WARNING	Bei diesem Skript handelt es sich um ein mit AI generiertes Skript. Basis hierfür ist das Transkript zur Videoeinheit sowie die Folien der Präsentation. Das Skript dient als Grundlage für die Erstellung von Lernkarten und Zusammenfassungen. Es ist nicht als vollständiger Ersatz für die Videoeinheit zu verstehen. Insbesondere ist es nicht fehlerfrei und sollte in Kombination mit der Videoeinheit verwendet werden. Fehler können gerne in Teams gemeldet werden oder in GitLab behoben werden.

[page 001] | *page_001.png*

Kapitel 01: Einführung in die Softwarearchitektur

In diesem Kapitel wird die Einführung in die Vorlesung zur Softwarearchitektur behandelt. Die Inhalte wurden bereits teilweise in der Präsenzveranstaltung besprochen und werden hier abschließend zusammengefasst und vertieft.

Zunächst wird das Thema softwareintensive Systeme betrachtet, also Systeme, bei denen Software eine zentrale Rolle spielt und die den Bedarf an einer durchdachten Softwarearchitektur begründen. Die wesentlichen Inhalte hierzu wurden bereits in der Präsenzveranstaltung vermittelt und werden hier nur kurz rekapituliert.

Im Anschluss wird der Fokus auf die Softwarearchitektur selbst gelegt. Es wird erläutert, was unter Softwarearchitektur zu verstehen ist und wie sie definiert wird. Darauf aufbauend werden die Methodiken für den Entwurf von Softwarearchitekturen vorgestellt. Zudem wird die Einbettung der Softwarearchitektur in den gesamten Softwareentwicklungsprozess betrachtet, um ihre Rolle und Bedeutung im Kontext der Softwareentwicklung zu verdeutlichen.

Ein weiterer Schwerpunkt liegt auf der Rolle des Softwarearchitekten. Es wird untersucht, welche Fähigkeiten ein Architekt benötigt und welche Aufgaben er im Entwicklungsprozess übernimmt.

Abschließend wird ein Überblick über die Historie der Softwarearchitektur gegeben. Obwohl dieser Teil nicht prüfungsrelevant ist, bietet er interessante Einblicke in die Entwicklung und die Meilensteine der Softwarearchitektur.

[page 002] | *page_002.png*

[page 003] | *page_003.png*

Zunächst wird auch auf die verschiedenen Arten von softwareintensiven Systemen eingegangen, um ein umfassendes Verständnis für die Bedeutung und Vielfalt dieser Systeme zu vermitteln. Es wurde kurz auf Spiele eingegangen, die ebenfalls als Software betrachtet werden können. Darüber hinaus gibt es extremere Echtzeitsysteme, wie beispielsweise Wettsysteme, sowie eingebettete Systeme, die sehr hardwarenah arbeiten. Eingebettete Systeme finden sich häufig in mobilen Anwendungen, etwa in Fahrzeugen oder anderen tragbaren Geräten. Zusätzlich gibt es Informationssysteme, die hier in drei Archetypen zusammengefasst werden.

[page 004] | *page_004.png*

Eingebettete Systeme zeichnen sich dadurch aus, dass sie eng mit der Hardware verbunden sind. Diese enge Verknüpfung führt dazu, dass sie oft spezielle Anforderungen erfüllen müssen, wie etwa

einen sparsamen Einsatz von Rechenressourcen. Zudem sind sie häufig sicherheitskritisch und müssen eine hohe Zuverlässigkeit gewährleisten. Ein weiteres Merkmal ist ihre oft lange Lebensdauer, da sie in Maschinen oder Fahrzeugen integriert sind, die über Jahrzehnte hinweg im Einsatz bleiben.

Von diesen eingebetteten Systemen sind Echtzeitsysteme abzugrenzen, auch wenn es in der Praxis Überschneidungen geben kann. Echtzeitsysteme können beispielsweise auch E-Commerce-Systeme umfassen, die kontinuierlich online verfügbar sein müssen. Solche Systeme müssen in der Lage sein, eine hohe Anzahl von Nutzern gleichzeitig zu bedienen und dabei strenge Anforderungen an die Echtzeitfähigkeit zu erfüllen.

Abschließend sind Informationssysteme zu betrachten, die primär dazu dienen, die Organisation eines Unternehmens effizienter zu unterstützen. Sie ermöglichen es, die richtigen Informationen zur richtigen Zeit an die richtigen Personen weiterzugeben, um Geschäftsprozesse zu optimieren. Häufig handelt es sich dabei um Expertensysteme, die von Fachkräften genutzt werden, die mit diesen Systemen ihre spezifischen Aufgaben effizienter erledigen können.

[page 005] | *page_005.png*

Ein Beispiel für die enge Verzahnung von Software- und Hardwarearchitektur lässt sich in der Entwicklung moderner Fahrzeugarchitekturen finden. Hier zeigt sich ein Trend hin zu zentralisierten Computersystemen, die zunehmend leistungsfähiger werden. Diese zentralen Systeme sind mit verschiedenen Zonen innerhalb des Fahrzeugs verbunden, was verdeutlicht, wie stark Software- und Hardwarearchitektur in solchen Anwendungen gemeinsam gedacht und entwickelt werden müssen.

[page 006] | *page_006.png*

Ein weiteres Beispiel für ein Echtzeitsystem ist ein E-Commerce-System wie das von Amazon. In solchen Systemen ist die Echtzeitfähigkeit von entscheidender Bedeutung, da Verzögerungen oder Ausfälle unmittelbar zu erheblichen finanziellen Verlusten führen können.

[page 007] | *page_007.png*

Als Beispiel für ein Informationssystem kann das SAP-System Ariba genannt werden. Dieses System ist speziell darauf ausgelegt, die Verwaltung von Zulieferern zu unterstützen und somit die Effizienz und Transparenz in der Lieferkette eines Unternehmens zu erhöhen.

Die Beschaffung von Materialien für die Produktion sowie die Verwaltung von Dienstleistungen können über solche Systeme effizient organisiert werden. Dies korrespondiert gut mit einem Beispielsystem, das im weiteren Verlauf häufiger thematisiert wird: dem ERK-System (Einkaufs- und Reisekapazitätensystem), welches bereits kurz vorgestellt wurde. Es dient als exemplarisches System, um verschiedene Aspekte der Softwarearchitektur zu veranschaulichen – sowohl methodische Ansätze als auch technische Architekturen, die in diesem Kontext eine Rolle spielen.

Die technischen Architekturen, die in solchen Systemen zum Einsatz kommen, sind oft auch auf andere Systemarten übertragbar. Beispielsweise verfügen viele Systeme über ein Frontend, wie bereits am Beispiel von Amazon erläutert. Allerdings gibt es in spezifischen Anwendungsbereichen, wie etwa in Fahrzeugen, besondere Anforderungen. Im Automobilbereich wird das Frontend häufig in Form eines Head-up-Displays dargestellt, was spezielle technische und gestalterische

Ansätze erfordert. Trotz dieser Unterschiede lassen sich viele der hier vorgestellten Konzepte und Erkenntnisse auch auf andere Systemtypen, wie Informationssysteme, übertragen.

[page 009] | *page_009.png*

Im Detail wurde bereits erläutert, dass solche Systeme dazu dienen, Prozesse zu unterstützen, wie sie beispielsweise anhand der Porter'schen Wertschöpfungskette dargestellt werden. Dabei werden sowohl die primären Kernprozesse, die direkt zur Wertschöpfung und somit zur Generierung von Einnahmen beitragen, als auch unterstützende Prozesse, wie etwa der Einkauf, abgedeckt. Diese Unterstützung erfolgt durch die Bereitstellung der richtigen Informationen zur richtigen Zeit, was eine effiziente und effektive Arbeitsweise ermöglicht. Dies führt entweder zu Kosteneinsparungen oder zu einer Steigerung der Produktivität, was letztlich das Ziel verfolgt, den Gewinn des Unternehmens zu maximieren.

[page 010] | *page_010.png*

Im weiteren Verlauf wird das ERK-System (Einkaufs- und Reisekapazitätensystem) als zentrales Beispiel herangezogen. Dieses System ist speziell für die Anforderungen eines Reiseveranstalters konzipiert und dient der Unterstützung des Einkaufsprozesses. Die gesamte Wertschöpfungskette eines Touristikunternehmens wird dabei betrachtet – beginnend bei der Planung, welche Reiseangebote gemacht werden sollen, bis hin zur Steuerung des Einkaufs, bei dem es darum geht, Kapazitäten wie Hotels oder andere Dienstleistungen zu sichern. Dieses System wird im Verlauf der weiteren Kapitel immer wieder thematisiert, um exemplarisch aufzuzeigen, wie eine Softwarearchitektur für ein solches Szenario gestaltet und entworfen werden kann.

Das zuvor genannte Beispiel ist lediglich ein fiktives Szenario und dient der Veranschaulichung. Es erhebt keinen Anspruch auf fachliche Korrektheit. In den bereitgestellten PDF-Unterlagen finden sich ergänzende Folien, die weitere Anforderungen und Beispiele enthalten. Diese sollen verdeutlichen, wie viel Logik und wie viele Regeln in solchen Systemen berücksichtigt werden müssen. Für die nachfolgenden Beispiele sind diese Details jedoch nicht von zentraler Bedeutung, weshalb sie an dieser Stelle nicht weiter ausgeführt werden.

[page 011] | *page_011.png*

[page 012] | *page_012.png*

[page 013] | *page_013.png*

[page 014] | *page_014.png*

Wesentlich ist jedoch die Frage, warum softwareintensive Systeme, unabhängig davon, ob es sich um Informationssysteme oder andere Arten von Systemen handelt, eine durchdachte Architektur oder Softwarearchitektur benötigen. Hierbei ist entscheidend, dass die Architektur darauf abzielt, die gestellten Anforderungen zu erfüllen. Diese Anforderungen lassen sich in funktionale und nicht-funktionale Anforderungen unterteilen. Während funktionale Anforderungen beschreiben, welche Funktionen ein System bereitstellen soll, liegt der Schwerpunkt der Softwarearchitektur häufig auf den nicht-funktionalen Anforderungen. Diese definieren, wie ein System arbeiten soll, beispielsweise in Bezug auf Performance, Skalierbarkeit, Zuverlässigkeit oder Wartbarkeit.

[page 016] | *page_016.png*

Es ist theoretisch möglich, ein System zu entwickeln, das die geforderten Funktionen erfüllt, selbst wenn der zugrunde liegende Code unstrukturiert und schwer wartbar ist, wie es bei sogenanntem „Spaghetti-Code“ der Fall wäre. Solche Ansätze sind jedoch in der Praxis nicht nachhaltig, da sie langfristig zu Problemen bei der Wartung und Weiterentwicklung des Systems führen können. Allerdings wäre ein solches System nicht wartbar. Wartbarkeit ist jedoch eine der zentralen nicht-funktionalen Anforderungen, die in der Softwareentwicklung häufig anzutreffen sind. Wenn diese Anforderung nicht erfüllt wird, führt dies langfristig zu erheblichen Problemen bei der Pflege und Weiterentwicklung des Systems. Dennoch ist es möglich, dass ein System mit unstrukturiertem Code, wie beispielsweise „Spaghetti-Code“, die funktionalen Anforderungen erfüllt, auch wenn dies mit erheblichen Schwierigkeiten verbunden ist. Wartbarkeit bedeutet unter anderem, dass der Code leicht anpassbar und erweiterbar sein sollte.

Der Fokus der Softwarearchitektur liegt jedoch insbesondere darauf, die nicht-funktionalen Anforderungen zu erfüllen. Je nach Typ des Systems und den spezifischen Anforderungen, die an softwareintensive Systeme gestellt werden, können sich diese Anforderungen stark unterscheiden. Aus diesem Grund werden verschiedene Systemtypen unterschieden, und es entstehen oft unterschiedliche Softwarearchitekturen, die auf die jeweiligen Anforderungen zugeschnitten sind.

[page 017] | *page_017.png*

In dieser Vorlesung sollen die grundlegenden Prinzipien, Heuristiken und Methoden vermittelt werden, die bei der Entwicklung von Softwarearchitekturen angewendet werden können. Dies wird am Beispiel eines Informationssystems veranschaulicht. Nachdem bisher die Gründe für die Notwendigkeit von Softwarearchitektur sowie deren Zielsetzung thematisiert wurden, wird im nächsten Kapitel genauer darauf eingegangen, was Softwarearchitektur eigentlich ist. Zunächst ist festzuhalten, dass der Begriff „Architektur“ im Alltag häufig verwendet wird und seinen Ursprung in der räumlichen Architektur hat. Laut der Definition, wie sie beispielsweise in Wikipedia zu finden ist, bezieht sich Architektur auf die handwerkliche und ästhetische Auseinandersetzung des Menschen mit dem gebauten Raum.

[page 018] | *page_018.png*

Diese Analogie wurde bewusst gewählt, da sich viele der grundlegenden Prinzipien der räumlichen Architektur auf die Softwarearchitektur übertragen lassen. Ein zentraler Gedanke ist dabei, dass Architektur entworfen wird, um spezifische Anforderungen zu erfüllen. Diese Idee findet sich sowohl in der räumlichen als auch in der Softwarearchitektur wieder. So gibt es beispielsweise Architekturen, die der Verteidigung dienen, wie etwa der Bau von Mauern in früheren Zeiten. Ebenso existieren Architekturen zur Organisation, etwa bei der Planung und Strukturierung komplexer Straßensysteme.

Ein weiteres Beispiel ist die Architektur für Mobilität, wie sie etwa bei Konzepten wie dem „Turning House Movement“ zu finden ist. Dabei handelt es sich um die Idee, ein Haus auf Rädern zu konzipieren, das flexibel an verschiedene Standorte verlegt werden kann. Solche Anforderungen erfordern eine spezifische und durchdachte Konzeption, die auf die jeweiligen Bedürfnisse und Rahmenbedingungen abgestimmt ist.

[page 019] | *page_019.png*

Wie ein System letztendlich aufgebaut wird, hängt von den spezifischen Anforderungen ab. Genau

aus diesem Grund existieren unterschiedliche Architekturen, die auf verschiedene Anforderungen zugeschnitten sind. Überträgt man diese Überlegungen auf die Softwarearchitektur, stellt sich die Frage, wie eine solche Architektur konkret gestaltet sein sollte. Dabei ist es offensichtlich, dass eine naive Herangehensweise, bei der man lediglich eine große Blackbox definiert und diese als das gesamte System betrachtet, nicht zielführend ist. Ein solches Vorgehen würde den Anforderungen an eine durchdachte Softwarearchitektur nicht gerecht werden.

Ein Beispiel hierfür wurde bereits im Zusammenhang mit den Windkraftanlagen betrachtet. Es ist essenziell, die interne Struktur eines Systems zu analysieren und darzustellen, wie die verschiedenen Strukturelemente miteinander interagieren. Die Softwarearchitektur sollte also nicht nur die äußere Hülle eines Systems beschreiben, sondern auch die inneren Zusammenhänge und die Organisation der einzelnen Komponenten.

[page 020] | *page_020.png*

Eine mögliche Definition von Softwarearchitektur könnte daher wie folgt lauten: Softwarearchitektur umfasst die grundlegenden Prinzipien und Designentscheidungen eines Systems. Sie beschreibt, welche Prinzipien dem Entwurf zugrunde liegen und welche Entscheidungen auf Basis dieser Prinzipien getroffen und dokumentiert wurden. Insbesondere bei Informationssystemen zeigt sich die Softwarearchitektur in der Konzeption von Komponenten, Bausteinen oder Services, die zusammen das Gesamtsystem bilden. Strukturelemente dienen dazu, die Blackbox eines Systems aufzubrechen und die Komplexität zu reduzieren. Nach dem Prinzip des „Divide and Conquer“ werden diese Elemente in kleinere, handhabbare Einheiten zerlegt, die jedoch miteinander interagieren müssen. Um diese Interaktionen zu ermöglichen, sind klar definierte Schnittstellen zwischen den Elementen erforderlich, ebenso wie eine präzise Beschreibung der Abhängigkeiten zwischen Bausteinen und Schnittstellen. Diese Struktur kann in verschiedenen Sichten dargestellt werden, ähnlich wie in der räumlichen Architektur, bei der Baupläne oder ähnliche Darstellungen verwendet werden.

[page 021] | *page_021.png*

Ein bewährter Leitfaden zur Dokumentation von Softwarearchitektur ist das Modell „arc42“, das bereits in einem vorherigen Kapitel behandelt wurde. Die zugehörige Dokumentation ist über den in der vorherigen Präsentation angegebenen Link zugänglich. Dieses Modell listet die wesentlichen Elemente auf, die in einer Softwarearchitektur dokumentiert werden sollten, und bietet eine einheitliche Taxonomie. Diese ermöglicht es, die Architektur eines Systems schnell und effizient zu verstehen.

Ein zentraler Aspekt der Softwarearchitektur, wie bereits in der Definition erwähnt, ist die Betrachtung der Architektur aus verschiedenen Perspektiven und Sichten. Die Idee der verschiedenen Sichten lässt sich an dieser Stelle gut veranschaulichen. Grundsätzlich geht es darum, ein komplexes Konstrukt – wie Software oder auch die Softwarearchitektur selbst – aus unterschiedlichen Perspektiven darzustellen. Diese Perspektiven sind jeweils für sich genommen einfacher zu verstehen.

[page 022] | *page_022.png*

Das Konzept kann mit dem auf der Folie dargestellten Bild verglichen werden: Ein komplexes Gebilde in der Mitte ist auf den ersten Blick schwer zu erkennen. Beleuchtet man es jedoch von

verschiedenen Seiten, werden klare Strukturen sichtbar. Genau dies ist der Ansatz der Sichten: Bestimmte Bereiche eines Systems gezielt und detailliert zu betrachten.

Dieses Vorgehen entspricht einem grundlegenden Architekturprinzip, das später noch genauer behandelt wird – dem Prinzip der Trennung von Zuständigkeiten. Jede Sicht sollte sich auf einen spezifischen Aspekt konzentrieren und keine überlappenden Inhalte mit anderen Sichten aufweisen. Andernfalls würde die Komplexität erneut zunehmen, da man versuchen müsste, mehrere Aspekte aus einer einzigen Darstellung zu extrahieren und zu verstehen. Ziel ist es, durch die Betrachtung aus unterschiedlichen Perspektiven eine klare und strukturierte Sicht auf das Gesamtsystem zu ermöglichen. Für unterschiedliche Stakeholder sind verschiedene Aspekte von Bedeutung, die durch die vier zentralen Sichten abgedeckt werden. Diese Sichten werden im Folgenden näher betrachtet. Es ist wichtig, sich diese Sichten einzuprägen, da sie eine wesentliche Rolle bei der Strukturierung und Darstellung eines Systems spielen.

