sollte dann eine Kante zwischen $m(i_1)$ und $m(i_2)$ geben. Ist das nicht der Fall, sprechen wir von einer *Divergenz* zwischen Implementierung und Architektur. Beispielsweise gibt es zwischen den beiden Architekturkomponenten *Middleend* und *Backend* im obigen Diagramm keine Kante. Wenn aber Code, der dem *Middleend* zugeordnet ist, Code benutzt, der dem *Backend* zugeordnet ist, dann verstößt die Implementierung gegen die Architekturvorgabe beziehungsweise weist die Architektur eine Dokumentationslücke auf, falls das hätte erlaubt sein sollen. Im letzteren Fall könnte die Architektin das

Problem beheben, indem sie der Architektur eine Kante zwischen den beiden Komponenten hinzufügt. Aus der Divergenz würde dann eine *Konvergenz* werden: Architektur und Implementierung stimmen in diesem Aspekt überein. Möglich ist auch eine Situation, in der eine Architektur eine Abhängigkeit vorsieht und damit auch in der Implementierung erwartet, in der Implementierung aber keine solche Abhängigkeit gefunden werden kann. Dieser Fall wird als *Absenz* bezeichnet.

Bei der hier beschriebenen Prüfung sind wir vereinfachend davon ausgegangen, dass Komponenten nicht geschachtelt sind. Das ist bei realen Architekturen großer Systeme natürlich sehr wohl der Fall. Wie die Prüfung auf hierarchische Architekturen erweitert werden kann, haben wir an anderer Stelle genau beschrieben und wir verweisen den interessierten Leser auf diesen Artikel [1].

Das Ergebnis dieser Prüfung kann seinerseits graphisch dargestellt werden. Im rechten Teil von Abbildung 1 sehen wir die Konvergenzen in Form grüner Pfeile, die Divergenzen als rote Pfeile und die Divergenzen als gestrichelte orangefarbene Pfeile.

Visualisierung von Architekturen in Virtual/Augmented Reality

Architekturen sollen als abstrakte Beschreibungen dienen, auf Basis derer verschiedene Beteiligte sich informiert austauschen können, ohne vom Code erschlagen zu werden. So könnten Tester und Entwickler beispielsweise sich zu Aspekten der Testbarkeit eines Programms austauschen. Experten verschiedener technischer Bereiche (z.B. der Experte des Hardware-Abstraction-Layers und der Experte der spezifischen Logik des Programms) sollen gemeinsam nach Ursachen für Fehler suchen können. Ein Projektmanager möchte sich von den Entwicklern erläutern lassen können, wie der Projektfortschritt in der Implementierung voranschreitet. Würden all diese Beteiligten sich dieser Sachverhalte allein auf Code-Ebene zu nähern versuchen, würden sie den Wald vor lauter Bäumen nicht mehr sehen. Sie würden sehr rasch von ganz alleine anfangen, ad hoc Grafiken ihrer Vorstellung vom System an einem Whiteboard zu erstellen, um gemeinsam über die relevanten Sachverhalte reden zu können. Diese Arbeit kann ihnen abgenommen werden, wenn es bereits eine Visualisierung gibt.

Eine hilfreiche Visualisierung ist nicht nur ein starres Bild, sondern unterstützt Interaktionen, beispielweise um in Details hinein zu zoomen, Unwesentliches auszublenden, Dinge zu suchen und auf Bedarf auch den Quelltext anzuzeigen. Um diese Interaktionen zu ermöglichen, sollte man die Visualisierung somit einem geeigneten Visualisierungswerkzeug übergeben. Bei der heute zunehmend verteilten Entwicklung, beim Trend zum Home-Office und gerade in der Corvid-19-Pandemie kann man allerdings nicht mehr voraussetzen, dass die Menschen, die sich mittels einer Visualisierung austauschen wollen, sich alle gemeinsam vor einem Computer-Monitor versammeln. Ein Visualisierungswerkzeug muss also eine Plattform sein, auf der verschiedene Nutzer zeitgleich und kooperativ mit der Visualisierung interagieren können und zwar von verschiedenen Orten der Welt aus. Natürlich gibt es Videokonferenzsysteme, die auch Screen-Sharing unterstützen. Das sind jedoch generische Werkzeuge für die Kooperation, die kein Wissen darüber haben, was sie eigentlich darstellen. Die Interaktionen mit der Visualisierung werden dadurch nicht direkt unterstützt. Vielmehr muss einer der Betrachter das Tool steuern, dessen

