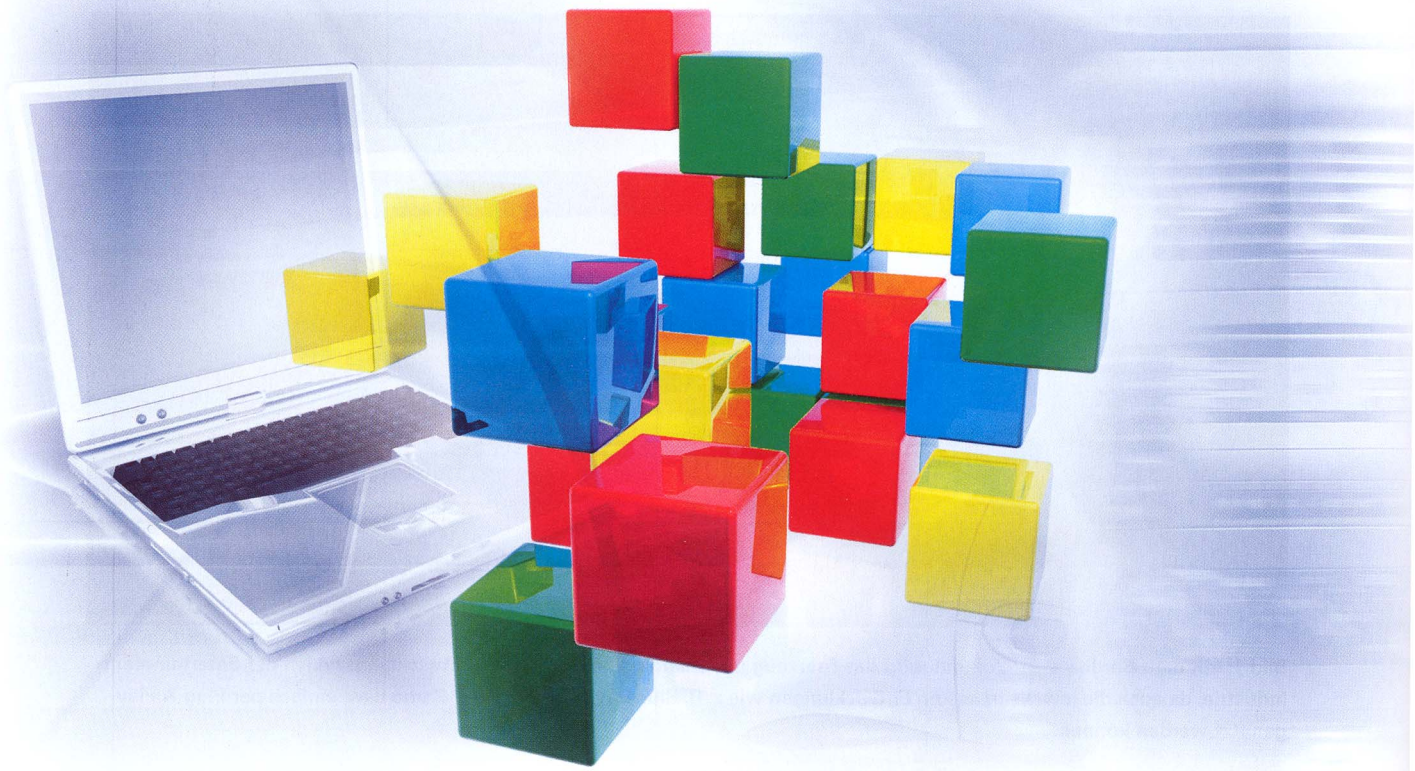

APPLIKATIONSSOFTWARE OPTIMAL IN KOMPONENTEN SCHNEIDEN

Qualitäts-Software-Architektur

Basierend auf der Metapher der „Software-Blutgruppen" definiert dieser Beitrag Regeln für den optimalen Zuschnitt der Anwendungssoftware in Komponenten und Schnittstellen. Hierbei kommen die Prinzipien von Quasar – dem Architekturparadigma von sd&m – zur Anwendung, in das die Erfahrungen aus über tausend Softwareentwicklungsprojekten eingeflossen sind.