[page 023] | *page_023.png*

Ein Vergleich mit der klassischen Architektur verdeutlicht die Notwendigkeit solcher Sichten: Wenn man ein Gebäude entwerfen möchte, würde man nicht einfach eine grobe Skizze des Hauses anfertigen und den Handwerkern überlassen, ohne detaillierte Pläne bereitzustellen. Aus einer solchen Skizze ließen sich weder die Position der Wände noch die Anordnung von Fenstern und Türen ableiten. Stattdessen sind präzise Ansichten erforderlich, die beispielsweise die genaue Platzierung von Wänden, Fenstern und Türen sowie die Verankerung des Gebäudes im Boden darstellen. Ebenso ist es in der Softwarearchitektur notwendig, verschiedene Aspekte eines Systems strukturiert und detailliert darzustellen, um eine klare und umsetzbare Grundlage für die Umsetzung zu schaffen. Die Elektrotechnik eines Hauses, einschließlich der gesamten Verkabelung, muss ebenfalls separat dargestellt werden. Dies ist ein weiterer Aspekt, der in der klassischen Architektur berücksichtigt wird. Letztlich basiert die Softwarearchitektur auf einer ähnlichen Idee. Auch hier werden verschiedene Sichten verwendet, um die unterschiedlichen Aspekte eines Systems zu strukturieren und darzustellen.

[page 024] | *page_024.png*

Diese vier zentralen Sichten werden oft durch ikonische Darstellungen veranschaulicht, wie sie beispielsweise im Buch „Effektive Softwarearchitektur“ zu finden sind. Dazu gehören die Kontextsicht, die das System als Blackbox beschreibt, die Bausteinsicht, die die Strukturierung und das Zusammenspiel der Bausteine zeigt, sowie die Verteilungssicht, die darstellt, wo die Bausteine bereitgestellt und betrieben werden.

[page 025] | *page_025.png*

Um den Vergleich zwischen klassischer Architektur und Softwarearchitektur weiter zu vertiefen, lässt sich eine weitere interessante Parallele ziehen. In der klassischen Architektur gibt es oft ikonische Bauwerke, die als Beispiele für herausragendes Design und Ingenieurskunst dienen. Zu diesen zählen etwa die Oper von Sydney oder die Elbphilharmonie in Hamburg. Häufig wird in der klassischen Architektur Wert darauf gelegt, dass Bauwerke nicht nur funktional, sondern auch ästhetisch ansprechend, modern und zeitgemäß sind. Sie sollen etwas Wertvolles und Beständiges darstellen. In der Softwarearchitektur ist der Ansatz in gewisser Weise anders, doch auch hier gibt es Parallelen zur klassischen Architektur. Ein anschauliches Beispiel aus der klassischen Architektur ist der Bauhaus-Stil. Dieser Architekturstil legt den Fokus auf die Funktionalität eines

Gebäudes – wie im Fall der Bauhaus-Schule – und strebt an, diese so effektiv und effizient wie möglich umzusetzen, während gleichzeitig durch Schlichtheit eine gewisse Ästhetik bewahrt wird.

Dieses Prinzip des Bauhaus-Stils spiegelt die Grundidee der Softwarearchitektur wider. Softwarearchitektur sollte eine klare und verständliche Struktur bieten, die durch ihre Einfachheit Eleganz ausstrahlt. Dabei geht es nicht um aufwendige oder verspielte Elemente, wie sie beispielsweise bei ikonischen Bauwerken wie der Oper von Sydney zu finden sind, sondern vielmehr um die Konzentration auf das Wesentliche. Der Leitsatz „Weniger ist mehr“ ist hier durchaus passend. Die Softwarearchitektur sollte sich klar auf ihren Zweck ausrichten, nämlich die Anforderungen des jeweiligen Systems präzise und effizient abzubilden. Es ist unwahrscheinlich, dass ein Anwender oder Nutzer einer Software die Anforderung stellt, dass Architekturdokumente oder die Architektur selbst besonders ästhetisch oder ausgefallen gestaltet sein sollen. Für die tatsächlichen Nutzer der Software, die diese im Alltag verwenden, spielt das keine Rolle, da sie die Architektur in der Regel nicht direkt wahrnehmen. Die Softwarearchitektur ist vielmehr ein Hintergrundelement, das für die Funktionalität und Struktur der Software verantwortlich ist. Dies verdeutlicht einen wesentlichen Unterschied: In der Softwarearchitektur liegt der Fokus nicht auf der äußeren Erscheinung, sondern auf der Erfüllung der funktionalen Anforderungen.

Ein zentraler Aspekt der Softwarearchitektur ist daher die Beschränkung auf das Wesentliche, also auf die Anforderungen, die tatsächlich relevant sind. In der Praxis kommt es jedoch vor, dass Architekten übermäßig komplexe oder abstrakte Konstrukte entwerfen, die zwar für Fachleute beeindruckend wirken mögen, aber den eigentlichen Zweck der Software nicht erfüllen. Solche übertriebenen oder unnötig komplizierten Elemente, die oft als „goldene Henkel“ bezeichnet werden, können die Software unnötig verkomplizieren und ihre Effizienz beeinträchtigen.

[page 026] | *page_026.png*

Abschließend sei noch auf einige formale Definitionen der Softwarearchitektur hingewiesen, die immer wiederkehrende Elemente enthalten. Eine prägnante Definition stammt beispielsweise von Taylor: Softwarearchitektur ist die Menge der grundlegenden Designentscheidungen, die für ein Softwaresystem getroffen wurden. Eine weitere Definition stammt von Reusner, der die Softwarearchitektur als die Organisation eines Systems beschreibt. Dabei betont er insbesondere die Aufteilung in Komponenten und die Organisation der Abhängigkeiten und Beziehungen zwischen diesen Komponenten. Auch die Umgebung des Systems spielt eine wichtige Rolle, was die Bedeutung der Kontextsicht unterstreicht. Zudem hebt Reusner die Relevanz von Prinzipien hervor, die bei der Gestaltung der Architektur berücksichtigt werden sollten.

Darüber hinaus gibt es weitere Definitionen, die versuchen, die Essenz der Softwarearchitektur zu erfassen. Neben den formalen Definitionen existieren auch prägnante, eingängige Formulierungen, die bestimmte Aspekte der Architektur betonen, jedoch nicht immer die gesamte Komplexität und Tragweite des Themas abbilden.

[page 027] | *page_027.png*

Ein Beispiel hierfür ist die Definition von Perry und Wolf: „Softwarearchitektur ist das, was gebaut werden soll, wie es gebaut werden soll und warum.“ Besonders der Aspekt des „Warum“ ist von Bedeutung, da er die Anforderungen und die dahinterliegenden Beweggründe beleuchtet. Eine weitere Definition beschreibt die Softwarearchitektur als „die durch jene Aspekte bestimmte Architektur der implementierten Software, die schwer zu ändern sind“. Diese Formulierung hebt

die langfristige Bedeutung von grundlegenden Designentscheidungen hervor, die die Flexibilität und Anpassungsfähigkeit eines Systems maßgeblich beeinflussen. Ein interessanter Aspekt, der in dieser Definition hervorgehoben wird, ist die Betonung darauf, dass die Softwarearchitektur die grundlegenden Entscheidungen eines Systems umfasst, die oft nur schwer zu revidieren sind. Dies unterscheidet sich von anderen formalen Definitionen, da hier explizit auf die Tragweite und die langfristigen Konsequenzen dieser Entscheidungen eingegangen wird. Fehler in der Softwarearchitektur können später erhebliche Probleme verursachen, da grundlegende Entscheidungen nicht ohne Weiteres rückgängig gemacht oder angepasst werden können.

Ein Vergleich mit der klassischen Architektur verdeutlicht dies: Wenn tragende Elemente, wie beispielsweise Stahlpfeiler in einem Gebäude, falsch positioniert oder dimensioniert werden, kann dies die Stabilität des gesamten Bauwerks gefährden. Ähnlich verhält es sich mit bestimmten Entscheidungen in der Softwarearchitektur, die als tragende Säulen des Systems betrachtet werden können. Diese Aspekte werden im weiteren Verlauf noch detaillierter behandelt, insbesondere im Hinblick auf die verschiedenen Arten von Entscheidungen, die in diesem Kontext eine Rolle spielen.

Ein weiterer Punkt ist die Definition von Ralph Johnson, die den Begriff der Softwarearchitektur auf das Wesentliche reduziert: „The important stuff, whatever that is.“ Diese Formulierung hebt hervor, dass die Architektur die entscheidenden Aspekte eines Systems umfasst, wobei die genaue Bedeutung von „wichtig“ vom jeweiligen Kontext abhängt. Eine weitere prägnante Definition beschreibt die Softwarearchitektur als „die Menge der Entwurfsentscheidungen, deren falsche Umsetzung das Projekt scheitern lassen kann“. Auch diese Perspektive unterstreicht die zentrale Bedeutung sorgfältig getroffener Entscheidungen für den Erfolg eines Projekts.

[page 028] | *page_028.png*

Eine Abgrenzung der Softwarearchitektur zu anderen Architekturen oder Architekturbegriffen in der IT ist hilfreich, um die unterschiedlichen Ebenen und deren jeweilige Reichweite zu verstehen. Diese Unterscheidung erfolgt insbesondere anhand des Grades der Globalität, auf dem man sich bewegt, und der Anzahl der betroffenen Personen innerhalb einer Organisation, die an der Softwareentwicklung beteiligt sind. Je globaler die Ebene, desto weniger granular sind die betrachteten Aspekte, und desto mehr Personen und Systeme können potenziell betroffen sein.

Auf der höchsten Ebene befindet sich die sogenannte Enterprise-Architektur. Diese befasst sich mit der gesamten Anwendungslandschaft eines Unternehmens, also der Gesamtheit aller eingesetzten Anwendungen. In großen Konzernen kann diese Anwendungslandschaft aus mehreren Tausend Systemen bestehen. Häufig gibt es dabei Überschneidungen oder Redundanzen, das heißt, es existieren mehrere Systeme, die dieselben Aufgaben erfüllen. Durch Konsolidierung und die Zusammenführung solcher Systeme auf ein führendes System können erhebliche Einsparungen erzielt werden.

Ein Beispiel für die Arbeit von Enterprise-Architekten ist die Integration von Anwendungslandschaften nach einer Fusion oder Übernahme von Unternehmen. In solchen Fällen kann es vorkommen, dass die kombinierte Anwendungslandschaft aus mehreren Tausend Systemen besteht. Ziel ist es dann, diese Landschaft zu optimieren und zu konsolidieren, um die Effizienz zu steigern und Kosten zu senken. Eine Reduktion der Anwendungslandschaft von beispielsweise 1700 auf eine deutlich geringere Anzahl von Systemen kann zu erheblichen Kosteneinsparungen führen. Allerdings ist dies auch mit Herausforderungen verbunden,

insbesondere auf organisatorischer Ebene. Es erfordert, dass viele Mitarbeitende, die an die Nutzung eines bestimmten Systems gewöhnt sind, auf ein neues System umsteigen. Dieser Übergang gestaltet sich nicht immer einfach.

Um diesen Prozess erfolgreich zu gestalten, ist es notwendig, ein sogenanntes Change Management durchzuführen. Dabei handelt es sich um einen strukturierten Ansatz, der darauf abzielt, die betroffenen Mitarbeitenden durch den Veränderungsprozess zu begleiten. Es reicht nicht aus, den Nutzenden einfach ein neues System vorzugeben und zu erwarten, dass sie es sofort akzeptieren und nutzen. Vielmehr müssen sie aktiv in den Prozess eingebunden werden, um sicherzustellen, dass die Effizienz und die bestehenden Geschäftsprozesse erhalten bleiben.

Diese Überlegungen gehören zur Ebene der Enterprise-Architektur, die in dieser Vorlesung jedoch nicht weiter behandelt wird. Die Abgrenzung dient lediglich der Einordnung. Der Fokus dieser Veranstaltung liegt auf der Architektur eines einzelnen Systems, insbesondere eines Informationssystems, und darauf, wie ein solches System aufgebaut werden kann. Ein zentraler Bestandteil ist dabei die Konstruktion von Bausteinen innerhalb einer Softwarearchitektur. Es wird auch auf Design Patterns und Aspekte der Implementierungsarchitektur eingegangen. Gelegentlich werden dazu Codebeispiele herangezogen, wobei diese eher an der Grenze dessen liegen, was im Rahmen dieser Vorlesung behandelt wird. Dies entspricht im Wesentlichen den Inhalten, die bereits in vorherigen Vorlesungen behandelt wurden. Die Thematik der Enterprise-Architektur wird hier nur am Rande erwähnt. Am Ende wird ein kurzer Ausblick darauf gegeben, was sich hinter diesem Konzept verbirgt und welche Methodiken in diesem Bereich existieren. Damit ist der Begriff der Softwarearchitektur und dessen Bedeutung in diesem Kontext grob umrissen.

[page 029] | *page_029.png*

Die zentrale Frage, die sich nun stellt, lautet: Wie gelangt man zu einer vollständigen Softwarearchitektur? Das angestrebte Endergebnis sollte mittlerweile klarer sein. Anhand des zuvor gegebenen Beispiels, das sich an Standards wie dem Architekturstandard A42 orientiert, wurde verdeutlicht, dass das Resultat in Form einer umfassenden Dokumentation vorliegt. Diese enthält verschiedene Sichten auf das System sowie die zugrunde liegenden Designentscheidungen. Die entscheidende Frage ist jedoch, wie diese Entscheidungen getroffen werden, insbesondere in Bezug auf die Strukturierung und Abgrenzung eines Systems.

[page 030] | *page_030.png*

Hierbei ist es zunächst wichtig zu betonen, dass Architektinnen und Architekten auf Methoden zurückgreifen können, die eine systematischere Herangehensweise ermöglichen. Darüber hinaus können Heuristiken eine wertvolle Unterstützung bieten. Diese werden in einer separaten Veranstaltung detaillierter behandelt. Es ist jedoch essenziell, sich bewusst zu machen, dass es keine universelle Lösung oder „silberne Kugel“ gibt, die alle Herausforderungen der Softwarearchitektur auf einmal löst. Der Begriff der „silbernen Kugel“ ist ein metaphorisches Wortspiel, das ursprünglich aus der Mythologie stammt, insbesondere im Zusammenhang mit Werwölfen. Es bezeichnet eine universelle Lösung, die jedes Problem zuverlässig löst. In der Softwarearchitektur existiert jedoch keine solche „silberne Kugel“. Es gibt keinen standardisierten Fahrplan, der in wenigen einfachen Schritten direkt zu einer vollständigen und perfekten Architektur führt. Der Prozess ist vielmehr komplex und dynamisch.

Um eine tragfähige Softwarearchitektur zu entwickeln, sind sowohl fundierte Methoden als auch

Heuristiken erforderlich. Dabei spielt die Erfahrung der Architektinnen und Architekten eine entscheidende Rolle. Es ist jedoch wichtig zu beachten, dass jede entworfene Architektur auf einem bestimmten Wissensstand basiert, der niemals vollständig sein kann. Daher müssen Entscheidungen stets auf Basis des besten verfügbaren Wissens und mit einem gewissen Maß an Unsicherheit getroffen werden.

[page 031] | *page_031.png*

Gernot Starke, ein anerkannter Experte auf diesem Gebiet, unterteilt die Methoden zur Entwicklung einer Softwarearchitektur in vier größere Phasen. Die erste Phase, die zugleich die Grundlage für den gesamten Prozess bildet, besteht darin, ein möglichst umfassendes Verständnis der Anforderungen zu erlangen. Obwohl der Wissensstand nie vollständig sein kann, ist es essenziell, die Anforderungen so präzise wie möglich zu erfassen, um fundierte und nachhaltige Entscheidungen treffen zu können. Ein zentraler Aspekt der ersten Phase ist es, ein klares Verständnis darüber zu entwickeln, welche Leistungen das System erbringen soll, warum es diese erbringen soll und welche Rahmenbedingungen dabei zu berücksichtigen sind. Diese Rahmenbedingungen stellen gewissermaßen Leitplanken dar, die den Lösungsraum einschränken und nicht beliebig verändert werden können. Solche Einschränkungen treten in Unternehmen häufig auf und können sowohl technologischer als auch organisatorischer Natur sein. Diese Aspekte werden im weiteren Verlauf noch detaillierter erläutert.

Bereits in dieser Phase ist es sinnvoll, potenzielle Risiken zu identifizieren. Es gilt zu analysieren, welche Bereiche besondere Aufmerksamkeit erfordern, um diese Risiken frühzeitig zu minimieren und fundierte Entscheidungen zu treffen.

Die zweite Phase des Prozesses widmet sich dem Entwurf der Softwarearchitektur. Auf Basis des in der ersten Phase gewonnenen Verständnisses werden in dieser Phase die grundlegenden Strukturen und Bausteine des Systems entworfen. Dabei geht es nicht nur um die Definition der einzelnen Komponenten, sondern auch um die Entwicklung technischer und querschnittlicher Konzepte. Ziel ist es, Lösungsstrategien für spezifische Anforderungen zu erarbeiten, die im weiteren Verlauf des Projekts auftreten könnten. Diese Aspekte werden ebenfalls noch ausführlicher behandelt. Der Prozess der Softwarearchitektur ist keine lineare Abfolge von Schritten, sondern vielmehr ein iterativer und dynamischer Vorgang. Dies zeigt sich insbesondere daran, dass während der Arbeit an einer Phase häufig neue Fragen oder Erkenntnisse auftauchen, die eine Rückkopplung zu vorherigen Phasen erfordern. Beispielsweise können bei der Entwicklung der Architektur Fragen an den Fachbereich entstehen, die ein tiefergehendes Verständnis der Anforderungen notwendig machen. In solchen Fällen ist es erforderlich, die Anforderungen erneut zu klären und auf dieser Grundlage die Architektur weiterzuentwickeln. Dieser iterative Prozess erfordert Erfahrung und die Fähigkeit, potenzielle Unklarheiten zu erkennen und gezielt zu analysieren, um fundierte Entscheidungen treffen zu können.