Visualisierung übertragen wird. Üblicherweise hat dann nur einer die Steuerung. Zudem hat durch die Übertragung jeder das gleiche Bild. Es ist nicht möglich, dass einer der Beteiligten andere Teile oder den gleichen Teil aus einer anderen Perspektive betrachtet.

An dieser Stelle können neuere Medien und Technologien helfen. Virtual Reality (VR) und Augmented Reality (AR) gibt es schon seit langer Zeit. Mittlerweile ist die Hardware aber erschwinglich und wird auch zunehmend praxistauglicher, so dass man über ernsthafte Anwendungen außerhalb der Computerspielebranche nachdenken kann. Und tatsächlich werden diese Technologien in der Logistik, Fertigung oder Wartung im Rahmen industrieller Fertigungen bereits erfolgreich eingesetzt. In der Softwaretechnik befinden sich deren Anwendungen jedoch noch im Forschungsstadium. Forscher in der Softwaretechnik haben aber durchaus schon vor zwanzig Jahren begonnen, mit diesen Techniken Software zu visualisieren (für eine Übersicht siehe [2]). Seit dieser Zeit ist eine Reihe von Konzepten, wie Software geeignet dargestellt werden kann, entstanden. Eine Form der Darstellung folgt der Metapher einer Stadt, bei der Komponenten als Gebäude visualisiert werden, die an Straßen liegen oder in Distrikten angeordnet sind. Im englischen Sprachraum ist der Begriff der *Code-City* hierfür gebräuchlich. Diese Darstellungen bedienen sich aller drei Dimensionen. Auch die ersten Ideen der Darstellung von Software als Stadt liegt schon sehr lange zurück. Was jedoch bislang noch nicht wirklich erforscht wurde ist die Visualisierung von Software-Städten in VR/AR mit dem besonderen Fokus auf Kooperation. Bislang wurde lediglich untersucht, inwiefern ein einzelner Betrachter von der Darstellung profitiert. Ein wesentlicher Vorteil von VR/AR ist jedoch, dass er räumliche Grenzen aufhebt. Nutzer können sich in der virtuellen Welt treffen, auch wenn zwischen ihnen tausende Kilometer liegen. Das Potential dieser Technologien in der Softwaretechnik ist somit noch gar nicht ausreichend untersucht. In genau diesem Bereich ist unsere eigene Forschung zu diesem Thema angesiedelt.

Unsere Forschung zu Software-Städten in VR/AR

An der Universität Bremen forschen wir an der Nutzung von VR/AR für die Visualisierung von Software und ihren Architekturen in Form von Software-Städten mit dem Fokus auf kooperativem Verständnis. Hierzu schaffen wir virtuelle Welten, die von mehreren Nutzern zeitgleich betreten werden können und zwar über verschiedene Medien und Technologien hinweg. So könnte der eine Nutzer eine VR-Umgebung nutzen, bei der er ein Head-Mounted-Display trägt und mit Hilfe von in der Hand gehaltenen VR-Controllern interagiert, ein anderer Nutzer könnte eine Hololense 2 tragen, die AR unterstützt und mit Gesten und Sprache gesteuert wird, und ein dritter Nutzer sitzt an einem klassischen Computer mit Monitor, Tastatur und Maus. Alle Nutzer können sich unabhängig voneinander frei in der virtuellen Welt bewegen, verschiedene Perspektiven einnehmen und eigenständig mit der Visualisierung interagieren. Dabei sehen sie sich gegenseitig in Form von Avataren, deren Handgesten für Zeigebewegungen übertragen werden, und sie können miteinander sprechen. Wenn mehrere Nutzer mit AR-Brillen sich im selben Raum befinden, sehen sie sich sogar in Realität und können unmittelbar miteinander kommunizieren.