Der stark steigende Softwareanteil im Automobil hat zu einer für die Automobilindustrie schwer beherrschbaren Komplexität und zu explodierenden Entwicklungskosten bei der Fahrzeugentwicklung geführt. Um Abhilfe zu schaffen, setzen moderne Architekturkonzepte für E/E-Komponenten auf Softwarekomponenten mit klar definierten Schnittstellen. Allgemeine technische Basisdienste wie Buskommunikation oder Speicherverwaltung werden zunehmend standardisiert. Sie helfen jedoch nicht bei der Identifikation und Strukturierung von Komponenten auf der Applikationsebene. Andererseits kommt genau dieser Fragestellung immer größere Bedeutung zu, da die wettbewerbsdifferenzierende fachliche Funktionalität immer komplexer wird.

Grundlagen von Quasar (Qualitäts-Software-Architektur)

Die Komponente ist die wesentliche Einheit des Entwurfs, der Implementierung und der Planung. Sie kann sich aus einer beliebigen Anzahl von Modulen (Quellcode-Dateien) zusammensetzen und kommuniziert über definierte Schnittstellen. Durch Komposition entstehen hierarchische Komponenten. Die Software-Blutgruppen gelten als Weiterentwicklung des Konzeptes der Trennung der Zuständigkeiten. Sie kategorisieren Komponenten und Schnittstellen und helfen dadurch fachlichen und technischen Code zu trennen, da diese jeweils eigene Zuständigkeiten, Abhängigkeiten und Änderungszyklen haben. Jede Schnittstelle und nicht hierarchische Komponente sollte zu genau einer dieser Blutgruppen gehören:

```
void showNaviTargetScreen()
{
    Button startButton=framework.drawButton(„Start route");
    framework.installClickHandler(startButton, &calculate-
Route, &showNaviScreen);
```

Bild 1: Codefragment im MMI eines Navigationssystems als Negativ-Beispiel.

- 0-Software ist neutral, unabhängig von Anwendung und Technik (z. B. mathematische Bibliotheken). Sie ist ideal wiederverwendbar, für sich alleine aber ohne Nutzen.
- A-Software ist bestimmt durch die Anwendung, aber unabhängig von der Technik. Sie definiert und implementiert die Anwendungsfälle (z. B. Fahrerassistenz).
- T-Software ist unabhängig von der Anwendung und bestimmt durch die Technik bzw. durch die technischen APIs (z. B. MOST-API).
- R-Software transformiert fachliche Objekte in externe Repräsentationen (z. B. Nachrichtenformat) und wieder zurück.

Durch Mischung von 0 und A beziehungsweise T entsteht wieder A beziehungsweise T. Mischt man A und T, so entsteht die „unreine" Blutgruppe AT: Diese ist ein sowohl von der Technik als auch von der Fachlichkeit abhängiger Code. Erfahrungsgemäß ist dieser Code unübersichtlich und schwer wartbar, daher sollte er vermieden werden.

Zehn Designrichtlinien für den optimalen Zuschnitt von Komponenten

Auch für Software im Auto bringt Quasar signifikanten Mehrwert. Dazu sollten folgende Regeln beachtet werden:

- Komponentenkategorien identifizieren: Zuerst bestimmen zu welchen Blutgruppen die einzelnen Teile der Anwendung gehören und darauf aufbauend die Komponenten identifizieren, die darin vorkommen können.
- Trennung der Sichten auf Komponenten: Beim Design benötigt man zunächst nur die Außensicht der Komponenten (kurze Beschreibung, importierte/exportierte Schnittstellen). Die Innensicht einer Komponente ist lediglich für die zuständigen Entwickler relevant.
- Bruchlose Umsetzung von Anwendungsfällen: Die im Lastenheft definierten und informell beschriebenen Anwendungsfälle und Datentypen werden in der Zielsprache ausformuliert, bleiben aber so technikfrei, wie sie es in der Spezifikation waren.
- Kommunikation über Schnittstellen: Die zu einer Komponente gehörenden Module können sich direkt aufrufen, Module unterschiedlicher Komponenten dürfen nur über Schnittstellen kommunizieren. A-Komponenten dürfen untereinander nur mittels A- oder 0-Schnittstellen, mit Technikkomponenten nur über R- oder 0-Schnittstellen kommunizieren.
- Zentrale Funktionalitäts- und Datenhoheit: Anwendungsfälle sollten nicht über mehrere Komponenten verteilt werden. Dies gilt insbesondere für die Verwal-

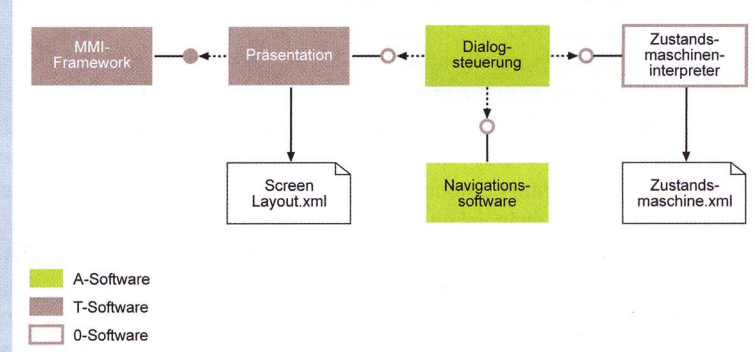

Bild 2: Trennung der Blutgruppen im MMI des Navigationssystems.

ABBILDUNG 3

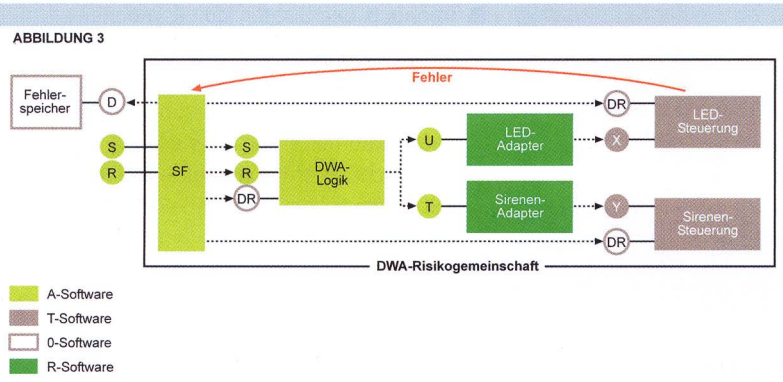

Bild 3: Fehlerbehandlung mit Sicherheitsfassade in einer Diebstahlwarnanlage.

tung von Daten (z. B. Codierung) und Zuständen (z. B. Betriebszustand).

- Umgang mit Variabilität: Anwendungsfälle, bei denen Änderungen zu erwarten sind (z. B. neue Regel-Algorithmen) oder von denen es Varianten (z. B. für US oder EU) geben wird, müssen in eigene Komponenten strukturiert werden, damit die Änderung lokal bleibt.
- Trennung der Schnittstellen: Die Verwendungszwecke dürfen nicht vermischt werden. So sollten zum Beispiel Operationen, die für Testzwecke den Zugriff auf interne Variablen ermöglichen, in eine eigene Testschnittstelle ausgegliedert werden.
- Externe oder Legacy-Komponenten einbinden: Die Einbindung in die Anwendungsarchitektur sollte über eigene Adapter-Komponenten geschehen, um die technischen Abhängigkeiten zu kapseln.
- Standardisierte Komponenten (wieder-)verwenden: Wenn Komponenten gleiche Funktionen ausführen, sollten diese in eigene Utility-Komponenten ausgegliedert werden. Dies gilt sowohl für 0- oder T-Software (z. B. Zustandsautomaten, Fehlerbehandlung) als auch für komplette Anwendungsfälle (A-Software, z. B. Einklemmschutz).
- Fehlerbehandlung: Für jede Komponente müssen die möglichen Fehler beschrieben werden. Die Behandlungsstrategien (z. B. Re-Try) sollten dabei nicht auf mehrere Komponenten verteilt sein. Voneinander abhängige Komponenten lassen sich zu Risikogemeinschaften zusammenfassen und die Fehlerbehandlung in einer eigenen SF-Komponente (Sicherheitsfassade) realisieren. Dazu muss jede Komponente eine DR-Schnittstelle (Diagnose- und Reparatur) bereitstellen, mit der der Zustand abgefragt und beeinflusst werden kann.