In der abschließenden Phase des Prozesses wird die erarbeitete Architektur schließlich implementiert. Nachdem die grundlegenden Strukturen und Bausteine definiert wurden, erfolgt die Übergabe an die Entwicklungsteams. Häufig sind mehrere Teams beteiligt, wobei jedes Team für die Implementierung eines spezifischen Bausteins verantwortlich sein kann. In dieser Phase ist es essenziell, die Architektur klar zu kommunizieren und umfassend zu dokumentieren. Darüber hinaus ist es notwendig, die Umsetzung aktiv zu begleiten, um sicherzustellen, dass die entwickelten Lösungen den definierten Anforderungen und der geplanten Architektur entsprechen. Heutzutage wird zunehmend davon abgesehen, lediglich eine

Architekturdokumentation zu erstellen, diese an die Entwicklungsteams weiterzugeben und die Implementierung ohne weitere Begleitung zu überlassen. Zwar war ein solches Vorgehen in der Vergangenheit verbreitet und ist in einigen Unternehmen möglicherweise noch anzutreffen, jedoch hat sich gezeigt, dass es nicht besonders effizient ist. Es ist entscheidend, dass Architektinnen und Architekten aktiv in den Umsetzungsprozess eingebunden sind, um zu überprüfen, wie gut die entworfene Architektur in der Praxis funktioniert und wo möglicherweise Anpassungen erforderlich sind.

Dabei ist es wichtig, die Erkenntnisse der Entwicklerinnen und Entwickler während der Implementierung zu berücksichtigen, um die Architektur kontinuierlich zu evaluieren und gegebenenfalls anzupassen. Dies schließt die Überprüfung ein, ob die getroffenen Designentscheidungen weiterhin angemessen sind. Im Verlauf der Umsetzung nimmt das Wissen über das System, das Umfeld sowie den gesamten Lösungs- und Problemraum stetig zu. Nach mehreren Monaten kann es daher notwendig sein, bestimmte Entscheidungen zu überdenken oder anzupassen, um den veränderten Anforderungen und Erkenntnissen gerecht zu werden. Dieses iterative Vorgehen ermöglicht es, die Architektur flexibel und effektiv weiterzuentwickeln.

[page 032] | *page_032.png*

Ein weiteres Vorgehensmodell wird hier vorgestellt, da später noch auf Domain-driven Design (DDD) eingegangen wird. Auch für DDD existiert ein spezifisches Vorgehensmodell, das eine relativ agile Umsetzung dieser Methode ermöglicht. Dieses Modell ist öffentlich zugänglich, und die entsprechenden Quellen sind in den begleitenden Materialien angegeben. Die sogenannte „DDD Crew“ ist eine Community, die verschiedene hochwertige Ressourcen bereitstellt, um die Anwendung von Domain-driven Design zu unterstützen.

Domain-driven Design, kurz DDD, ist eine Methodik, die auf dem gleichnamigen Buch von Eric Evans basiert, das im Jahr 2004 veröffentlicht wurde. Obwohl das Buch bereits vor einiger Zeit erschienen ist, hat die Methodik in den letzten Jahren erheblich an Popularität gewonnen. Der Kernansatz von DDD, wie der Name andeutet, besteht darin, den Entwurf eines Systems an der Fachlichkeit auszurichten. Dabei geht es nicht nur darum, die Domäne eines Systems zu verstehen, sondern auch die Struktur und Logik der Fachlichkeit im Systemdesign widerzuspiegeln.

Ein zentraler Aspekt von DDD ist das tiefgehende Verständnis der Fachlichkeit. Dies beginnt mit der Analyse des Geschäftsmodells des Unternehmens, um die grundlegenden Anforderungen und Bedürfnisse zu identifizieren. Nur durch ein fundiertes Verständnis der fachlichen Zusammenhänge kann ein System entworfen werden, das den tatsächlichen Anforderungen gerecht wird. Die Benutzer des Systems werden zunächst analysiert, um mögliche Lösungen für ihre Anforderungen zu identifizieren. Dieser Prozess wird kollaborativ gestaltet, was bedeutet, dass verschiedene Stakeholder gemeinsam an der Entwicklung von Lösungen arbeiten. Dieser Ansatz wird im weiteren Verlauf auch praktisch erprobt. In dieser Phase geht es darum, die Anforderungen und Rahmenbedingungen des Systems zu klären, was eine Grundlage für die nächsten Schritte bildet.

Im Anschluss daran wird das System in einzelne Komponenten unterteilt, ein Prozess, der im Englischen als „Decomposition“ bezeichnet wird. Dabei wird untersucht, wie die einzelnen Teile des Systems miteinander verbunden und integriert werden können. Im Kontext von Domain-driven Design (DDD) gibt es für diesen Aspekt verschiedene Muster, die sich als besonders effektiv und tiefgreifend erwiesen haben. Diese Muster sind essenziell, um die Bausteine des Systems

sinnvoll zu verknüpfen.

Ein weiterer wichtiger Schritt ist die Organisation der entwickelten Komponenten. Nachdem die Bausteine verteilt und miteinander verbunden wurden, muss festgelegt werden, welches Team die Verantwortung für welchen Baustein übernimmt. In der Terminologie von DDD wird dies als „Bounded Context“ bezeichnet. Dieser Begriff wird später noch genauer erläutert. Ziel ist es, Strategien zu entwickeln, die es ermöglichen, die einzelnen Bausteine entsprechend ihrer Bedeutung und Relevanz im Gesamtsystem effizient umzusetzen. Im nächsten Schritt werden konkrete Rollen definiert, um festzulegen, wer welche Aufgaben übernimmt. Anschließend erfolgt die Implementierung des Codes. Dieser Prozess ist iterativ und kann in verschiedenen Zyklen ablaufen, bei denen es wichtig ist, auf Basis neuer Erkenntnisse auch Rückschritte in Kauf zu nehmen, um Entscheidungen und Methoden aus vorherigen Phasen zu überarbeiten und zu verfeinern.

[page 033] | *page_033.png*

Ein Beispiel für eine weitere Methodik ist von Capgemini gegeben. Diese Methodik, die als „ADAM“ (Agile Design Applied Method) bezeichnet wird, soll im Folgenden kurz vorgestellt werden, um eine weitere Herangehensweise an die agile Steuerung von Softwarearchitektur zu veranschaulichen. ADAM ist inzwischen auch publiziert worden, unter anderem durch Artikel im ID-Spektrum und in verschiedenen Fachvorträgen. Obwohl diese Methodik nicht prüfungsrelevant ist, könnte sie dennoch von Interesse sein, da sie zeigt, wie Unternehmen die Strukturierung und Gestaltung von Softwaredesigns angehen können.

[page 034] | *page_034.png*

Die ADAM-Methodik weist Ähnlichkeiten mit den zuvor besprochenen Ansätzen auf. Es handelt sich dabei jedoch nicht um einen starren Fahrplan, sondern vielmehr um einen Baukasten aus verschiedenen Methoden und Werkzeugen, die je nach Bedarf angewendet werden können. Die Herangehensweise ist auch hier relativ agil und iterativ, wobei der Fokus stark auf der zugrunde liegenden Fachlichkeit liegt. Dabei wird bewusst darauf verzichtet, technische Details zu stark vorzugeben, da diese als ein separater Bereich betrachtet werden. Das gesamte Konzept ist in sogenannte „Sophia Haupt Building Blocks“ organisiert, die in Form von Kreisen dargestellt werden. Um diese Kreise ist ein weiterer Kreis angeordnet, der die iterative und dynamische Natur des Prozesses symbolisiert. Dies verdeutlicht, dass es möglich ist, zwischen den verschiedenen Bereichen flexibel zu wechseln und Anpassungen vorzunehmen.

Die grundlegende Idee besteht darin, zunächst eine strategische Perspektive einzunehmen. Dabei wird analysiert, welche Vision hinter dem Produkt steht und warum es entwickelt werden soll. Auf dieser Basis wird eine Roadmap definiert, die sich primär mit dem sogenannten Problemraum beschäftigt. Der Problemraum umfasst die Identifikation des geschäftlichen Nutzens, den das System erfüllen soll, sowie die Klärung der Frage, welchen Zweck das System verfolgt.

Im nächsten Schritt erfolgt der Übergang in die sogenannte „Discover“-Phase. In dieser Phase wird untersucht, welches Produkt am besten geeignet ist, um die identifizierten Anforderungen zu erfüllen. Dabei wird versucht, die notwendigen Funktionalitäten zu definieren, den Umfang eines Minimum Viable Product (MVP) zu bestimmen und erste Backlog-Items abzuleiten. Zudem wird eine erste Vorstellung der Architektur des Systems entwickelt, die in dieser Phase als „intentional Architecture“ bezeichnet wird. Die Architektur, die zunächst beabsichtigt wird, umzusetzen, wird

als „Intentional Architecture“ bezeichnet. Diese unterscheidet sich von der Architektur, die in der nächsten Phase, der sogenannten „Deliver“-Phase, entsteht. In dieser Phase wird die Software iterativ, inkrementell und agil entwickelt, wobei kontinuierlich neue Erkenntnisse gewonnen werden. Diese neuen Erkenntnisse können dazu führen, dass die ursprünglich geplante Architektur angepasst wird.

Das Ergebnis dieses Prozesses ist die sogenannte „Emergent Architecture“. Diese stellt die endgültige Architektur dar, die den größtmöglichen Wert liefert und die Anforderungen bestmöglich erfüllt. Darüber hinaus gibt es eine weitere Phase, die in der Praxis seltener explizit berücksichtigt wird, jedoch in die Methodik aufgenommen wurde, um ihre Bedeutung hervorzuheben.

Bereits in der strategischen Phase, in der die Vision und der Zweck der Software definiert werden, wird auch darauf geachtet, wie der Erfolg der Software gemessen werden kann. Dies geschieht durch die Definition von Key Performance Indicators (KPIs), also Kennzahlen, die es ermöglichen, den Grad der Zielerreichung zu bewerten. Die letzte Phase, die zunehmend auch in der Praxis an Bedeutung gewinnt, befasst sich mit der Bewertung des Erfolgs der Software. Diese Bewertung kann auf verschiedenen Wegen erfolgen, beispielsweise durch die Analyse von Feedback oder durch die Untersuchung der Auswirkungen der Software auf die gesteuerten Geschäftsprozesse. Dabei wird geprüft, ob diese Prozesse durch die Software besser unterstützt werden, ob sie schneller, effektiver oder effizienter ablaufen. Ziel ist es, diese Aspekte messbar zu machen, auch direkt innerhalb der Software, um daraus Erkenntnisse zu gewinnen.

Diese Erkenntnisse können genutzt werden, um Schwachstellen zu identifizieren und entsprechende Anpassungen vorzunehmen. Dies kann sowohl die Architektur als auch die fachlichen Anforderungen betreffen. Ziel ist es, durch iterative Verbesserungen das volle Potenzial des Softwareprodukts auszuschöpfen.

[page 035] | *page_035.png*

Adam gliedert diesen Prozess in verschiedene Methodiken, die wiederum in mehrere Schritte unterteilt sind. Zunächst wird der Kontext, in dem sich die Software befindet, analysiert und verstanden. Auf dieser Grundlage wird eine klare Vision für die Software formuliert. Aus dieser Vision wird schließlich eine Roadmap, also ein konkreter Fahrplan, abgeleitet, der die weitere Entwicklung der Software strukturiert. In der Discover-Phase liegt der Fokus darauf, das Problem umfassend zu analysieren und ein tiefgehendes Verständnis dafür zu entwickeln. Ziel ist es, eine Lösungsidee zu erarbeiten, die der definierten Vision entspricht. Diese Idee wird anschließend weiter verfeinert und konkretisiert, um eine detaillierte Planung zu ermöglichen. Dabei wird die Roadmap weiter ausgearbeitet und präzisiert. Ein Ergebnis dieser Phase ist die sogenannte „Intentional Architecture“, die als gezielte Architektur die Grundlage für die weitere Entwicklung bildet.

In der Deliver-Phase wird der Lösungsraum erschlossen, indem die Software entwickelt wird. Dabei wird ein möglichst effizienter Entwicklungsprozess angestrebt, der auf den Prinzipien agiler Methoden basiert. Während der Entwicklung wird kontinuierlich gelernt und der Prozess entsprechend angepasst. Das Ergebnis dieser Phase ist die „Emergent Architecture“, die sich dynamisch aus den Anforderungen und Erkenntnissen der Entwicklung ergibt.

Abschließend folgt die Phase, in der Feedback gesammelt wird. Hierbei werden Metriken erhoben,

um wertvolle Erkenntnisse („Insights“) zu gewinnen. Diese dienen dazu, die Software weiter zu optimieren und die Architektur sowie die Funktionalität zu verbessern.

[page 036] | *page_036.png*

Im Rahmen des Ansatzes von ADAM wurden zudem verschiedene Artefakte definiert, die in den einzelnen Phasen entstehen. Diese Artefakte dokumentieren die Ergebnisse der jeweiligen Schritte und dienen als Grundlage für die weitere Arbeit. Es ist jedoch wichtig zu betonen, dass nicht alle dieser Artefakte direkt architekturrelevant sind. Dennoch spielt die Architektur eine zentrale Rolle, insbesondere im Lösungsraum. Hier geht es darum, Bausteine zu identifizieren, die spezifische Funktionen erfüllen, Informationen verarbeiten und gegebenenfalls über Benutzeroberflächen interagieren können. Im Kontext der Architekturentwicklung wird auch hier wieder der Ansatz verfolgt, Bausteine zu etablieren, die in den jeweiligen Kontext passen. Diese Bausteine können als Ergebnisse des Prozesses betrachtet werden, ohne dass sie zwangsläufig in einer bestimmten Form dokumentiert werden müssen. Es handelt sich zunächst um eine grobe Übersicht, die sich an einem übergeordneten Bild orientiert. Dieses Bild dient als Leitfaden, an dem sich der gesamte Prozess ausrichtet.

Der Entwicklungsprozess ist dabei stark iterativ und nicht strikt linear oder durch einen festen Plan vorgegeben. Dies spiegelt die Realität wider, in der es oft nicht möglich ist, von Anfang an eine endgültige Lösung zu definieren. Architekturarbeit ist in diesem Sinne eine kreative, aber auch anspruchsvolle Tätigkeit, da Entscheidungen getroffen werden müssen, obwohl nicht alle relevanten Informationen vorliegen. Beispielsweise ist es unmöglich, zukünftige Anforderungen oder neue funktionale Anforderungen, die im Laufe der Zeit entstehen könnten, vollständig vorherzusehen. Solche Unwägbarkeiten stellen eine zentrale Herausforderung in der Architekturentwicklung dar. Die Architektur wird maßgeblich von Faktoren beeinflusst, die zum Zeitpunkt der Planung möglicherweise noch nicht vollständig bekannt sind. Beispielsweise können Anforderungen aus dem Fachbereich, die sich erst später konkretisieren, einen erheblichen Einfluss auf die Architektur haben.

[page 037] | *page_037.png*

An dieser Stelle soll ein kurzer Exkurs zur Modellierung eingefügt werden, um die Bedeutung und die Möglichkeiten der Darstellung von Strukturen im Rahmen des Entwurfsprozesses zu beleuchten. Modellierung ist ein zentraler Aspekt, der es ermöglicht, die entworfenen Strukturen zu visualisieren und weiterzuentwickeln.

[page 038] | *page_038.png*

Ein Modell dient dabei als Abbild einer Struktur und stellt deren wesentliche Merkmale dar. Die Modellierung selbst ist die Tätigkeit, bei der solche Modelle erstellt werden. Häufig basieren Modellierungsmethoden auf spezifischen Systematiken, Techniken und teilweise auch auf formalen Theorien. Diese Methoden können je nach Anwendungsfall unterschiedlich komplex sein.

[page 039] | *page_039.png*

Der Hauptzweck der Modellierung liegt in der Visualisierung von Strukturen, um diese besser analysieren, kommunizieren und weiterentwickeln zu können. Darüber hinaus verfolgt jedes Modell einen bestimmten Zweck, der je nach Kontext variieren kann. Es gibt eine Vielzahl von Modellierungsansätzen, die je nach Anforderungen und Zielsetzung eingesetzt werden können. Ein

Ziel der Modellierung kann darin bestehen, Erkenntnisse über ein System zu gewinnen. Dies ist beispielsweise bei formalen Modellen im Bereich der Elektrotechnik der Fall, wenn Maschinen entworfen werden. Hierbei wird ein Modell erstellt, um zu analysieren, wie ein System funktioniert. Wenn das Modell beispielsweise in Form eines Petri-Netzes formuliert wird, können mithilfe spezifischer Techniken und Methoden zeitbasierte Analysen durchgeführt werden. Dadurch lässt sich beispielsweise überprüfen, ob das Petri-Netz in einen Deadlock-Zustand geraten könnte. Dies stellt ein typisches Anwendungsbeispiel für die Nutzung von Modellen dar.

Häufig dient die Modellierung jedoch der Erklärung und Veranschaulichung. Ziel ist es, darzustellen, wie ein System strukturiert ist. Gleichzeitig ermöglicht die Modellierung, das zugrunde liegende Originalsystem – in diesem Fall die Software – besser zu verstehen und zu beherrschen. Dies ist insbesondere wichtig, um die Komplexität der Software zu kontrollieren und zukünftige Änderungen oder Erweiterungen effizient vornehmen zu können.

[page 040] | *page_040.png*

Ein weiteres Ziel der Modellierung kann darin bestehen, eine Grundlage für Entscheidungen zu schaffen, beispielsweise durch die Abschätzung des erforderlichen Aufwands. Zudem können Modelle genutzt werden, um verschiedene Varianten eines Systems durchzuspielen und Optimierungsmöglichkeiten zu identifizieren, ohne diese direkt in der Praxis umsetzen zu müssen. Simulationen können ebenfalls ein Ziel der Modellierung sein, insbesondere bei technischen Systemen. Allerdings ist der Einsatz von Simulationen in der Softwareentwicklung weniger verbreitet und oft mit Herausforderungen verbunden. Es gab Versuche, Simulationen auf Basis von UML-Diagrammen oder ähnlichen Modellen durchzuführen, jedoch waren diese Ansätze nach aktuellem Wissensstand nur selten wirklich erfolgreich.

Ein weiteres Ziel der Modellierung kann die Unterstützung bei der Planung sein. Auf Grundlage eines Softwaremodells lassen sich Rückschlüsse darauf ziehen, welche Komponenten in welcher Reihenfolge am besten umgesetzt werden sollten. Dies erleichtert die Planung und Priorisierung der Entwicklungsarbeit.

Darüber hinaus spielt die Modellierung eine wichtige Rolle bei der Konstruktion. Sie bietet den Entwicklern eine Grundlage, um das Gesamtsystem besser zu verstehen und dieses Wissen bei der Implementierung zu berücksichtigen.