In unserer Forschung adressieren wir dabei bislang folgende Anwendungsszenarien:

- Die Nutzer schauen dem System bei der Ausführung zu, um einen Fehler zu finden oder das Verhalten der Software zu verstehen. Man sieht hier zum Beispiel, welche Funktionen welche anderen Funktionen aufrufen, sowohl auf der abstrakteren visuellen Ebene aber auch bis hinunter auf Quellcode-Zeilen wie bei einem Quelltext-Debugger.
- Die Nutzer betrachten die Evolution einer Software im Zeitraffer. Man kann sehen, wie sich die Software-Stadt ausbreitet, welche Dinge von einer Version zur nächsten hinzukommen, wegfallen oder sonst irgendwie verändert werden. Damit lassen sich Erklärungen in der Vergangenheit finden oder Trends erkennen. Ein Projektmanager bekommt einen Überblick über den Fortschritt der Entwicklung.
- Die Nutzer wollen einen Einblick in die innere Qualität ihrer Programme bekommen. Hier kann man verschiedene Software-Qualitätsmetriken auf visuelle Attribute wie Höhe, Tiefe, Breite oder Farbe der Gebäude, Straßen oder Distrikte anwenden. Darüber hinaus gehende Metriken lassen sich in klassischen Charts darstellen, mit denen aber interagiert werden kann. So kann man zum Beispiel Punkte in den Charts auswählen und die dazu korrespondierenden Komponenten in der Software-Stadt werden grafisch hervorgehoben.
- Die Nutzer wollen die Architektur prüfen oder rekonstruieren. Dazu bedienen sie sich der oben eingeführten Methode.

Auf das letzte Anwendungsszenario gehen wir im Folgenden etwas näher ein. Es sei vorab angemerkt, dass unsere bisherigen Entwicklungen, die wir vor etwas mehr als zwei Jahren begonnen haben, noch Prototypen sind. Wir haben keine fertigen Produkte. Teile unserer Entwicklungen haben einen unterschiedlichen Reifegrad. Die Darstellung der Software als Stadt am Desktop-Computer und in VR mit einer ganzen Reihe alternativer Layouts existiert jedoch und die ersten drei Anwendungsszenarien sind bereits prototypisch umgesetzt. Auch ist es bereits möglich, wie bei einem Multiplayer-Spiel, dass verschiedene Nutzer sich zeitgleich in der virtuellen Welt in Form von Avataren aufhalten und mit anderen anwesenden Nutzern sprechen können.

Architekturprüfung in VR/AR

Im Folgenden erläutern wir einige Aspekte unserer Visualisierung, die bewusste Design-Entscheidungen sind und die wir hier motivieren werden. In Abbildung 2 sieht man eine Beispielszene aus der Sicht eines Nutzers, der über einen Desktop-Computer Teil der virtuellen Welt ist. Der eingeblendete Avatar gehört zu einem anderen Nutzer, der über eine VR-Umgebung präsent ist. Jeder Nutzer kann ein Bild seines eigenen Gesichts für seinen Avatar verwenden. Hier wird bewusst nicht der ganze Körper dargestellt, da ansonsten Teile unnötig verdeckt werden könnten. Der Kopf genügt, damit andere Teilnehmer erkennen können, wohin ein Nutzer blickt. Die Hände werden bei Bewegungen eingeblendet, so dass Zeigegesten möglich sind und andere erkennen, womit der Nutzer in diesem Moment interagiert. Gerne würden wir selbstverständlich auch die Mimik übertragen können. Da VR/AR-Teilnehmer aber Brillen aufhaben, ist das mit heutiger Technologie noch nicht ohne weiteres möglich.