Anwendung der Regeln in der Praxis

In einer naiven Implementierung des Mensch-Maschine-Interfaces eines Navigationssystems könnte das in **Bild 1** gezeigte Codefragment vorkommen. Es ist ein Mix aus fachlichen Funktionen (calculateRoute), technischen Funktionsaufrufen (drawButton) und dem Wissen über die Reihenfolge (vom NaviTargetScreen gelangt man zum NaviScreen). Diese Verwebung führt zu monolithischem AT-Code. Sollte zum Beispiel das eingesetzte MMI-Framework ausgetauscht oder neben der manuellen Bedienung auch Sprachsteuerung eingesetzt werden, wirkt sich dies auf die gesamte Software aus. Die Analyse der Blutgruppen führt hingegen zu einer klaren Trennung der Zuständigkeiten, wie in **Bild 2** dargestellt. In diesem Fall bewirkt ein Austausch des MMI-Frameworks lediglich eine Anpassung der Komponente Präsentation. Die Ergänzung um Sprachsteuerung als zusätzlichen Kanal kann mittels einer zweiten Präsentationskomponente gelöst werden.

Die Berücksichtigung von Fehlern führt oft dazu, dass eine elegante Komponentenarchitektur im Laufe der Zeit verunstaltet wird. Ein weiteres Beispiel zeigt daher den sinnvollen Einsatz einer Sicherheitsfassade in einer Diebstahlwarnanlage (DWA). Die Funktionalität der DWA teilt sich in drei Komponenten auf (**Bild 3**): Die DWA-Logik, die den Auslösealgorithmus implementiert, die Sirenen- und die LED-Steuerung. Jede dieser Komponenten bietet neben ihren fachlichen (S, R) und technischen (X, Y) auch eine DR-Schnittstelle an. Die Sicherheitsfassade implementiert dieselben Schnittstellen wie die DWA-Komponente und behandelt alle Fehler (inklusive Fehlerspeichereintrag). So wird die Fehlerbehandlung nicht auf verschiedene Komponenten verteilt, sondern zentral an einer Stelle zusammengefasst. Dadurch können die Komponenten einfacher programmiert werden, da die Entwickler in einer „heilen" Welt arbeiten können.

Fazit

Die Trennung von Anwendung und Technik kostet nur unwesentlich mehr Maschinenzyklen bei der Ausführung und etwas mehr Programmcode, verbessert aber nachhaltig die Wartbarkeit, Wiederverwendbarkeit und Robustheit der Anwendung. Die hier skizzierten Regeln leisten einen wesentlichen Beitrag zur klaren Anwendungsarchitektur und zum optimalen Zuschnitt von Komponenten für Software im Automobil. (oe)

Dr. Zoltán Ádám Mann ist Berater bei der sd&m AG. Als Mitglied des Center of Competence Fahrzeugentwicklung berät er Automobilhersteller und Zulieferer zu Fragen des Software-Engineering.

Dipl.-Inf. (FH) Marco Prucha ist bei sd&m als Senior-Softwareingenieur mit Schwerpunkt Automotive tätig und Leiter der Car-IT-Community.

sd&m AG
www.sdm.com