Schließlich kann die Modellierung auch zur Verifikation genutzt werden. Dabei wird überprüft, ob das Modell potenzielle Probleme aufzeigt, die sich negativ auf das System auswirken könnten. Ein Beispiel hierfür ist die Identifikation von Zyklen in Bausteinsichten, die die Wartbarkeit des Systems erschweren könnten. Ein weiteres Beispiel, das bereits erwähnt wurde, ist die Analyse von Deadlocks in Maschinenmodellen. Solche Verifikationen helfen, Schwachstellen frühzeitig zu erkennen und zu beheben. Die Steuerung und Kontrolle kann ebenfalls ein Ziel der Modellierung sein. Auf Basis eines Modells können Mechanismen entwickelt werden, um die Softwareentwicklung gezielt zu steuern und zu überwachen. In manchen Fällen kann ein Modell sogar als Ersatz für das Originalsystem dienen. Es gibt zahlreiche unterschiedliche Zwecke, die ein Modell erfüllen kann oder soll. In unserem Beispiel bezieht sich das Modell auf die Softwarearchitektur, die entworfen wird.

Ein solches Modell dient in erster Linie dazu, das Originalsystem zu erklären und dadurch ein besseres Verständnis zu schaffen. Wie diese Modelle verwendet werden, ist vielfältig. Sie können

beispielsweise deskriptiv eingesetzt werden, um ein System zu beschreiben, oder preskriptiv, indem sie als Vorlage für die Entwickler dienen, mit der Aufforderung, das System entsprechend umzusetzen.

Darüber hinaus können Modelle konzeptionell genutzt werden, um technische Konzepte zu erläutern und zu erklären. Sie können auch exemplarisch oder experimentell verwendet werden, um verschiedene Architekturentscheidungen zu skizzieren und deren Vor- und Nachteile durch Vergleiche zu analysieren. Es gibt eine Vielzahl weiterer Einsatzmöglichkeiten, die von der gestalterischen Nutzung bis hin zur Unterstützung bei der Planung reichen.

Insbesondere werden Architekturmodelle häufig deskriptiv eingesetzt, um bestehende Systeme zu beschreiben, oder preskriptiv, um Vorgaben für die Umsetzung zu machen. Ein weiterer zentraler Anwendungsfall ist die gestalterische Nutzung, bei der ein Plan für die Umsetzung entwickelt wird.

[page 041] | *page_041.png*

Betrachtet man die Ergebnisse von Architekturmodellen, so lassen sich verschiedene Qualitätseigenschaften identifizieren, die mit diesen einhergehen. Ein Modell kann beispielsweise anhand von Korrektheitskriterien bewertet werden: Wie genau oder glaubwürdig ist das Modell? Entspricht es dem Originalsystem? Kann das Modell zur Überprüfung des Originals herangezogen werden? Solche Fragen sind essenziell, um die Qualität eines Modells zu beurteilen. Allgemeine Qualitätskriterien umfassen zudem die Aktualität des Modells. Es ist entscheidend, dass ein Modell den aktuellen Zustand des Systems widerspiegelt. Veraltete Dokumentationen oder Architekturmodelle, die nicht mehr den tatsächlichen Gegebenheiten entsprechen, können zu Verwirrung führen und sind in der Praxis häufig anzutreffen, beispielsweise bei Architektur-Audits von Software. Daher ist es von großer Bedeutung, Modelle kontinuierlich zu pflegen und aktuell zu halten.

Ein weiterer zentraler Aspekt der Qualität von Architekturmodellen ist deren Nützlichkeit. Ein Modell sollte nicht nur um seiner selbst willen existieren, sondern einen klaren Zweck erfüllen. Es muss für die Entwickler relevant sein und ihnen als Orientierung dienen, um die Entwicklung effizienter zu gestalten und zu unterstützen. Ein Modell, das diesen Anforderungen nicht gerecht wird, verliert an Wert. Ebenso ist die Verständlichkeit eines Modells von großer Bedeutung. Ein unverständliches Modell wird in der Regel nicht genutzt, da es seinen Zweck, die Kommunikation und Unterstützung der Entwicklung, nicht erfüllen kann. Idealerweise sollte ein Architekturmodell auch neue Erkenntnisse liefern. Dies ist besonders relevant für neue Teammitglieder, die in ein Projekt einsteigen, oder für Entwickler, die mit der Arbeit an einem System beginnen. Ein solches Modell sollte Informationen enthalten, die ansonsten mühsam selbst abgeleitet werden müssten, und dadurch einen echten Mehrwert bieten.

Letztlich ist es entscheidend, dass Architekturmodelle nützlich, verständlich und informativ sind. Sie sollten den Entwicklern helfen, das System besser zu verstehen und effizienter damit zu arbeiten. Häufig werden Architekturmodelle mit Landkarten verglichen, die verschiedene Perspektiven auf ein System ermöglichen. Dies ist insbesondere bei großen Systemen mit vielen beteiligten Teams von Vorteil, da es hilft, das Gesamtsystem zu überblicken. Zudem erleichtert es die Identifikation von Abhängigkeiten zwischen den Teams.

Ein weiterer Nutzen von Architekturmodellen zeigt sich bei der Fehlerbehebung. Anhand der Fehlerbeschreibung kann ein gut gepflegtes Modell dabei unterstützen, die betroffenen Bereiche

des Systems zu identifizieren und gezielt Anpassungen vorzunehmen. Da ein Architekturmodell das Originalsystem widerspiegelt, bietet es eine wertvolle Orientierungshilfe, um effizient und zielgerichtet vorzugehen. Ein Architekturmodell sollte gezielt dabei unterstützen, den Ausgangspunkt für Analysen zu identifizieren und mögliche Fehlerquellen im System einzugrenzen. Es gibt verschiedene Modellierungssprachen, die in diesem Zusammenhang relevant sind. Einige davon sind möglicherweise bereits bekannt. Im Folgenden wird ein kurzer Überblick gegeben, um diese Sprachen in den Kontext der Softwarearchitektur einzuordnen und ihre potenzielle Rolle zu verdeutlichen.

[page 042] | *page_042.png*

Eine zentrale Rolle spielt dabei die Unified Modeling Language (UML), die häufig in der Softwarearchitektur verwendet wird. Daneben ist auch das Entity-Relationship-Modell (ERM) weiterhin von Bedeutung. Insbesondere in Informationssystemen, die primär der Speicherung und Verarbeitung von Informationen dienen, ist es notwendig, diese Informationen strukturiert abzubilden. Das Entity-Relationship-Modell bietet hierfür eine etablierte Methode, um Daten in Form von Entitäten und deren Beziehungen zu modellieren.

Darüber hinaus ist die Modellierung von Geschäftsprozessen ein wesentlicher Aspekt, insbesondere bei der Entwicklung von Informationssystemen. Hier kommen Methoden wie die Business Process Model and Notation (BPMN) sowie die Ereignisgesteuerte Prozesskette (EPK) zum Einsatz. Beide Ansätze ermöglichen es, Geschäftsprozesse zu visualisieren und deren Abläufe zu strukturieren.

[page 043] | *page_043.png*

Das Entity-Relationship-Modell, das in der Regel bereits in Datenbankvorlesungen behandelt wird, zielt darauf ab, Daten in Form von Entitäten und deren Beziehungen zu modellieren. Es stellt eine grundlegende Methode dar, um die Datenstruktur eines Systems zu definieren und zu organisieren. Assoziationen zwischen Entitäten können ebenfalls Attribute besitzen, und es ist möglich, Kardinalitäten zu definieren, die angeben, wie viele Beziehungen zwischen den Entitäten bestehen können. Dies ist ein wesentlicher Bestandteil der Darstellung eines Datenmodells.

[page 044] | *page_044.png*

In der Praxis wird jedoch häufig die Unified Modeling Language (UML) verwendet, um solche Modelle zu erstellen. UML ist ein offener Standard für Modellierungssprachen, der von der Object Management Group (OMG) entwickelt wurde. Dieser Standard hat sich im Laufe der Zeit weiterentwickelt und umfasst verschiedene Versionen.

UML strukturiert die Modellierungssprache in unterschiedliche Diagrammtypen, die jeweils spezifische Aspekte eines Systems visualisieren. Es gibt zahlreiche Software-Tools auf dem Markt, die diesen Standard unterstützen und die Erstellung von UML-Diagrammen erleichtern. Die Nutzung von UML kann auf unterschiedliche Weise erfolgen: Einige Anwender setzen die Sprache sehr formal ein, während andere sie eher als Skizzierwerkzeug verwenden. In der Praxis hat sich gezeigt, dass die formale Anwendung von UML oft mit einem hohen Aufwand verbunden ist, der nicht immer im Verhältnis zum Nutzen steht. Häufig weicht der resultierende Code von der ursprünglichen Modellierung ab, weshalb eine weniger formale, skizzenhafte Nutzung in vielen Fällen bevorzugt wird. Die Unified Modeling Language (UML) umfasst eine Vielzahl von Konzepten

und Diagrammtypen, die ein breites Spektrum an Modellierungsanforderungen abdecken. Sie hat sich als Standard etabliert, der ein gemeinsames Verständnis für die Darstellung von Systemen und deren Strukturen ermöglicht. Insbesondere in der Informatik sind viele der Diagrammtypen, wie beispielsweise Klassendiagramme, weit verbreitet und werden häufig in der Ausbildung behandelt. Dadurch können Fachleute in der Regel ohne zusätzliche Erläuterungen oder Legenden die Inhalte solcher Diagramme interpretieren, was die Kommunikation und Zusammenarbeit erleichtert.

[page 045] | *page_045.png*

Es gibt eine Vielzahl von Software-Tools, die die Erstellung von UML-Diagrammen unterstützen. Zu den verfügbaren Online-Tools gehören unter anderem Lucidchart, Visual Paradigm und diagrams.net, die sich durch unterschiedliche Funktionalitäten und Benutzerfreundlichkeit auszeichnen. Diese Werkzeuge bieten oft nicht nur Unterstützung für UML, sondern auch für andere Modellierungssprachen, was ihre Einsatzmöglichkeiten erweitert. In der Praxis wird UML häufig weniger formal verwendet, und die Tools erlauben es, Diagramme flexibel und an die jeweiligen Bedürfnisse angepasst zu erstellen. Im Desktop-Bereich gibt es beispielsweise das Tool „Sparx Systems Enterprise Architect“. Dieses ist auch in einer Web-Version verfügbar und zeichnet sich durch umfangreiche Funktionalitäten aus. Im Vergleich zu typischen Online-Tools bietet es die Möglichkeit, deutlich mehr Informationen an ein Modell anzuhängen. So können beispielsweise spezielle UML-Informationen bis auf Attributebene in Form von Tags hinzugefügt werden, was bei vielen anderen Tools nicht möglich ist.

Ein weiteres Beispiel ist Eclipse, das ebenfalls UML-Unterstützung bietet. Allerdings wird es in der Praxis oft als weniger benutzerfreundlich und eher formal sowie sperrig wahrgenommen. Microsoft Visio ist ein weiteres bekanntes Tool, das vielseitig einsetzbar ist und auch für die Erstellung von UML-Diagrammen genutzt werden kann. Darüber hinaus gibt es Tools wie ArgoUML, das eine Open-Source-Alternative darstellt, sowie Archi, das speziell für die ArchiMate-Notation entwickelt wurde. ArchiMate ist besonders im Bereich der Enterprise-Architektur relevant, wird jedoch gelegentlich auch in der Software-Architektur eingesetzt.

ArchiMate ist ein Beispiel für eine sehr formale Modellierungssprache, und das zugehörige Tool Archi ist entsprechend restriktiv in der Anwendung. Es erlaubt nur die Modellierung innerhalb der vorgesehenen Sprachregeln. Wenn diese nicht eingehalten werden, ist eine Modellierung nicht möglich. Dies ist ein wichtiger Aspekt, den Anwender beachten sollten. Darüber hinaus existieren für verschiedene integrierte Entwicklungsumgebungen (IDEs) zahlreiche Plugins, die die Erstellung und Bearbeitung von UML-Diagrammen unterstützen. Dies gibt einen Überblick über einige der verfügbaren Tools auf dem Markt. Im Folgenden werden verschiedene Diagrammtypen betrachtet, die für die System- oder Softwarearchitektur von Bedeutung sein können. Ein prominentes Beispiel ist das Use-Case-Diagramm, das vermutlich bereits bekannt ist. Dieses Diagramm dient dazu, ein grundlegendes Verständnis des Systems zu schaffen, indem es die Frage beantwortet: „Was leistet das System?“.

[page 046] | *page_046.png*

In einem Use-Case-Diagramm wird das System als eine Box dargestellt, die verschiedene Anwendungsfälle (Use Cases) implementiert. Diese Use Cases können in Beziehung zueinander stehen. Zusätzlich werden Akteure (Actors) dargestellt, die mit dem System interagieren und die jeweiligen Use Cases nutzen. Dieses Diagramm bietet eine grobe Übersicht über die Funktionalitäten des Systems und ermöglicht es, die Struktur und mögliche Abhängigkeiten

zwischen den Use Cases zu erkennen.

Anwendungsfälle können sich gegenseitig einbeziehen oder erweitern. Diese Beziehungen werden durch die Begriffe „Include“ und „Extend“ beschrieben, die in diesem Zusammenhang bekannt sein sollten. Solche Diagramme können beispielsweise verwendet werden, um eine Kontextsicht eines Systems darzustellen. Dabei wird das System als eine Art Blackbox betrachtet, die ihre Funktionalitäten nach außen hin bereitstellt. Allerdings wird diese Blackbox-Sicht in vielen Projekten nicht konsequent umgesetzt, da sie durch die detaillierte Darstellung der Use Cases teilweise aufgehoben wird. Insbesondere bei größeren Systemen kann dies zu einer sehr detaillierten und komplexen Darstellung führen. Bei komplexen Systemen können die Anzahl der Use Cases schnell in die Hunderte gehen, was eine übersichtliche Darstellung erschwert. Während Use-Case-Diagramme für einfache Systeme eine geeignete Alternative darstellen können, ist ihre Anwendung bei umfangreicheren Systemen weniger empfehlenswert. Ein Beispiel hierfür ist das ERK-System. Dieses System bietet unter anderem die Möglichkeit, Anbieter zu verwalten. Diese grundlegende Funktionalität kann durch weitere Features erweitert werden, wie etwa die Durchführung von Performance-Bewertungen für Anbieter oder die Durchführung von Risikoanalysen.

[page 047] | *page_047.png*

Ein Beispiel für eine solche Risikoanalyse könnte die Bewertung der Wahrscheinlichkeit sein, dass bestimmte Reisen nicht angeboten werden können oder dass vereinbarte Kontingente nicht erfüllt werden. Weitere Funktionen könnten die Verwaltung von Raten umfassen, wie etwa die Genehmigung von Raten durch Manager. Diese Beispiele verdeutlichen, dass es sich hierbei lediglich um eine Auswahl der Anwendungsfälle und Anforderungen des ERK-Systems handelt. Ein vollständiges Use-Case-Diagramm für ein derart komplexes System wäre entsprechend noch umfangreicher und schwer zu handhaben.

Daher ist die Nutzung von Use-Case-Diagrammen vor allem bei einfachen Systemen sinnvoll. Für komplexere Systeme sind sie aufgrund der Vielzahl an Anwendungsfällen und der damit verbundenen Komplexität weniger geeignet. Im Folgenden wird ein weiteres Verhaltensdiagramm betrachtet.

[page 048] | *page_048.png*

Ein Aktivitätsdiagramm ist eine einfache und effektive Methode, um Prozesse darzustellen, die innerhalb eines Systems ablaufen können. Es eignet sich insbesondere dazu, Haupt-Use-Cases zu visualisieren, indem Startzustände, Aktivitäten und deren Verzweigungen sowie Bedingungen abgebildet werden. Dadurch lässt sich ein grober Überblick über das Verhalten eines Systems gewinnen, wobei auch die Interaktion mehrerer Objekte berücksichtigt werden kann.

[page 049] | *page_049.png*

Diese Diagramme bieten eine unkomplizierte Möglichkeit, Geschäftsprozesse zu erfassen und visuell darzustellen. Elemente wie Verzweigungen, Parallelisierung und Synchronisierung können dabei ebenfalls modelliert werden. Darüber hinaus lassen sich mit Aktivitätsdiagrammen auch Algorithmen innerhalb eines Systems anschaulich darstellen. Ein Beispiel hierfür könnte die Visualisierung eines Bubble-Sort-Mechanismus sein, bei dem der Ablauf des Algorithmus übersichtlich in einem Diagramm abgebildet wird.

Allerdings ist es bei der Darstellung von Algorithmen auf dieser Ebene gelegentlich fraglich, ob ein solches Diagramm für Entwickler tatsächlich hilfreicher ist als der eigentliche Quellcode. In einigen Fällen kann der Code selbst eine klarere und präzisere Darstellung des Algorithmus bieten als ein visuelles Modell.

[page 050] | *page_050.png*

Ein weiterer Aspekt, der in der Software-Architektur gelegentlich Anwendung findet, sind Zustandsmodelle. Diese Modelle ermöglichen es, Zustände eines Systems sowie die Übergänge zwischen diesen Zuständen basierend auf bestimmten Ereignissen darzustellen. Insbesondere bei komplexen Zustandsautomaten kann dies sinnvoll sein, beispielsweise wenn ein Geschäftsobjekt innerhalb eines Prozesses viele verschiedene Zustände durchlaufen kann und dabei spezifische Regeln für jeden Zustand gelten. Zustandsmodelle können in solchen Fällen eine klare Übersicht darüber liefern, unter welchen Bedingungen ein Zustandswechsel erfolgen kann oder nicht.

[page 051] | *page_051.png*

In der Architekturdokumentation finden sich jedoch häufiger Darstellungen, die sich auf die Bausteinsicht beziehen, während die bisher betrachteten Modelle eher für Laufzeitsichten relevant sind. Für die Bausteinsicht werden oft Klassen- und Schnittstellendiagramme in UML verwendet, ebenso wie Komponentendiagramme.

Klassendiagramme, die Attribute und Operationen enthalten, sind besonders nützlich, wenn man eine detaillierte Analyse eines Bausteins auf einer feineren Ebene durchführen möchte, um die zu erstellenden Artefakte zu identifizieren. Noch häufiger anzutreffen sind jedoch Komponentendiagramme, da diese die grundlegenden Bausteine eines Systems darstellen.

In der Darstellung von Komponenten in der Software-Architektur gibt es verschiedene Möglichkeiten, Schnittstellen und deren Beziehungen zu visualisieren. Häufig werden Schnittstellen durch Symbole gekennzeichnet, beispielsweise durch eine sogenannte „Lollipop-Notation“, die angebotene Schnittstellen repräsentiert. Benötigte Schnittstellen hingegen werden oft durch einen Halbkreis dargestellt. Diese Darstellungsformen bieten unterschiedliche Ansätze, um die Interaktion zwischen Komponenten zu verdeutlichen. Im Rahmen detaillierter Entwürfe können diese Notationen weiter verfeinert werden, um das Zusammenspiel der Schnittstellen zu optimieren und die Wartbarkeit der Architektur zu gewährleisten.