Es gibt aber durchaus schon erste Entwicklungen, die Bewegungen der Gesichtsmuskeln messen und diese dann auf virtuelle Gesichter übertragen. Wenn die Teilnehmer an einem Desktop-Computer sitzen, der über eine Videokamera verfügt, könnte man aber bereits heute Videos des Gesichts übertragen, wie wir das von Werkzeugen wie Zoom kennen.

Die Szene stellt einen Raum mit Tisch und Sitzgelegenheiten dar. Letztere sind eigentlich unwesentlich, erwecken aber eine etwas realistischere Umgebung. Dieser Raum ist bewusst spartanisch eingerichtet, um unnötige Ablenkungen zu vermeiden. Es ist aber auch möglich, fotorealistische Umgebungen wie ein Büro oder eine Bibliothek zu verwenden. Erlaubt ist, was das Erlebnis verbessert, ohne von der eigentlichen Aufgabe zu sehr abzulenken.

Abbildung 2: Virtuelle Software-Städte für Implementierung und Architektur

Auf dem Tisch sind zwei Software-Städte in verschiedenen Layouts dargestellt. Die linke Stadt ist hierbei der statische Abhängigkeitsgraph aus der Implementierung des oben bereits erwähnten Compilers. Rechts davon sehen wir dessen Architektur. Die Architektur kann in der Welt selbst modelliert oder einfach von einem anderen Werkzeug wie etwa von einem UML-Tool nebst Layout importiert werden.

Eine weitere wichtige Design-Entscheidung von uns ist es, die Software-Städte in Miniatur darzustellen, so dass sich die Nutzer über sie beugen können, stets den Überblick haben und auch Teile davon tatsächlich mit ihren virtuellen Händen greifen können. Zu Anfang unserer Entwicklung gingen wir von einer Stadt aus, die

maßstabsgetreu groß ist und durch die die Nutzer fliegen können. Steht man jedoch auf dem Boden zwischen den Gebäuden, so bietet sich einem Betrachter ein Manhattan-Erlebnis: man blickt von der Straße hoch zu Wolkenkratzern und weiß nicht, wo man eigentlich gerade ist. Wenn mehrere Nutzer kooperieren wollen, dann sollen sie sich auch sehen können. Aus diesem Grund werden die Städte nunmehr vor den Nutzern in Miniatur ausgebreitet. Da bei großen Systemen die Gebäude sehr klein werden können und man deren Details aber sehen möchte, gibt es die Möglichkeit, beliebig in die Städte auf dem Tisch zu zoomen und diese zu verschieben.

Die Abbildung der Implementierungs- auf die Architekturkomponenten ist mit Hilfe direkter Manipulation möglich. Dazu zieht man einfach eine Implementierungskomponente auf eine Architekturkomponente. Hierbei müssen die Besonderheiten der Umgebungen Berücksichtigung finden. An einem Desktop mit Monitor wird man eine Maus verwenden, um Dinge zu selektieren, und Tasten auf der Tastatur drücken, um Aktionen auszulösen. In VR und AR erwartet man aber, dass man die Dinge mit seinen virtuellen Händen anfassen kann. Zudem gibt es an VR-Controllern nur sehr wenige Tasten. Bei einer Hololense 2 gibt es gar keine Controller. Zwar kann man auch bei VR und AR virtuelle Tastaturen einblenden, die sind aber wegen des fehlenden haptischen Feedbacks nicht so einfach zu bedienen. Und man will ja eigentlich gerade nicht eine Desktop-Umgebung in der virtuellen Welt nachbauen. Das wäre, als ob man die Wählscheiben alter Telefone auf ein modernes Handy übertragen wollte. Für ein intuitiveres Erlebnis sollte man also mit Hilfe von Handgesten und Sprache interagieren können.