Eine alternative, ausführlichere Darstellung von Schnittstellen besteht darin, diese als eigenständige Interface-Elemente darzustellen, die mit einem entsprechenden Stereotyp als „Interface“ gekennzeichnet sind. Darüber hinaus existiert das Konzept der sogenannten Ports, die als Zugänge zu einer Komponente dienen. Über Ports können verschiedene Kommunikationskanäle abgebildet werden. Allerdings wird diese Darstellungsweise in der Praxis vergleichsweise selten verwendet.

[page 052] | *page_052.png*

Für die Strukturierung in der Bausteinsicht ist das Klassendiagramm insbesondere auf einer detaillierten, niedrigeren Ebene von Bedeutung. Es wird häufig verwendet, wenn ein einzelner Baustein genauer dargestellt werden soll, beispielsweise ein System für das Lieferantenmanagement. In einem solchen Fall könnte ein Service modelliert werden, der Lieferanten verwaltet, etwa mit einer Funktion wie „findAllSuppliers“. Dabei können auch die

Daten eines Lieferanten modelliert werden, beispielsweise durch Attribute wie eine eindeutige Kennung (Code), einen Namen, eine Adresse und ein zugehöriges Land. Gegebenenfalls könnten auch Enumerationen für bestimmte Eigenschaften eines Lieferanten verwendet werden.

In der Regel werden in solchen Modellen einfache Getter- und Setter-Methoden, die in Programmiersprachen wie Java häufig vorkommen, weggelassen, um die Darstellung übersichtlicher zu halten. Stattdessen konzentriert man sich auf die wesentlichen Eigenschaften und Schnittstellen. Zu jeder angebotenen Schnittstelle gehört in der Regel auch eine Implementierung, die die Funktionalität bereitstellt. Darüber hinaus können in Klassendiagrammen auch Framework-Klassen modelliert werden, die in der Architektur verwendet werden.

Es ist ebenfalls gängig, dass Dienste auf einer abstrakten Service-Klasse basieren, die grundlegende Funktionalitäten bereitstellt. In der Vergangenheit umfassten solche Basisklassen oft Funktionen wie Logging, Authentifizierung oder Autorisierung. Mit modernen Frameworks, wie beispielsweise Spring im Java-Umfeld, gibt es jedoch alternative Ansätze, um solche Funktionen zu implementieren und wiederzuverwenden.

[page 053] | *page_053.png*

Auf einer höheren Abstraktionsebene wird häufig das Paketdiagramm (Package Diagram) verwendet. Dieses dient dazu, die Struktur eines Systems durch die Darstellung von Paketen und deren Abhängigkeiten zu visualisieren. Es wird oft genutzt, um ein Paket als eine Komponente darzustellen, was eine Bausteinsicht auf einer höheren Ebene ermöglicht. Dabei wird deutlich, wie die Abhängigkeiten zwischen den Paketen organisiert sind. Allerdings werden in einem Paketdiagramm keine Schnittstellen explizit dargestellt.

[page 054] | *page_054.png*

Wenn eine detailliertere Darstellung erforderlich ist, wird häufig ein Komponentendiagramm verwendet. Dieses erlaubt es, Schnittstellen explizit zu modellieren und Abhängigkeiten zwischen Komponenten darzustellen, auch wenn diese nur abstrakt und ohne genaue Spezifikation der Abhängigkeitsdetails beschrieben werden. Ein Komponentendiagramm bietet somit eine feinere Granularität als ein Paketdiagramm und ist ein wichtiges Element der UML (Unified Modeling Language).

Es ist zu beachten, dass der Begriff „Komponente“ in der UML sehr flexibel ist. Eine Komponente kann verschiedene Bedeutungen haben, je nach Kontext. Sie kann einen Baustein, eine Schicht, eine Deployment Unit oder sogar ein komplettes System repräsentieren. Daher ist es in Projekten oft notwendig, den Begriff „Komponente“ klar zu definieren und entsprechende Konventionen festzulegen, um Missverständnisse zu vermeiden.

[page 055] | *page_055.png*

Interaktionsdiagramme spielen eine wichtige Rolle, um die Zusammenarbeit und den Austausch zwischen Bausteinen oder Komponenten eines Systems zu visualisieren. In der UML wird hierfür häufig das Sequenzdiagramm verwendet. Es dient dazu, Abläufe und Interaktionen zwischen verschiedenen Akteuren und Systemkomponenten darzustellen.

Ein typisches Beispiel könnte wie folgt aussehen: Ein Benutzer (User) legt einen Lieferanten

(Supplier) an. Diese Aktion wird über einen Enterprise Service Bus (ESB) veröffentlicht. Der ESB ist ein Konzept, das den Austausch von Informationen und Nachrichten zwischen verschiedenen Systemen an einer zentralen Stelle ermöglicht. In diesem Fall wird über den ESB die Information verbreitet, dass ein neuer Lieferant angelegt wurde. Anschließend kann der Benutzer weitere Aktionen durchführen, wie beispielsweise eine Bewertung für den Lieferanten abzugeben oder eine Freigabe anzufordern.

In einem alternativen Ablauf, der in einem sogenannten „Alternative Path“-Fragment dargestellt wird, kommt ein zweiter Akteur ins Spiel. Dieser kann entweder die Freigabe erteilen oder ablehnen. Solche Sequenzdiagramme bieten eine anschauliche Möglichkeit, Abläufe und Entscheidungswege in einem System zu modellieren.

Ein ähnliches Beispiel wurde bereits in einem anderen Kontext, etwa bei einem Windradsystem, vorgestellt. Obwohl diese Diagramme nicht ausschließlich auf die UML beschränkt sind, finden sie auch in anderen Modellierungsmethoden und in der Praxis von Informationssystemen häufig Anwendung.

[page 056] | *page_056.png*

Ein weiteres Konzept, das in diesem Zusammenhang kurz erwähnt werden soll, ist das Fundamental Modeling Concepts (FMC). Dieses Modellierungskonzept wurde am Hasso-Plattner-Institut entwickelt und wird dort häufig gelehrt. Es findet insbesondere im SAP-Kontext breite Anwendung, da viele Informationssysteme Schnittstellen zu SAP-Systemen aufweisen, die weltweit stark verbreitet sind. Aus diesem Grund begegnet man der FMC-Notation in der Praxis immer wieder.

Die FMC-Notation bietet eine vergleichsweise einfache Möglichkeit, ein System oder einen Systemverbund darzustellen. Dabei werden Akteure häufig durch ein Strichmännchen in einem Kasten symbolisiert. Neben menschlichen Akteuren können auch andere Entitäten, wie beispielsweise Organisationen oder Systeme, als Akteure modelliert werden. Ein Beispiel, das auf der offiziellen Webseite von FMC zu finden ist, zeigt eine Reiseagentur. In diesem Szenario kann ein Kunde über die Reiseagentur Reservierungen anfragen. Dies wird durch ein kleines Pfeilsymbol dargestellt, das anzeigt, wer eine Beziehung initiiert. In diesem Fall wird eine Bestellung angefragt, um ein Ticket zu erhalten. In einem weiteren Szenario könnten interessierte Personen lediglich Informationen von der Reiseagentur erhalten wollen, beispielsweise im Rahmen von Werbe- oder Beratungsanfragen. Dieser Austausch ist bidirektional, das heißt, es wird keine eindeutige Richtung der Kommunikation vorgegeben. Die Reiseagentur selbst wird im Modell als ein System dargestellt, das aus verschiedenen Interkomponenten oder Bausteinen besteht, wie etwa einem Reservierungssystem und einem Informations-Helpdesk.

Zusätzlich gibt es in der Darstellung sogenannte Informationskanäle, über die Informationsobjekte, wie beispielsweise Reiseinformationen oder Kundendaten, ausgetauscht werden. Diese Informationsobjekte werden in sogenannten Speicherorten (Storage Locations) oder einfach „Locations“ abgelegt. Innerhalb dieser Speicherorte befinden sich Kanäle, die beispielsweise Reservierungen, Kundendaten oder andere relevante Informationen enthalten. Der Datenfluss zwischen den einzelnen Komponenten wird durch Pfeile visualisiert.

Für Personen, die mit der UML-Notation vertraut sind, kann diese Art der Darstellung zunächst ungewohnt wirken. Obwohl diese Notation nicht so häufig verwendet wird, wurde sie der

Vollständigkeit halber hier ebenfalls vorgestellt.

[page 057] | *page_057.png*

Im Anschluss wird auf weitere Modellierungssprachen eingegangen, darunter die Geschäftsprozessmodellierung. Diese wird insbesondere durch SAP stark geprägt, beispielsweise durch die Ereignisgesteuerten Prozessketten (EPK). Die grundlegende Idee der Ereignisgesteuerten Prozessketten (EPK) besteht darin, Ereignisse und Funktionen zu modellieren. Ein Beispiel hierfür ist ein Kundenauftrag, der als Ereignis dargestellt wird. Daraufhin folgt eine Funktion, wie etwa die Überprüfung des Kundendatensatzes, um festzustellen, ob der Kunde den Anforderungen des Auftrags entspricht. An dieser Stelle können Verzweigungen auftreten, die durch Symbole wie „X“ (exklusives Oder) oder „O“ (inklusives Oder) formal dargestellt werden.

Falls der Kundendatensatz nicht vorhanden ist, wird ein neues Ereignis ausgelöst, das die Anlage eines neuen Kundendatensatzes beschreibt. Sobald der Datensatz vorhanden ist, kann der Prozess fortgesetzt werden, beispielsweise mit der Bestätigung des Auftrags. Alternativ könnte auch eine Prüfung der Zahlungsmoral des Kunden erfolgen. Auf diese Weise lassen sich Prozesse übersichtlich und strukturiert darstellen. Dieses Modellierungskonzept wird auch heute noch in bestimmten Bereichen eingesetzt.

[page 058] | *page_058.png*

Allerdings hat sich mittlerweile ein anderer Standard etabliert, der im Folgenden betrachtet wird: die Business Process Model and Notation (BPMN). Darüber hinaus wird kurz auf eine weitere Modellierungssprache eingegangen, die Case Management Model and Notation (CMMN). Obwohl CMMN anfangs auf große Begeisterung stieß, konnte sich dieser Standard nicht in gleichem Maße durchsetzen. Der Standard der Business Process Model and Notation (BPMN) basiert auf der Verwendung von sogenannten Swimlanes, die verschiedene Akteure innerhalb eines Prozesses darstellen. Beispielsweise könnte eine Swimlane für eine Finanzinstitution stehen, eine andere für einen Einzelhändler. Innerhalb des Einzelhandels können wiederum weitere Akteure wie der Vertrieb oder die Logistik in separaten Swimlanes dargestellt werden.

In BPMN werden Prozesse durch Start-Ereignisse initiiert, gefolgt von Aktivitäten, die als Aufgaben definiert sind. Ein Beispiel für eine solche Aufgabe könnte die Festlegung der Zahlungsmethode sein. An dieser Stelle können Gateways verwendet werden, um Verzweigungen im Prozess darzustellen. So könnte etwa entschieden werden, ob die Zahlung per Scheck, Barzahlung oder Kreditkarte erfolgt.

Im Fall einer Kreditkartenzahlung ist eine Autorisierung erforderlich. Hierbei wird eine Nachricht an die Finanzinstitution gesendet, die die Zahlung autorisiert und eine Rückmeldung gibt. Erst nach Erhalt dieser Rückmeldung kann der Prozess fortgesetzt werden, indem die Kreditkartenzahlung verarbeitet wird.

Bei einer Barzahlung hingegen ist der Prozess in der Regel einfacher: Die Zahlung wird akzeptiert, das Paket wird bereitgestellt und an den Kunden geliefert. Damit ist der Prozess der Bestellung und Auslieferung abgeschlossen. Dieses Modell ist relativ leicht verständlich und wird häufig verwendet.

Herausfordernd wird es jedoch, wenn Prozesse nicht sequenziell ablaufen, sondern komplexere Abhängigkeiten oder parallele Abläufe enthalten. In solchen Fällen stößt das Modell an seine

Grenzen. In der Praxis treten solche komplexen Szenarien relativ häufig auf. Zwar lassen sich Prozesse oft als sequenzielle Abläufe modellieren, jedoch gibt es zahlreiche Ausnahmen, die berücksichtigt werden müssen. Insbesondere in der Logistik, einem Bereich, der durch vielfältige Abhängigkeiten und Unwägbarkeiten geprägt ist, zeigt sich dies deutlich. Ein Beispiel hierfür ist der Transport von Waren, etwa von Hongkong nach Hamburg. Während eines solchen Transports können zahlreiche unvorhergesehene Ereignisse eintreten, die den Prozess beeinflussen. Ein bekanntes Beispiel ist die Blockade des Suezkanals durch ein Schiff, die umfangreiche Umplanungen erforderlich machte.

In solchen Fällen ist der Prozessverlauf nicht immer klar definiert, da es zu verschiedenen Abweichungen kommen kann. Beispielsweise können während des Transports unerwartete Änderungen auftreten, wie etwa die Entscheidung des Kunden, eine Verzollung vorzunehmen oder einen anderen Dienstleister zu beauftragen. Solche Szenarien führen zu einer erheblichen Komplexität in der Prozessgestaltung.

[page 059] | *page_059.png*

Um Geschäftsprozesse flexibler zu gestalten und besser auf solche Ausnahmen reagieren zu können, wurde die Idee verfolgt, die Case Management Model and Notation (CMMN) einzusetzen. Dieses Modell ermöglicht es, Geschäftsprozesse dynamischer und anpassungsfähiger zu gestalten, um den Anforderungen solcher komplexen und variablen Abläufe gerecht zu werden. In diesem Beispiel aus dem Versicherungsbereich wird von sogenannten „Cases“ gesprochen. Ein „Case“ bezeichnet dabei einen spezifischen Fall. Dies könnte beispielsweise ein Transportfall in der Logistik sein oder, im Kontext von Versicherungen, ein Schadensfall, bei dem der Versicherungsnehmer eine Entschädigung für einen eingetretenen Schaden beantragt. In solchen Fällen wird ein sogenanntes „Claims File“ als Case erstellt. Dieser Case durchläuft verschiedene Phasen, in denen unter anderem Verantwortlichkeiten geklärt, Basisinformationen hinzugefügt und der Fall insgesamt bearbeitet wird.

Die einzelnen Phasen des Prozesses werden durch sogenannte Meilensteine (Milestones) miteinander verknüpft. Ein Meilenstein kann dabei auch ein Eintrittskriterium definieren, das erfüllt sein muss, bevor der nächste Schritt im Prozess erreicht werden kann. Auf diese Weise lassen sich die verschiedenen Phasen und Aufgaben eines Cases strukturiert miteinander verbinden.

Ein Beispiel für die Flexibilität dieses Ansatzes ist die Möglichkeit, Prozessaufgaben (Process Tasks) zu hinterlegen. Diese können entweder als eingebettete BPMN-Prozesse (Business Process Model and Notation) oder als manuelle Aufgaben definiert werden, die von einer Person ausgeführt werden müssen. Dieser Ansatz bietet eine deutlich höhere Flexibilität im Vergleich zu einem strikt sequenziellen Ablauf. Die Grundidee besteht darin, ein Modell zu erstellen, das eine hohe Anpassungsfähigkeit ermöglicht. Es wird ein Rahmen definiert, der typische Aufgaben und Abläufe abbildet. Innerhalb dieses Rahmens kann der tatsächliche Ablauf im Alltag überwacht, gesteuert und analysiert werden. Dadurch lassen sich Erkenntnisse über typische Prozessmuster gewinnen, und es wird möglich, den Benutzern gezielte Unterstützung in spezifischen Situationen anzubieten.

Dieser Ansatz war eine Zeit lang recht populär. Der aktuelle Stand dieser Methodik ist nicht bekannt, dennoch wird sie hier der Vollständigkeit halber erwähnt. Sie stellt einen interessanten alternativen Ansatz dar und verdeutlicht, dass ein Modell niemals vollständig mit der Realität übereinstimmt. Dies kann zu Missverständnissen führen, insbesondere wenn Modelle wie BPMN

(Business Process Model and Notation) verwendet werden. Solche Modelle können in der Praxis an ihre Grenzen stoßen, beispielsweise wenn Sonderfälle auftreten, die den Prozess blockieren und ein Fortlaufen verhindern.

Die Idee hinter diesen flexiblen Prozessen ist es, die Abläufe tatsächlich durchführbar zu machen. Es gibt Systeme, die Geschäftsprozesse modellieren und diese dann automatisiert ausführen. Diese Systeme überwachen zudem den Status der Prozesse und ermöglichen eine kontinuierliche Nachverfolgung, in welchem Zustand sich ein bestimmter Prozess gerade befindet. Hinter einer Aktivität können tatsächlich REST-Aufrufe auf Systeme hinzugefügt werden. Es gab eine Zeit lang relativ viele Produkte in diesem Kontext, und diese Ansätze waren eine Weile sehr beliebt. Mit der Zeit hat sich jedoch zunehmend gezeigt, dass diese Ansätze nicht immer vollständig mit der Realität übereinstimmen und daher nicht in allen Szenarien passend sind. Aus diesem Grund sind solche Lösungen heute weniger verbreitet, wenngleich sie nicht vollständig verschwunden sind. Der Wunsch, Prozesse zu steuern und deren Status zu kennen, bleibt weiterhin relevant.

Wenn ein solches System vorhanden ist, bietet es die Möglichkeit, auf Basis der erfassten Daten Schwachstellen in Prozessen zu identifizieren und gezielt Verbesserungen vorzunehmen. Die Modellierung von Geschäftsprozessen und deren systemische Unterstützung stellt ein eigenständiges und umfangreiches Themengebiet dar. Für die Architektur von Informationssystemen ist dies von Bedeutung, da Geschäftsprozesse häufig auch informell implementiert und abgebildet werden.

Ein Beispiel für eine solche Lösung war Camunda, eine Open-Source-Plattform, die eine Zeit lang sehr populär war. Sie bot die Möglichkeit, Workflow-Engines in ein System zu integrieren, um Prozesse zu verfolgen und zu steuern.

[page 060] | *page_060.png*

Im Folgenden wird das nächste Kapitel behandelt. Die Verortung der Architektur innerhalb eines Entwicklungsprozesses hängt maßgeblich von der Art des gewählten Entwicklungsprozesses ab. Es existiert keine einheitliche Vorgehensweise, da verschiedene Modelle unterschiedliche Ansätze und Anforderungen mit sich bringen. Grundsätzlich lassen sich zwei grundlegende Kategorien von Entwicklungsmodellen unterscheiden.

[page 061] | *page_061.png*

Zum einen gibt es die klassischen, traditionellen Modelle wie das Wasserfallmodell, das in seiner iterativen Variante häufiger Anwendung findet. Ein weiteres Beispiel ist das V-Modell, das insbesondere im öffentlichen Sektor nach wie vor weit verbreitet ist. Auch in der Automobilindustrie wird es teilweise eingesetzt, insbesondere wenn Sicherheitsstandards dies vorschreiben und regulatorische Vorgaben erfüllt werden müssen. Ein weiteres traditionelles Modell ist der Rational Unified Process (RUP), der in der Vergangenheit ebenfalls häufig verwendet wurde. Diese Modelle zeichnen sich dadurch aus, dass sie stark auf Projektplanung setzen, um den Entwicklungsprozess zu steuern.

Auf der anderen Seite stehen die agilen Modelle, die sich zunehmend durchsetzen. Diese zeichnen sich durch ihre Flexibilität und iterative Herangehensweise aus. Beispiele hierfür sind Methoden wie Lean, Extreme Programming, Kanban und insbesondere Scrum, das in der Praxis sehr häufig Anwendung findet. In der Praxis finden sich die genannten Modelle, insbesondere die agilen

Ansätze, immer häufiger.

[page 062] | *page_062.png*

Betrachtet man jedoch ein klassisches Projekt, stellt sich die Frage, wie die Architektur in einem solchen Entwicklungsprozess verortet ist. Im Wasserfallmodell beispielsweise beginnt der Prozess typischerweise mit der Phase des Requirements Engineerings. Diese Phase dient der Klärung und Spezifikation der Anforderungen. Ziel ist es, die Anforderungen möglichst vollständig und konsistent zu erfassen.

In der Realität zeigt sich jedoch häufig, dass die angestrebte Vollständigkeit und Konsistenz der Anforderungen nicht erreicht wird. Während der Implementierung treten oft Sonderfälle oder unberücksichtigte Aspekte auf, die zu Inkonsistenzen führen können. Diese Unvollständigkeiten haben zur Folge, dass der ursprüngliche Projektplan nicht wie vorgesehen eingehalten werden kann.

Die Architekturarbeit beginnt in diesem Kontext mit der Übergabe einer Spezifikation, in der die Anforderungen detailliert beschrieben und heruntergebrochen sind. Diese Spezifikation bildet die Grundlage für die architektonische Arbeit. Ziel ist es, auf Basis dieses Inputs die verschiedenen Architektursichten zu entwerfen. Dabei wird zunächst ein grobkörniger Entwurf erstellt, der im weiteren Verlauf immer detaillierter ausgearbeitet wird. Der Output der Architekturarbeit in klassischen Projekten kann sehr detailliert sein. Er reicht häufig bis hinunter auf die Ebene von Klassendiagrammen. Solche detaillierten Entwürfe wurden in der Vergangenheit oft erstellt und genutzt, um die Implementierung zu steuern und zu planen. Dabei wird festgelegt, wer welche Aufgaben in welchen Phasen übernimmt. Zudem werden Richtlinien entwickelt, an denen sich die Entwickler orientieren sollen. Bereits in dieser Phase können auch potenzielle Risiken identifiziert werden.

[page 063] | *page_063.png*

Im Gegensatz dazu stellt sich bei agilen Projekten zunächst die Frage, ob es in diesem Kontext überhaupt eine explizite Architektur gibt. Agile Vorgehensweisen, wie sie beispielsweise im Scrum-Framework angewendet werden, basieren auf der Zusammenarbeit in funktionsübergreifenden Teams und einem iterativen Entwicklungsprozess. Dabei entsteht häufig der Eindruck, dass Architektur eine untergeordnete Rolle spielt.

Es ist jedoch wichtig zu betonen, dass die Prinzipien des agilen Manifests, wie etwa „Funktionierende Software vor umfassender Dokumentation“, nicht bedeuten, dass Dokumentation oder Architektur vollständig vernachlässigt werden sollen. Vielmehr wird damit ausgedrückt, dass der Fokus auf der Entwicklung funktionierender Software liegt, ohne dabei die Bedeutung von Dokumentation oder Architektur grundsätzlich infrage zu stellen. In agilen Projekten wird bewusst darauf verzichtet, ein umfassendes, vollständig ausgearbeitetes Design des gesamten Systems im Voraus zu erstellen, wie es beim sogenannten „Big Design Up Front“ der Fall ist. Stattdessen wird das System schrittweise und iterativ entworfen, wobei der Fokus auf einzelnen Inkrementen liegt. Diese Herangehensweise berücksichtigt die Tatsache, dass sich Anforderungen im Laufe der Entwicklung ändern können, und ermöglicht es, flexibel darauf zu reagieren.

Ein zentrales Ziel ist es, möglichst schnell funktionierende Ergebnisse zu liefern. Dabei wird jedoch nicht auf Architektur verzichtet. Auch wenn Architektur in agilen Methoden wie Scrum nicht

explizit als formales Artefakt vorgesehen ist, entsteht sie dennoch zwangsläufig. Selbst wenn es keine bewusste Entscheidung für eine Architektur gibt, wird sich das Team auf eine Struktur einigen und diese schaffen, um effektiv arbeiten zu können.

In agilen Projekten ist es üblich, dass die Verantwortung für die Architektur nicht bei einer einzelnen Person, wie einem dedizierten Architekten, liegt. Stattdessen übernimmt das gesamte Team diese Aufgabe gemeinschaftlich. Die Architektur wird dabei evolutionär entwickelt, Schritt für Schritt, mit jedem neuen Inkrement. Dies stellt einen wesentlichen Unterschied zu traditionellen, stark vorab geplanten Ansätzen dar. In einem Projekt, in dem ein Audit durchgeführt wurde, wurde das Team, das nach agilen Prinzipien arbeitete, gefragt, ob es eine Architekturbeschreibung oder Überlegungen zur Architektur gibt. Die Antwort lautete, dass sie bewusst keine Architektur erstellen wollten. Diese Aussage war überraschend, da in den meisten Fällen zumindest anerkannt wird, dass eine Architektur notwendig ist.

Bei einer späteren Analyse des Codes zeigte sich jedoch, dass dennoch eine Art von Struktur vorhanden war. Diese war jedoch von schlechter Qualität, was darauf hindeutete, dass der bewusste Verzicht auf Architekturgedanken dem Projekt nicht zuträglich war. Das Team hatte offenbar die Absicht, unnötige Komplexität zu vermeiden, was grundsätzlich ein nachvollziehbares Ziel ist. Allerdings verdeutlicht dieses Beispiel ein häufiges Missverständnis: Der Verzicht auf jegliche Architekturarbeit führt nicht zu einer besseren, sondern oft zu einer chaotischen und ineffizienten Struktur.

Das Ziel sollte vielmehr sein, die Architektur auf das Wesentliche zu reduzieren und sich bei jedem Inkrement zu fragen, welche architektonischen Entscheidungen für den nächsten Schritt notwendig sind. Dies hilft, überflüssige Komplexität zu vermeiden, wie sie beispielsweise durch überdimensionierte Lösungen oder unnötigen Aufwand entstehen könnte, der keinen Mehrwert für die Software liefert. Gleichzeitig muss jedoch ein Gleichgewicht gefunden werden, um zu verhindern, dass durch fehlende Planung und Überlegungen später umfangreiche Refactorings erforderlich werden. Ein unüberlegtes Vorgehen ohne architektonische Leitlinien kann langfristig zu erheblichen Problemen führen. Später kann es vorkommen, dass erkannt wird, dass das bisherige Vorgehen nicht funktioniert und umfangreiche Refactorings notwendig werden. Je mehr jedoch über die Anforderungen und den übergeordneten Kontext nachgedacht wird, desto besser können solche Refactorings vermieden werden. Es handelt sich hierbei um einen Balanceakt, bei dem es darauf ankommt, ein Gleichgewicht zu finden. Die Idee besteht darin, parallel zum Minimum Viable Product (MVP) auch eine minimale Architektur zu entwickeln – eine Architektur, die gerade ausreichend ist, um das MVP zu tragen.

Bei großen Projekten, insbesondere solchen mit vielen verschiedenen Teams, die agil arbeiten und gemeinsam an einem System oder einer Lösung arbeiten, wird dies jedoch deutlich komplexer. Solche Szenarien sind in der Praxis durchaus verbreitet.

[page 064] | *page_064.png*

An dieser Stelle stellt sich die Frage, was eine passende Architektur für ein Projektinkrement bedeutet und welche Herausforderungen sich bei der Entwicklung einer Architektur im Rahmen eines agilen Vorgehens ergeben. Hierzu gibt es von der Open Group ein hilfreiches agiles Architektur-Framework, das einige Aspekte hervorhebt, die agile Architektur beeinflussen und charakterisieren. Zunächst ist es wichtig, die verschiedenen Kräfte zu betrachten, die auf die Architektur einwirken. Es gibt verschiedene Einflussfaktoren, die auf die Architektur einwirken.

Ein zentraler Aspekt ist das Konzept der „Knowns“ und „Unknowns“. Dabei geht es darum, dass nicht alle Informationen im Voraus bekannt sein können. Es ist jedoch möglich, sich bewusst zu machen, welche Informationen fehlen. Beispielsweise ist zu Beginn eines Projekts oft unklar, wie viele Kunden das Produkt nutzen werden oder welche spezifischen Anforderungen in Zukunft unterstützt werden müssen. Dieses Bewusstsein über die Ungewissheiten ist hilfreich, um zu priorisieren, welche Entscheidungen tatsächlich getroffen werden müssen.

Ein weiterer wichtiger Punkt ist die Identifikation der wesentlichen Designentscheidungen für jedes Projektinkrement. Dabei spielen verschiedene Faktoren eine Rolle, wie etwa die Geschwindigkeit, mit der Ergebnisse geliefert werden müssen, um im Wettbewerb bestehen zu können, oder die Risikobereitschaft, die bei der Entwicklung und Gestaltung des Systems an den Tag gelegt wird. Auch die organisatorische Kultur, die in einem Unternehmen vorherrscht, hat einen erheblichen Einfluss. Es ist wichtig zu beachten, dass Agilität nicht ausschließlich auf Start-ups beschränkt ist. Auch in großen Konzernen gibt es agile Ansätze, die jedoch oft von spezifischen Rahmenbedingungen geprägt sind, die wiederum als Kräfte auf das Projekt wirken.

Schließlich ist auch die Frage von zentraler Bedeutung, welche Anforderungen und Erwartungen der Kunde an das Produkt oder die Lösung hat. Diese Kundenbedürfnisse stellen eine weitere wesentliche Einflussgröße dar, die bei der Entwicklung der Architektur berücksichtigt werden muss. Der Endbenutzer spielt eine zentrale Rolle bei der Gestaltung der Architektur. Dabei stellt sich die Frage, wie viel Agilität und Volatilität für den Endbenutzer akzeptabel sind. Betrachtet man beispielsweise Informationssysteme, die von Sachbearbeitern oder Experten genutzt werden, so ist zu beachten, dass diese Nutzergruppe in der Regel keine häufigen Änderungen an der Benutzeroberfläche wünscht. Ständige Anpassungen, wie etwa neue oder veränderte Bildschirmansichten, könnten dazu führen, dass die Nutzer ihre Arbeitsweise regelmäßig anpassen müssen, was die Effizienz beeinträchtigen kann. Diese Anwender bevorzugen in der Regel stabile und konsistente Systeme, die es ihnen ermöglichen, ihre Aufgaben effizient und ohne ständige Umstellungen zu erledigen.

Im Gegensatz dazu können Endkunden auf dem Markt oft eine höhere Frequenz und Geschwindigkeit von Änderungen erwarten und akzeptieren. Sie sind möglicherweise offener für Neuerungen und freuen sich darauf, neue Funktionen oder Designs kennenzulernen. In diesem Zusammenhang stellt sich auch die Frage nach der Stabilität und Beständigkeit der Vision und des Zwecks eines Produkts. Diese können sich im Laufe der Zeit ändern, was wiederum Auswirkungen auf die Architektur und die zugrunde liegenden Entscheidungen haben kann.

Diese Überlegungen verdeutlichen die verschiedenen Kräfte, die auf die Architektur und die damit verbundenen Entscheidungen einwirken. In diesem Zusammenhang ist auch die Unterscheidung zwischen Typ-1- und Typ-2-Entscheidungen relevant, ein Konzept, das von Jeff Bezos und Amazon eingeführt wurde. Typ-1-Entscheidungen sind solche, die schwer oder gar nicht rückgängig zu machen sind. Sie zeichnen sich durch ihre Unumstößlichkeit oder die Schwierigkeit aus, sie zu ändern. Ein anschauliches Bild hierfür ist eine Wand mit einem Durchgang, der zugemauert wird – ein Zurück gibt es in diesem Fall nicht mehr.

Im Gegensatz dazu stehen Typ-2-Entscheidungen, die leichter revidierbar sind. Sie lassen sich mit einer Tür vergleichen, die zwar geöffnet und geschlossen werden muss, was einen gewissen Aufwand erfordert, aber dennoch eine Rückkehr ermöglicht.

Im Kontext der Software-Architektur ist es von zentraler Bedeutung, Typ-1-Entscheidungen klar zu

identifizieren. Dabei gilt es, die verschiedenen Einflussfaktoren zu berücksichtigen, um herauszufinden, welche Entscheidungen besonders wichtig sind und daher mit größter Sorgfalt getroffen werden müssen. Eine mögliche Strategie besteht darin, solche Entscheidungen möglichst lange hinauszuzögern. Dies kann durch die Schaffung von Abstraktionen in der Architektur erreicht werden, die es erlauben, diese Entscheidungen erst zu einem späteren Zeitpunkt zu treffen, wenn mehr Informationen vorliegen, um fundiertere Entscheidungen zu ermöglichen.

Alternativ kann auch der Ansatz verfolgt werden, die Architektur so zu gestalten, dass eine spätere Änderung dieser Entscheidungen möglich bleibt. Dies würde bedeuten, die Architektur von Anfang an auf Evolution und Anpassungsfähigkeit auszurichten. Ein Beispiel für eine solche Herangehensweise wird im weiteren Verlauf erläutert. Eine weitere Strategie besteht darin, Entscheidungen bewusst zu treffen, auch wenn die Möglichkeit besteht, dass diese später verworfen werden müssen. In solchen Fällen wird die Entscheidung getroffen, weil sie zu diesem Zeitpunkt notwendig ist, obwohl sie weder aufgeschoben noch so gestaltet werden kann, dass sie flexibel anpassbar bleibt. Die Umsetzung erfolgt mit dem Bewusstsein, dass es im weiteren Verlauf erforderlich sein könnte, die getroffene Entscheidung zu revidieren und die entsprechende Komponente oder Lösung neu zu entwickeln. Auch der vollständige Neubau eines Systems oder von Teilen davon kann eine bewusste Strategie sein. Dies verdeutlicht das Spannungsfeld, in dem man sich bei der Entwicklung von Architekturen, insbesondere im agilen Kontext, bewegt.

Im agilen Vorgehen ist es charakteristisch, dass im Laufe der Zeit immer mehr Wissen über die Anforderungen und Rahmenbedingungen eines Systems gesammelt wird. Dieses Wissen wächst kontinuierlich, und rückblickend stellt sich oft die Erkenntnis ein, dass man mit den zu einem früheren Zeitpunkt verfügbaren Informationen möglicherweise andere Entscheidungen getroffen hätte. In vielen Projekten, insbesondere in solchen, die über einen längeren Zeitraum laufen, zeigt sich zudem häufig, dass die Architektur historisch gewachsen ist und daher nicht immer optimal auf die aktuellen Anforderungen abgestimmt ist. Der Begriff „historisch gewachsen“ beschreibt eine Situation, in der Entscheidungen auf Basis des damaligen Wissensstands getroffen wurden, die jedoch im Nachhinein nicht mehr optimal zu den aktuellen Anforderungen passen. Oftmals besteht in solchen Fällen keine einfache Möglichkeit, diese Entscheidungen rückgängig zu machen oder anzupassen, da sie tief in die bestehende Architektur integriert sind. Die Architektur hat sich somit im Laufe der Zeit entwickelt, ohne dass eine bewusste, langfristige Planung für die aktuellen Gegebenheiten erfolgt ist.

[page 065] | *page_065.png*

Das Konzept des „Last Responsible Moment“ (letzter verantwortungsvoller Moment) bietet eine Strategie, Entscheidungen so lange wie möglich hinauszuzögern, um sie auf einer fundierteren Wissensbasis treffen zu können. Die Möglichkeit, eine Entscheidung zu verschieben, hängt jedoch stark von ihrer Tragweite ab. Manche Entscheidungen, wie beispielsweise die Festlegung eines Datenmodells oder eines Datenbankschemas, müssen frühzeitig getroffen werden, da sie die Grundlage für die Arbeit mit den Daten bilden. Andere Entscheidungen, wie die Wahl eines spezifischen Frameworks, haben oft eine geringere Tragweite und können auch zu einem späteren Zeitpunkt angepasst werden.

Ein Beispiel hierfür ist die Abbildung von Geodaten in einem System. Selbst wenn diese Anforderung erst später konkretisiert wird, gibt es heutzutage Möglichkeiten, solche Änderungen umzusetzen. Moderne Technologien wie Hibernate bieten beispielsweise Abstraktionen, die es erleichtern, solche Anpassungen auch nachträglich vorzunehmen. Die Wahl des

Datenbankproviders sollte möglichst spät erfolgen, da diese Entscheidung oft mit langfristigen Verpflichtungen, wie etwa Lizenzgebühren oder Lizenzmodellen, verbunden ist, die eine größere Tragweite haben können. Es gibt jedoch einen Punkt, an dem tiefgreifende Datenbankfunktionen implementiert werden müssen. An diesem sogenannten „Last Responsible Moment“ müssen alle relevanten Entscheidungen getroffen und festgelegt werden. Dazu gehört unter anderem die Auswahl eines ORM-Frameworks, das die Abbildung von Geodaten unterstützt, sowie die Festlegung auf einen spezifischen Datenbankprovider, der die benötigten nativen Datenbankfunktionen bereitstellt.

In diesem Zusammenhang ist es wichtig, dass das Datenmodell ebenfalls final definiert wird, da es die Grundlage für die Implementierung der Datenbankfunktionen bildet. An diesem Punkt werden alle vorherigen Überlegungen und Entscheidungen zusammengeführt und endgültig fixiert. Bis zu diesem Moment besteht jedoch noch die Möglichkeit, Anpassungen vorzunehmen und Entscheidungen zu überdenken. Sobald jedoch die finale Entscheidung für eine bestimmte Datenbank getroffen wird, ist diese in der Regel nicht mehr ohne erheblichen Aufwand revidierbar.