Das Drag&Drop, um Komponenten aufeinander abzubilden, muss also in den verschiedenen Umgebungen unterschiedlich programmiert werden. Die Änderungen der Visualisierung als Ergebnis der neuen Abbildung ist dann aber wieder in allen Umgebungen einheitlich. Alle Nutzer sollen immer dasselbe sehen, unabhängig davon, welche Art von Interaktionsumgebung sie nutzen.

Die Abbildung wird visuell ausgedrückt, indem die abgebildete Implementierungskomponente in der Software-Stadt der Architektur auftaucht, während ihr Original in der Implementierungsstadt transparent erscheint – sie ist ja nun nicht mehr wirklich da. Die auf diese Weise verschobene Implementierungskomponente wird in die Architekturkomponente in der Architekturstadt, auf die sie abgebildet wurde, eingebettet. Das hat verschiedene Vorteile. Zum einen ist die Abbildung unmittelbar im Kontext der Architektur sichtbar. Alle oben aufgeführten Anwendungsszenarien funktionieren dann nicht nur für die Implementierung alleine, sondern sind ohne weiteres Zutun auf die Architektur übertragbar. Eine Ausführung eines Funktionsaufrufes beim Debugging kann somit auch direkt in Bezug zur Architektur gesetzt werden. Darüber hinaus gewinnt die Anordnung der Implementierungselemente eine Bedeutung. Ein statischer Abhängigkeitsgraph wird normalerweise durch eine statische Code-Analyse automatisch aus dem Code extrahiert. Er wird also nicht von einem Menschen modelliert. Damit hat er zunächst einmal keine Anordnung. Die Anordnung muss ihm durch ein automatisches Layout gegeben werden. Automatische Layouts nutzen für eine übersichtliche Anordnung verschiedene Kriterien, wie zum Beispiel die Minimierung von Kreuzungen der Kanten. Die Semantik der Komponenten ist ihnen aber nicht zugänglich. Menschen

geben Anordnungen allerdings gemäß der Gestaltstheorie Bedeutung. So werden zum Beispiel Dinge, die nah beieinander angeordnet sind, als verwandt wahrgenommen (Gesetz der Nähe). In der Regel unterscheiden sich räumliche Anordnungen von Menschen zu jenen von automatischen Layoutalgorithmen. Durch die manuelle Einbettung der Implementierungs-komponenten und der in die manuelle Anordnung der Architekturkomponenten eingeflossene Semantik, gewinnt die Anordnung von Implementierungskomponente nunmehr auch eine Semantik.

Die Konvergenzen und Absenzen, die sich durch die Architekturprüfung ergeben, werden durch Annotationen der Kanten der Architektur dargestellt. Für Divergenzen gibt es jedoch keine Entsprechung in der Architektur. Sie bezeichnen eine Situation, in der eine Abhängigkeit in der Implementierung existiert, für die es keine Entsprechung in der Architektur gibt. Somit kann hier keine Kante der Architektur annotiert werden. Deshalb werden von der Implementierung auch alle Kanten übertragen, wenn deren beiden Enden auf die Architektur abgebildet wurden. Somit kann man eigentlich bereits als Nutzer erkennen, wo die Unterschiede zwischen Architektur und Implementierung liegen und das durch bloßes Betrachten. Weil es aber viele solche Kanten geben kann, ist ein automatischer Algorithmus dennoch hilfreich, so dass nichts übersehen wird.

Die neue Abbildung einer Implementierungs- auf eine Architekturkomponente kann Auswirkungen auf Konvergenzen, Divergenzen und Absenzen haben. Ein inkrementeller Algorithmus berechnet die sich daraus ergebenden Veränderungen [3]. Die Annotationen werden an den Kanten in der Architekturstadt entsprechend angepasst. Damit sie auch wirklich einfach wahrgenommen werden können, werden sie animiert. Nunmehr können die verschiedenen Nutzer das Ergebnis ihrer Abbildung gemeinsam diskutieren. Unter Umständen nehmen sie eine Abbildung zurück, akzeptieren eine Divergenz, fügen eine Architekturkante hinzu oder entfernen diese. Das können alle Nutzer zeitgleich tun. Aus diesem Grund darf es nicht zu Verzögerungen bei der Aktualisierung der virtuellen Welten auf den verschiedenen Endgeräten kommen; sonst glauben möglicherweise zwei Nutzer gleichzeitig, eine bestimmte Kante oder Komponente gerade selbst in der Hand zu halten. Hierin liegen also auch technische Herausforderungen bei der Synchronisation der virtuellen Welten auf verschiedenen Geräten, die möglicherweise über weite Strecken des Internets miteinander verbunden sind.