Ein Beispiel für solche nativen Datenbankfunktionen könnten analytische Funktionen sein, die spezifisch für ein bestimmtes Datenbankprodukt entwickelt wurden. Solche Funktionen sind oft nicht ohne Weiteres auf andere Datenbanksysteme übertragbar, was die Bedeutung einer wohlüberlegten Entscheidung unterstreicht. Komplexe Datenbankabfragen umfassen nicht nur einfache Aggregationen wie die Berechnung einer Summe über eine Abfrage, sondern auch anspruchsvollere Operationen, beispielsweise die Berechnung von Aggregaten über Gruppen innerhalb eines Ergebnisses. Solche Funktionen sind jedoch oft spezifisch für den jeweiligen Datenbankanbieter implementiert und unterscheiden sich daher zwischen den Systemen. Dies bedeutet, dass man sich entweder auf den SQL-Standard der jeweiligen Datenbank festlegt oder auf spezifische Erweiterungen, die möglicherweise aus dem Bereich des sogenannten „NewSQL“ stammen. Diese Abhängigkeit verdeutlicht die Notwendigkeit, die Wahl des Datenbankanbieters sorgfältig zu treffen.

[page 066] | *page_066.png*

In der Phase vor der endgültigen Entscheidung besteht jedoch noch ein gewisser Spielraum, um Anpassungen vorzunehmen und Entscheidungen zu überdenken. Ein Ansatz, der in diesem Zusammenhang selten angewendet wird, aber eine interessante Möglichkeit bietet, ist das sogenannte „Set-based Design“. Im klassischen Vorgehen wird in der Regel eine einzelne Designoption ausgewählt und verfolgt. Diese wird dann bei Bedarf angepasst, wenn neue Anforderungen, Änderungen oder zusätzliche Erkenntnisse hinzukommen. Es kann jedoch vorkommen, dass sich im späteren Verlauf herausstellt, dass die gewählte Option nicht geeignet ist, was zu einem erheblichen Mehraufwand führen kann.

Das Konzept des Set-based Design verfolgt einen anderen Ansatz: Es werden mehrere Designentscheidungen parallel entwickelt und evaluiert. Im Laufe der Zeit werden diejenigen Optionen, die sich als ungeeignet erweisen, verworfen. Am Ende bleibt die optimale Lösung übrig, die bereits weitgehend ausgearbeitet ist. Dieser Ansatz kann dazu beitragen, den Entwicklungsprozess zu beschleunigen, da nicht erst nachträglich grundlegende Änderungen vorgenommen werden müssen, sondern die finale Lösung bereits auf einer fundierten Basis aufbaut. Die Anwendung des Set-based Design ist jedoch mit einem erheblichen Entwicklungsaufwand verbunden. Dieser Ansatz erfordert nicht nur mehr Ressourcen, sondern auch die Bereitschaft des Kunden, die damit verbundenen Kosten zu tragen. Diese Bereitschaft ist

nicht in jedem Projektumfeld gegeben. Dennoch kann Set-based Design eine sinnvolle Option sein, insbesondere wenn eine schnelle Entwicklung und Anpassung an sich ändernde Anforderungen von hoher Relevanz sind.

[page 067] | *page_067.png*

Abschließend soll noch ein Blick auf die Rolle der Architektur in Großprojekten geworfen werden. Ein prominentes Beispiel für die Strukturierung solcher Projekte ist das Scaled Agile Framework (SAFe). Dieses Framework wurde entwickelt, um große agile Programme zu organisieren und zu koordinieren. Es basiert auf einer hierarchischen Struktur, die mehrere Ebenen umfasst.

Auf der obersten Ebene steht das Portfolio, das eine Sammlung verschiedener Lösungen darstellt. Jede dieser Lösungen bildet eine weitere Ebene und wird in sogenannten Solution Trains entwickelt. Diese Solution Trains stellen eine spezifische Organisationseinheit dar, die wiederum in Agile Release Trains (ARTs) unterteilt wird. Ein Agile Release Train besteht aus mehreren Entwicklungsteams, die nach agilen Methoden wie Scrum oder Kanban arbeiten. Diese Teams bearbeiten ein gemeinsames Program Backlog und arbeiten kontinuierlich an der Umsetzung der geplanten Aufgaben.

Das Scaled Agile Framework zielt darauf ab, das gesamte Vorgehen – von der Portfolioebene über die einzelnen Lösungen bis hin zu den Entwicklungsteams – zu strukturieren und zu koordinieren. Dadurch wird eine effiziente Planung und Umsetzung von komplexen Projekten ermöglicht, die aus zahlreichen ineinandergreifenden Komponenten bestehen. Auf einer höheren Ebene, beispielsweise bei den sogenannten Program Increments (PIs), wird festgelegt, welche Ziele innerhalb eines Zeitraums von etwa drei Monaten erreicht werden sollen. Diese Ziele werden von den beteiligten Teams gemeinsam definiert und verbindlich vereinbart. Anschließend werden die Aufgaben innerhalb der Teams weiter aufgeteilt und konkretisiert. Dieser Prozess ist zeitaufwendig und Teil eines umfassenden Frameworks.

In diesem Zusammenhang gibt es auf den höheren Ebenen des Scaled Agile Frameworks spezifische Rollen wie den System Architect oder System Engineer. Diese Rollen sind dafür verantwortlich, die Struktur und das Zusammenspiel der verschiedenen Teams innerhalb eines Agile Release Trains zu koordinieren. Auf der Lösungsebene übernehmen Solution Architects oder Solution Engineers eine ähnliche Funktion. Sie sorgen dafür, dass die Arbeit der einzelnen Teams aufeinander abgestimmt ist und letztlich zu einer integrierten Gesamtlösung führt. Ohne diese übergreifende Koordination wäre es unwahrscheinlich, dass die Ergebnisse der einzelnen Teams den Anforderungen und Zielen auf der höheren Ebene gerecht werden.

Dies verdeutlicht, dass die Architektur auch in einem agilen Umfeld eine zentrale Rolle spielt, insbesondere bei der Sicherstellung, dass die verschiedenen Komponenten eines Projekts zu einer kohärenten und funktionierenden Gesamtlösung zusammengeführt werden.

[page 068] | *page_068.png*

Im nächsten Kapitel wird die Rolle des Architekten bzw. der Architektin näher betrachtet. Dabei ist es zunächst wichtig, eine grundlegende Unterscheidung zu treffen: Was genau versteht man unter dieser Rolle? Die Verantwortlichkeiten und Aufgaben dieser Rolle lassen sich zunächst unabhängig von der konkreten personellen Besetzung betrachten. Im Kern geht es darum, Anforderungen zu analysieren und in funktionale sowie nicht-funktionale Anforderungen, insbesondere

Qualitätsanforderungen, zu überführen. Diese Anforderungen müssen klar definiert werden, was eine zentrale Aufgabe darstellt. Im weiteren Verlauf wird noch detaillierter darauf eingegangen, wie dies im Einzelnen geschieht und welche Hilfsmittel dafür erforderlich sind.

[page 069] | *page_069.png*

Ein weiterer wesentlicher Aspekt dieser Rolle ist die Entwicklung und der Entwurf verschiedener Lösungsoptionen. Dabei obliegt es der verantwortlichen Person, fundierte Entscheidungen zu treffen, die sicherstellen, dass die definierten Qualitätsanforderungen erfüllt werden. Diese Entscheidungen müssen dokumentiert werden, um eine nachvollziehbare Grundlage für die Nachwelt zu schaffen. Darüber hinaus gehört es zu den Aufgaben, das Entwicklungsteam während des gesamten Prozesses beratend zu unterstützen.

Ein weiterer Schwerpunkt liegt auf der Kommunikation der Architektur. Die verantwortliche Person muss sicherstellen, dass alle Beteiligten ein gemeinsames Verständnis der Architektur haben. Zudem ist es essenziell, die mit der Architektur verbundenen Risiken zu identifizieren und zu überwachen. Dabei darf die Wirtschaftlichkeit der Lösung nicht außer Acht gelassen werden. Selbst die technisch beste Lösung ist nicht zielführend, wenn sie unverhältnismäßig hohe Kosten verursacht, zu viele Ressourcen bindet oder einen übermäßigen Aufwand erfordert.

Abschließend gehört es zu den Aufgaben dieser Rolle, die Architektur kontinuierlich zu bewerten und gegebenenfalls anzupassen. Diese fortlaufende Überprüfung stellt sicher, dass die Architektur den Anforderungen und Rahmenbedingungen des Projekts dauerhaft gerecht wird. Die Verantwortlichkeiten, die mit dieser Rolle einhergehen, sind klar definiert. Wer diese Rolle letztendlich ausfüllt, ist eine separate Frage. Unabhängig davon stellt sich die Frage, welche Fähigkeiten erforderlich sind, um diese Verantwortlichkeiten erfolgreich zu erfüllen.

[page 070] | *page_070.png*

Eine zentrale Anforderung ist ein breites Technologiewissen. Dieses wird oft durch das sogenannte „T-Skill-Profil“ veranschaulicht. Dabei symbolisiert der horizontale Balken des „T“ die Breite des Wissens über verschiedene Technologien und Themenbereiche. Zusätzlich wird von einer Architektin oder einem Architekten erwartet, dass sie oder er in mindestens einem spezifischen Bereich über tiefgehendes Fachwissen verfügt. Diese Spezialisierung wird durch den vertikalen Balken des „T“ dargestellt. Dieses Modell beschreibt das klassische Kompetenzprofil einer Architektin oder eines Architekten.

Darüber hinaus gibt es auch das sogenannte „Pi-Skill-Profil“, das eine Erweiterung des T-Modells darstellt. Hierbei verfügt die Person über zwei oder mehr Spezialisierungsbereiche, die durch mehrere vertikale Balken symbolisiert werden. Neben diesen fachlichen Kompetenzen sind auch analytisches Denken und ein systematisches Vorgehen essenzielle Fähigkeiten, um die Anforderungen dieser Rolle zu erfüllen. Eine weitere wichtige Fähigkeit ist die Abstraktionsfähigkeit. Diese ermöglicht es, von konkreten Anforderungen zu abstrahieren und zu analysieren, welche übergeordneten Bedürfnisse tatsächlich bestehen. Ebenso ist kritisches Denken von großer Bedeutung. Darüber hinaus sind Kreativität und eine gewisse Erfahrung unerlässlich. Ohne ausreichende Erfahrung wird es schwierig, fundierte Entscheidungen zu treffen. In manchen Fällen spielt auch ein gewisses Bauchgefühl eine Rolle, das in bestimmten Situationen durchaus relevant sein kann.

Zudem sollte eine Architektin oder ein Architekt mit den methodischen Ansätzen und den verschiedenen Phasen, die in diesem Kontext von Bedeutung sind, vertraut sein und diese systematisch anwenden können. Dabei geht es nicht darum, die Phasen starr und linear abzuarbeiten, sondern vielmehr darum, situationsabhängig zu erkennen, welche Tätigkeiten und Aktivitäten in einem bestimmten Moment sinnvoll sind, um das Projekt voranzubringen.

Schließlich sind Kommunikationsfähigkeiten und soziale Kompetenzen essenziell. Diese sind notwendig, um effektiv mit dem Team sowie mit den verschiedenen Stakeholdern, die im Projekt eine Rolle spielen, zu interagieren. In diesem Zusammenhang wird häufig auf die Übersicht und die methodischen Ansätze verwiesen, die beispielsweise von Starke beschrieben wurden.

[page 071] | *page_071.png*

Ein Softwarearchitekt oder eine Softwarearchitektin hat eine Vielzahl von Aufgaben, die unterschiedliche Aspekte umfassen. Dazu gehören organisatorische Aspekte, das Entwerfen von Systemstrukturen, die Berücksichtigung technischer Anforderungen sowie die Analyse nicht-funktionaler Anforderungen. All diese Tätigkeiten machen die Rolle eines Softwarearchitekten oder einer Softwarearchitektin aus. Man kann diese Rolle als eine Kombination aus Diplomat und Akrobat verstehen.

[page 072] | *page_072.png*

Ein Softwarearchitekt bewegt sich in einem Spannungsfeld, das durch verschiedene Anforderungen und Einschränkungen geprägt ist. Dieses Spannungsfeld umfasst unter anderem Kosten, Zeitvorgaben und Qualitätskriterien. Zu den Qualitätskriterien zählen beispielsweise die Performance des Systems, die Fähigkeit, schnell auf Anfragen zu reagieren, die Einfachheit der Nutzung, die Flexibilität für zukünftige Erweiterungen sowie die Wartbarkeit, um Fehler effizient beheben zu können. Die Herausforderung besteht darin, eine ausgewogene Lösung zu finden, die all diese Anforderungen bestmöglich erfüllt. Dieser Balanceakt stellt den akrobatischen Teil der Tätigkeit dar.

Gleichzeitig muss die gefundene Lösung gegenüber verschiedenen Stakeholdern kommuniziert und vertreten werden. Dazu gehören unter anderem Kunden, Projektleiter, Betriebsteams, andere Abteilungen und das Entwicklungsteam. Diese kommunikative und vermittelnde Aufgabe entspricht dem diplomatischen Anteil der Rolle. Die Metapher des Architekten als Diplomat und Akrobat veranschaulicht diese duale Verantwortung auf anschauliche Weise.

[page 073] | *page_073.png*

Bei der Betrachtung der Kommunikation mit Stakeholdern ist es wichtig, zunächst zu klären, welche Stakeholder im Kontext der Softwarearchitektur eine Rolle spielen. Eine Übersicht kann dabei helfen, die verschiedenen Akteure und ihre Beziehungen zum Architekten zu verdeutlichen. In diesem Zusammenhang wird der Architekt oft als zentrale Figur dargestellt, wobei diese Darstellung keine Priorisierung oder Gewichtung der Stakeholder impliziert, sondern lediglich der Veranschaulichung dient.

Ein zentraler Stakeholder ist das Entwicklungsteam. Dieses begleitet der Architekt während der Umsetzung der Architektur. Eine wesentliche Aufgabe besteht darin, dem Team die Architektur zu vermitteln, damit die definierten Regeln und Schnittstellen eingehalten werden. Ohne die aktive Mitwirkung des Entwicklungsteams kann die geplante Architektur nicht realisiert werden. Daher

ist das Entwicklungsteam ein essenzieller Partner, dessen Bedeutung stets berücksichtigt werden muss.

Ein weiterer wichtiger Aspekt sind die Anforderungen, die häufig in Zusammenarbeit mit dem Fachbereich oder dem Product Owner erarbeitet werden. Der Architekt steht in enger Kommunikation mit diesen Stakeholdern, um die Anforderungen zu klären und zu präzisieren. Diese Interaktion bildet die Grundlage für die Entwicklung einer Architektur, die den fachlichen und geschäftlichen Bedürfnissen gerecht wird. Ein weiterer zentraler Aspekt der Arbeit eines Architekten ist die enge Zusammenarbeit mit der Projektleitung. Diese Kooperation ist essenziell, um bei schwierigen Entscheidungen, Zeitverzögerungen oder anderen Herausforderungen beratend und unterstützend tätig zu sein. Der Architekt trägt dabei nicht nur zur Lösung solcher Probleme bei, sondern übernimmt auch eine inhaltliche Mitverantwortung für die Leitung des Projekts.

Darüber hinaus spielt die Unternehmensarchitektur eine bedeutende Rolle. Wie bereits im zweiten Kapitel erwähnt, entsteht Architektur auf verschiedenen Ebenen, und die Unternehmensarchitektur betrachtet alle Systeme eines Unternehmens in ihrer Gesamtheit. In diesem Kontext gibt es oft Verantwortliche, die unternehmensweite Richtlinien und Konventionen entwickeln, die für alle Systeme verbindlich sind. Ein Architekt muss diese Vorgaben kennen, einhalten und aktiv an ihrer Weiterentwicklung mitwirken.

Häufig existieren in Unternehmen auch sogenannte Architekturboards. Diese Gremien ermöglichen es Architekten, gemeinsam und kollaborativ Entscheidungen über neue Richtlinien oder Änderungen zu treffen. Ein weiterer wichtiger Stakeholder ist der Betrieb. Damit die entwickelte Software erfolgreich eingesetzt werden kann, ist es notwendig, die Anforderungen des Betriebs zu berücksichtigen und die Software entsprechend zu spezifizieren. Welche Anforderungen bestehen an die Hardware und die Infrastruktur? Diese Aspekte müssen mit dem Betrieb abgestimmt werden, um sicherzustellen, dass der Betrieb die entwickelte Software effektiv verwalten und überwachen kann. Es muss gewährleistet sein, dass der Betrieb in der Lage ist, den Status der Software zu überprüfen, Probleme zu identifizieren und geeignete Maßnahmen zu ergreifen, beispielsweise bei infrastrukturellen Herausforderungen wie begrenztem Speicherplatz. In der heutigen Zeit, insbesondere mit der Verlagerung vieler Systeme in die Cloud, sind solche Themen oft weniger komplex, da viele Aufgaben durch die Cloud-Dienste selbst übernommen werden. Dennoch existieren in einigen Unternehmen weiterhin dedizierte Betriebsabteilungen, auch wenn in modernen Ansätzen wie DevOps die Verantwortung für den Betrieb zunehmend auf die Entwicklungsteams übergeht. In vielen Szenarien gibt es jedoch nach wie vor eine zentrale Betriebsabteilung oder eine Plattform, mit der die Entwicklungsteams interagieren und sich abstimmen müssen.

In diesem Zusammenhang spielen auch betriebliche Richtlinien, insbesondere im Bereich der IT-Sicherheit, eine wichtige Rolle. Diese Vorgaben müssen bekannt sein, eingehalten werden und erfordern eine enge Abstimmung mit den zuständigen Stellen im Betrieb.

Es wird deutlich, dass in einem Softwareprojekt eine Vielzahl von Stakeholdern involviert ist. Neben den bereits genannten Hauptakteuren können weitere Beteiligte eine Rolle spielen, wie beispielsweise Qualitätssicherungsteams oder Tester, mit denen eine enge Zusammenarbeit erforderlich ist. Darüber hinaus kann es zentrale Einheiten für das Wissensmanagement geben, zu denen die Projektbeteiligten aktiv beitragen und ihr Wissen weitergeben müssen. In jedem Softwareprojekt ist es von zentraler Bedeutung, die relevanten Stakeholder zu identifizieren und

deren Rollen sowie Einfluss auf das Projekt zu analysieren. Dabei sollte nicht nur erfasst werden, welche Stakeholder beteiligt sind, sondern es ist ebenso wichtig, diese aktiv zu managen. Ein effektives Stakeholder-Management kann durch den Einsatz einer Matrix unterstützt werden, die eine strukturierte Herangehensweise ermöglicht.

[page 074] | *page_074.png*

Eine solche Matrix basiert auf zwei Dimensionen: Zum einen wird betrachtet, wie stark der jeweilige Stakeholder in das Projekt involviert ist, und zum anderen, wie stark sich das Ergebnis des Projekts – in diesem Fall die entwickelte Software – auf den Stakeholder auswirkt. Diese beiden Dimensionen erlauben eine Einteilung der Stakeholder in vier Quadranten, die jeweils unterschiedliche Strategien für den Umgang mit den Stakeholdern erfordern.

Ein Beispiel für diese Einteilung ist der Quadrant links unten, in dem Stakeholder mit geringer Projektbeteiligung und geringem Einfluss des Projektergebnisses auf sie eingeordnet werden. Für diese Gruppe können spezifische Strategien entwickelt werden, die ihrem geringen Einfluss und ihrer geringen Beteiligung Rechnung tragen. Stakeholder, die nur wenig in das Projekt involviert sind und auf die das Projektergebnis kaum Auswirkungen hat, können beispielsweise entfernte Software-Systeme oder Teams betreffen, die ein anderes System betreiben und nur am Rande mit dem Projekt in Berührung kommen. Für diese Gruppe ist in der Regel keine intensive Betreuung erforderlich. Es genügt, grundlegende Informationen bereitzustellen, um sie im Bedarfsfall zu informieren. Dies wird als reine Informationsstrategie bezeichnet.

Im Gegensatz dazu gibt es Stakeholder, wie beispielsweise Fachbereiche, die idealerweise eine sehr hohe Beteiligung am Projekt aufweisen sollten. Andernfalls könnte dies zu erheblichen Problemen führen. Die Auswirkungen des Projektergebnisses auf diese Stakeholder sind in der Regel enorm, da sie direkt mit der entwickelten Software arbeiten. Daher ist es essenziell, mit diesen Stakeholdern enge Kooperationen einzugehen, sie von der Richtigkeit der Projektentscheidungen zu überzeugen und sie kontinuierlich in den Prozess einzubinden. Diese Stakeholder fallen in den Quadranten unten rechts der Matrix.

Ein weiterer Quadrant umfasst Stakeholder, die zwar stark in das Projekt involviert sind, auf die das Projektergebnis jedoch nur geringe Auswirkungen hat. Diese Stakeholder können als Multiplikatoren genutzt werden. Es ist wichtig, sie zu überzeugen und einzubinden, damit sie die Projektziele und -ergebnisse weitertragen und unterstützen.

Der letzte Quadrant betrifft Stakeholder, die nur gering in ein Projekt involviert sind, auf die das Projektergebnis jedoch eine hohe Auswirkung hat, stellen eine besondere Herausforderung dar. Ein Beispiel hierfür könnte ein Software-System-Team sein, das noch nicht direkt in das Projekt integriert ist, jedoch in Zukunft eine wichtige Rolle spielen wird. Ebenso kann es vorkommen, dass ein Fachbereich nicht die gewünschte Beteiligung zeigt. In solchen Fällen ist es entscheidend, diese Stakeholder stärker einzubinden, sie von der Bedeutung ihrer Mitwirkung zu überzeugen und sie entsprechend zu befähigen, ihren Beitrag leisten zu können. Ziel ist es, diese Stakeholder in der Matrix in Richtung einer höheren Beteiligung zu verschieben.

[page 075] | *page_075.png*

Im Kontext von Projekten spiegelt sich diese Dynamik auch in den Entwicklungsprozessen wider. In klassischen Projekten war es häufig so, dass eine einzelne Person die Rolle des Architekten oder

der Architektin innehatte und die volle Verantwortung für die Architekturentscheidungen trug. Dies bedeutete auch, dass diese Person für etwaige Fehlentscheidungen zur Rechenschaft gezogen wurde.

In kleineren Projekten war es oft ein einzelner Architekt oder eine einzelne Architektin, die diese Verantwortung übernahm. In größeren Projekten hingegen wurde in der Regel ein Chefarchitekt oder eine Chefarchitektin eingesetzt, der oder die ein Team von Architekten und Architektinnen leitete. Diese Teammitglieder waren dann jeweils für spezifische Teilprojekte verantwortlich. Der Chefarchitekt oder die Chefarchitektin stand in regelmäßigem Austausch mit der Projektleitung, um die Koordination und Abstimmung sicherzustellen. Die Aufgabe eines Architekten oder einer Architektin besteht darin, Regeln und Richtlinien zu definieren, die von den Entwicklerteams umgesetzt werden sollen. Zudem wird sichergestellt, dass die Ergebnisse der Teams den definierten Vorgaben entsprechen. Dies schließt idealerweise auch eine Qualitätssicherung (QA) ein, um nicht nur Abweichungen vom Design zu identifizieren, sondern auch mögliche Schwachstellen oder Fehlerquellen im Prozess zu erkennen und entsprechende Rückschlüsse zu ziehen.

Ein weiterer Aspekt der Arbeit eines Architekten oder einer Architektin ist die Strukturierung und Zuweisung von Arbeitspaketen. Diese Arbeitspakete werden in den Projektplan integriert und detailliert ausgearbeitet, um festzulegen, welche Aufgaben in den jeweiligen Paketen enthalten sind. Dies ist ein wesentlicher Bestandteil der Projektplanung und -steuerung.

Idealerweise ist ein Architekt oder eine Architektin auch aktiv in die Arbeit des Teams eingebunden, wobei ein gewisser Anteil der Arbeitszeit für übergreifende Aufgaben reserviert bleibt. Dazu gehört die Abstimmung mit anderen Teams, die Sicherstellung der Kohärenz zwischen den verschiedenen Teamaktivitäten sowie die übergeordnete Steuerung des Projekts. In dieser Rolle fungiert der Architekt oder die Architektin auch als Berater oder Beraterin für das Team und stellt sicher, dass ausreichend Kapazitäten für diese Aufgaben vorhanden sind.

Ein Architekt oder eine Architektin kann auch die Rolle eines Coaches für die Entwickler übernehmen. Es ist jedoch ebenso möglich, dass die Person in einer eher passiven Rolle agiert, indem sie lediglich an Meetings teilnimmt und punktuell als Sparringspartner unterstützt. Beispielsweise könnte ein Teammitglied bei der Auswahl einer neuen Bibliothek auf die Expertise des Architekten oder der Architektin zurückgreifen, um eine fundierte Entscheidung zu treffen. Diese beratende Funktion wird durch die umfassende Kenntnis des Projekts und der Architektur ermöglicht, die der Architekt oder die Architektin von Beginn an begleitet und mitverantwortet hat. Dadurch kann er oder sie die Auswirkungen von Entscheidungen auf die Gesamtarchitektur besser einschätzen und gezielt steuern.

Ein wesentlicher Bestandteil dieser Beratungsfunktion ist die Identifikation und Berücksichtigung von Risiken. Mit wachsender Erfahrung ist es oft leichter, potenzielle Risiken frühzeitig zu erkennen und das Team darauf aufmerksam zu machen. Zudem kann der Architekt oder die Architektin dabei helfen, verschiedene Optionen zu bewerten und fundierte Entscheidungen zu treffen.

Letztlich fungiert der Architekt oder die Architektin als Enabler für das Team, indem er oder sie Methoden und Arbeitsweisen fördert, die ein strukturiertes und effizientes Vorgehen ermöglichen. Dazu gehört beispielsweise die Bereitstellung von Werkzeugen wie Dashboards, die Qualitätsmetriken visualisieren. Solche Instrumente helfen dem Team, die Auswirkungen ihrer Arbeit besser zu verstehen und notwendige Anpassungen eigenständig vorzunehmen. Auch die

Vermittlung von Wissen und die Förderung methodischer Kompetenzen sind zentrale Aufgaben, die zur Weiterentwicklung des Teams und zur Sicherstellung der Qualität der entwickelten Software beitragen. Nach der Einführung sollte nun ein grundlegendes Verständnis darüber bestehen, worum es in diesem Kapitel geht. Es wurde erläutert, was ein Informationssystem ist, welche Art von Systemen eine Software-Architektur erfordern, was unter einer Software-Architektur zu verstehen ist, welchen Zweck sie erfüllt und welche grundlegenden Methoden und Vorgehensweisen zur Erstellung einer solchen Architektur existieren. Zudem wurde darauf eingegangen, an welcher Stelle im Entwicklungsprozess diese Methoden sinnvoll eingesetzt werden können und wie sie in den Prozess integriert sind. Abschließend wurde die Rolle des Architekten oder der Architektin thematisiert, einschließlich der unterschiedlichen Ausprägungen dieser Rolle, die stark vom jeweiligen Softwareentwicklungsprozess abhängen.

[page 076] | *page_076.png*

Zum Abschluss dieses Kapitels wird ein kurzer Überblick über die historische Entwicklung der Software-Architektur gegeben. Dieser Überblick ist nicht als formale oder umfassende Darstellung zu verstehen, sondern soll lediglich eine grobe Orientierung bieten, um die Ursprünge und die Entwicklung des Themas besser einordnen zu können.

[page 077] | *page_077.png*

Die Geschichte der Software-Architektur beginnt bereits in den 1960er Jahren, einer Zeit, die von der sogenannten „Softwarekrise“ geprägt war. In dieser Ära wurden vor allem Großrechner- und Host-Systeme entwickelt. Ein bekanntes Beispiel ist ein System von IBM, das mit erheblichen Lieferverzögerungen von mehreren Jahren zu kämpfen hatte. Diese Probleme machten deutlich, dass die zunehmende Komplexität der Systeme die damaligen Entwicklungsmethoden überforderte und eine neue Herangehensweise erforderlich war, um die gewünschte Funktionalität erfolgreich umsetzen zu können. Ein Beispiel für eine damals weit verbreitete, jedoch problematische Technik ist die Verwendung des „GOTO“-Statements. Im Jahr 1968 veröffentlichte Edsger W. Dijkstra einen einflussreichen Aufsatz mit dem Titel „Go To Statement Considered Harmful“. Das GOTO-Statement erlaubte es, den Programmfluss an eine beliebige andere Stelle im Code zu springen, indem man auf ein zuvor definiertes Label verwies. Dies führte dazu, dass der Programmcode schwer nachvollziehbar und unübersichtlich wurde, da die Kontrollflüsse nicht mehr klar strukturiert waren.

Die uneingeschränkte Nutzung von GOTO-Statements resultierte in einem Abhängigkeitsgraphen, der kaum noch zu durchschauen war. Dies verdeutlicht die Notwendigkeit, Abhängigkeiten im Programmcode so zu gestalten, dass sie verständlich und wartbar bleiben. Dijkstra empfahl daher, die Verwendung von GOTO-Statements stark einzuschränken. Insbesondere sollte ein GOTO-Statement, wenn überhaupt, nur am Ende eines Kontrollflusses eingesetzt werden, um die Nachvollziehbarkeit und Wartbarkeit des Codes zu gewährleisten. Diese Überlegungen markieren einen frühen Schritt in der Entwicklung von Prinzipien, die auf eine klare und strukturierte Gestaltung von Software abzielen. Ein weiteres einflussreiches Paper aus dieser Zeit, veröffentlicht 1972 von David L. Parnas, befasste sich mit der Zerlegung von Systemen in Module. Parnas stellte die Frage, wie man die Komplexität von Systemen durch Modularisierung beherrschen kann und wie diese Module sinnvoll zu schneiden sind. In seinem Artikel führte er das Konzept des „Information Hiding“ ein, das auch als Geheimnisprinzip bekannt ist, und postulierte es als zentrales Kriterium für die Modularisierung.

Das Geheimnisprinzip besagt, dass Module so gestaltet werden sollten, dass jedes Modul ein bestimmtes „Geheimnis“ bewahrt. Dieses Geheimnis bezieht sich auf Details der Implementierung, die vor anderen Modulen verborgen bleiben. Dadurch wird sichergestellt, dass Änderungen an der internen Implementierung eines Moduls vorgenommen werden können, ohne dass andere Module davon betroffen sind. Diese Idee fördert die Flexibilität und Wartbarkeit eines Systems, da die Abhängigkeiten zwischen den Modulen minimiert werden. Der Artikel von Parnas ist bis heute von großer Bedeutung, da die darin beschriebenen Prinzipien weiterhin als grundlegende Leitlinien für die Strukturierung von Software gelten. Sie sind besonders relevant, wenn es darum geht, ein System in Abstraktionen zu unterteilen, die es ermöglichen, Designentscheidungen hinter einer klar definierten Schnittstelle flexibel zu halten.

Parnas veranschaulichte seine Überlegungen anhand eines konkreten Beispiels. In den 1980er Jahren gewann dann die Objektorientierung an Bedeutung. Diese wurde von vielen als eine natürliche Weiterentwicklung der Softwarearchitektur betrachtet, da sie darauf abzielt, die reale Welt durch Objekte zu modellieren. Die Objektorientierung wurde oft als ein Ansatz gesehen, der die Prinzipien der Modularisierung und des Information Hiding aufgreift und weiterführt, indem sie die Strukturierung von Software entlang der Konzepte von Objekten und deren Interaktionen ermöglicht. In den folgenden Jahren wurde jedoch deutlich, dass die zuvor beschriebenen Ansätze und Konzepte nicht ausreichten, um die Komplexität moderner Softwaresysteme vollständig zu bewältigen. Erst in den 1990er Jahren etablierte sich der Begriff der Software-Architektur als eigenständiges und anerkanntes Konzept in der Fachliteratur. Zu dieser Zeit begann man auch, systematisch Muster zu identifizieren und zu dokumentieren, die bei der Gestaltung von Software hilfreich sein können.

Ein grundlegendes Werk in diesem Zusammenhang ist das Buch „Design Patterns: Elements of Reusable Object-Oriented Software“, das von der sogenannten „Gang of Four“ (GoF) verfasst wurde. Der Name „Gang of Four“ leitet sich von den vier Autoren des Buches ab, die sich auf einem Symposium und bei weiteren Konferenzen zusammenfanden. In diesem Werk wurden Design Patterns beschrieben, die auf der objektorientierten Architektur basieren. Diese Muster bieten bewährte Lösungen für wiederkehrende Probleme in der Softwareentwicklung und zeigen, wie bestimmte Herausforderungen auf eine konsistente und effektive Weise gelöst werden können.

Die Bedeutung von Design Patterns geht jedoch über die objektorientierte Programmierung hinaus. Sie sind auch auf der Ebene der Software-Architektur von Relevanz, da sie helfen, wiederkehrende Probleme in der Strukturierung und Organisation von Systemen zu adressieren. Im weiteren Verlauf wird auf diese Design Patterns und ihre Anwendung in der Software-Architektur noch näher eingegangen.

Der Begriff der „Komponente“ und die damit verbundene komponentenorientierte Architektur gewannen erst um das Jahr 2000 an Bedeutung. Erst zu diesem Zeitpunkt wurde die systematische Gestaltung von Software auf Basis von Komponenten zu einem zentralen Thema in der Software-Architektur. Die Entwicklung und der Einsatz von komponentenbasierten Architekturen haben seitdem erheblich zur Modularität und Wiederverwendbarkeit von Software beigetragen. Zwischen 2003 und 2010 gab es Bestrebungen, die Prinzipien der komponentenorientierten Architektur auf die Ebene von Anwendungslandschaften zu übertragen. Dies führte zur Entwicklung von serviceorientierten Architekturen (Service-Oriented Architectures, SOA). Allerdings kam diese Entwicklung etwa um das Jahr 2010 weitgehend zum Stillstand. In derselben Zeit begann jedoch der Aufstieg des Cloud Computing, das insbesondere in jüngerer Vergangenheit einen erheblichen Einfluss auf die Software-Architektur ausgeübt hat.

Ab etwa 2011 entstanden die Konzepte der Microservices, die ebenfalls einen bedeutenden Einfluss auf die Gestaltung von Softwaresystemen hatten. Interessanterweise führte die Begeisterung für das Microservices-Paradigma dazu, dass viele der Prinzipien und Ansätze, die bereits in den komponentenorientierten Architekturen der frühen 2000er Jahre etabliert worden waren, in den Hintergrund gerieten. Ein Beispiel hierfür ist das Domain-Driven Design, das 2004 eingeführt wurde, aber mit dem Aufkommen der Microservices teilweise vernachlässigt wurde, da die Aufmerksamkeit stark auf die neuen Muster und Ansätze gerichtet war.

Im Laufe der Zeit zeigte sich jedoch, dass die unreflektierte Anwendung von Microservices auch zu zahlreichen Fehlschlägen führte. Um 2018 wurde zunehmend erkannt, dass es sinnvoller ist, modulare Architekturen zu entwickeln, die auf den Prinzipien der komponentenorientierten Architektur basieren. Diese modularen Architekturen, die im Wesentlichen eine Weiterentwicklung der komponentenorientierten Ansätze darstellen, werden im weiteren Verlauf des Kapitels genauer betrachtet, um zu analysieren, welche Konzepte und Prinzipien dabei von besonderer Bedeutung sind. Wie lässt sich eine sinnvolle Zerlegung in Bausteine und Komponenten erreichen? Diese Fragestellung ist zentral und wird im weiteren Verlauf der Vorlesung behandelt.

[page 078] | *page_078.png*

Zusammenfassend lässt sich festhalten, dass Software-Architektur im Wesentlichen der Bewältigung von Komplexität dient. Sie basiert auf den Anforderungen und hat die Aufgabe, diese zu erfüllen. Es existieren verschiedene Definitionen von Architektur, die bekannt sein sollten. Ein zentraler Aspekt ist das Konzept von Komponenten und Bausteinen sowie die Abhängigkeiten zwischen diesen, die durch klar definierte Schnittstellen beschrieben werden.

Ein weiterer wesentlicher Punkt ist, dass Architektur von einer spezifischen Rolle getragen wird. Diese Rolle erfordert eine Kombination aus verschiedenen Fähigkeiten, die oft mit Begriffen wie „Akrobat“ oder „Diplomat“ beschrieben werden, um die Vielseitigkeit und Komplexität der Aufgaben zu verdeutlichen.

Darüber hinaus ist es wichtig, den Unterschied zwischen Architektur in klassischen und agilen Projekten zu verstehen. Ein grundlegendes Verständnis des Entwurfsprozesses sollte ebenfalls vorhanden sein. In den folgenden Kapiteln wird dieser Prozess schrittweise vertieft, um ein fundiertes Verständnis zu vermitteln.

Abschließend sei angemerkt, dass die zuvor dargestellten historischen Entwicklungen und Zeitpunkte zwar interessant sind, jedoch nicht als prüfungsrelevant betrachtet werden.