Fazit

In der Visualisierung von Software mit Hilfe von Metaphern wie der Software-Stadt sehen wir ein großes Potential, wenn es darum geht, mehrere Menschen das gleiche Bild zu verschaffen, über das sie sich kooperativ austauschen wollen. Mit Hilfe moderner VR/AR-Technologie aber genauso mit klassischen Desktop-Geräten können räumliche Distanzen überwunden werden. Inwieweit hiermit Entwicklerinnen und Entwickler erfolgreich unterstützt werden können, muss empirisch untersucht werden. Wir entwickeln unsere Plattform, um genau solche Studien möglich zu machen. Dabei genügt es aus unserer Sicht nicht, lediglich einfache Usability-Studien oder kontrollierte Experimente durchzuführen, bei denen Probanden in einer Laborumgebung nach einem kurzen Training bestimmte Aufgaben an einer ihnen

eigentlich unbekannten Software erledigen sollen, wie es häufig in der wissenschaftlichen Forschung praktiziert wird. Es braucht vermutlich zum einen eine etwas längere Einarbeitungszeit, bis man ein Visualisierungswerkzeug wirklich verinnerlicht hat, insbesondere wenn es um VR und AR geht (regelmäßige Computer-Spieler einmal ausgenommen). Zum anderen gibt das Betrachten einer fremden Software die Realität nur unzureichend wieder. Entwickler haben selbstverständlich bereits viel Vorwissen über ihre eigene Software. Ob die von uns angedachten Ideen und Konzepte funktionieren, muss somit in der Praxis am eigenen Code über einen längeren Zeitraum untersucht werden. Dabei werden sicherlich Unzulänglichkeiten sichtbar und neue Wünsche der Nutzer aufkommen, so dass die Visualisierungen stetig weiterentwickelt werden müssen. Eine Forschungsmethodik, die genau dies als integralen Bestandteil des Vorgehens enthält, ist die so genannte Aktions-Forschung. Derer wollen wir uns bedienen und in enger Kooperation mit Partnern aus der Industrie unsere Forschung gemeinsam vorantreiben. Im besten Falle gelingt es uns auf diese Weise, Forschungsergebnisse möglichst rasch in die Praxis der Software-Entwicklung einfließen zu lassen.

Literatur

[1] Koschke, Rainer; Simon, Daniel (2003): *Hierarchical Reflexion Models*. In: Proceedings of the IEEE Working Conference on Reverse Engineering. S. 36-45.

[2] Koschke, Rainer; Steinbeck, Marcel (2020): *Code Cities in Virtual and Augmented Reality*. In: Tagungsband des GI-Workshops Software-Reengineering und -Evolution (WSRE), S. 43-46. https://fg-sre.gi.de/fileadmin/FG/SRE/wsre2020/WSRE2020_Proceedings.pdf.

[3] Koschke, Rainer (2013). *Incremental Reflexion Analysis*. Journal of Software: Evolution and Process; 25(6):601–638.

Über den Autor

Prof. Dr. Koschke ist ein international aktiver Forscher in den Bereichen Software-Qualität, Architektur und Programmanalyse und Lehrender im Bereich Softwaretechnik an der Universität Bremen. Er ist Mitgründer der Firma Axivion GmbH, die Lösungen zur Bekämpfung von Software-Erosion bereithält. koschke@uni-bremen.de