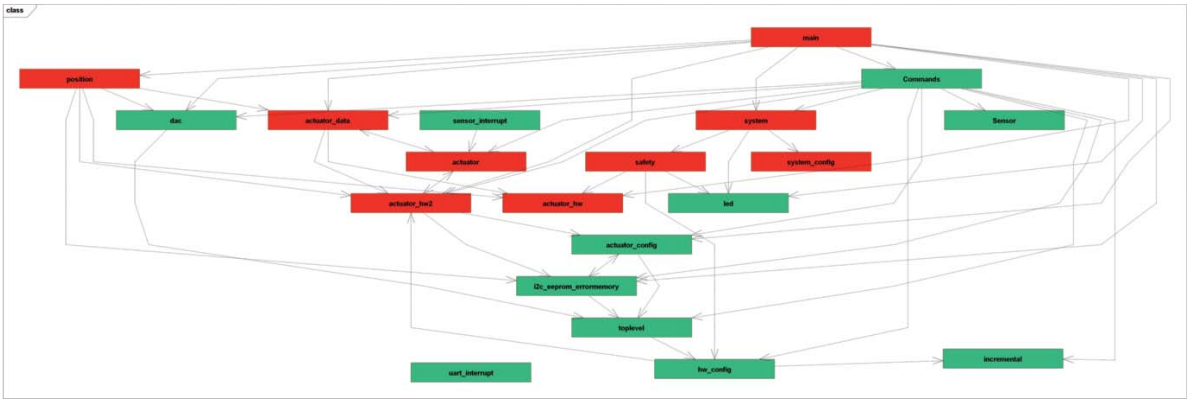


Bild: Tecmata GmbH

1 Beispiel für eine 'gewachsene' Architektur

SW-Architektur: unsichtbarer Qualitätsfaktor

Nie war die Konzeption einer Software-Architektur so wichtig für den Erfolg eines Software-Projektes. Sie ist der 'Klebstoff' zwischen den einzelnen Qualitätskomponenten. Viele Probleme haben hier ihren Ursprung. Entwurfsmuster helfen, Architekturen von Anfang an konsequent zu entwickeln oder zu verbessern.

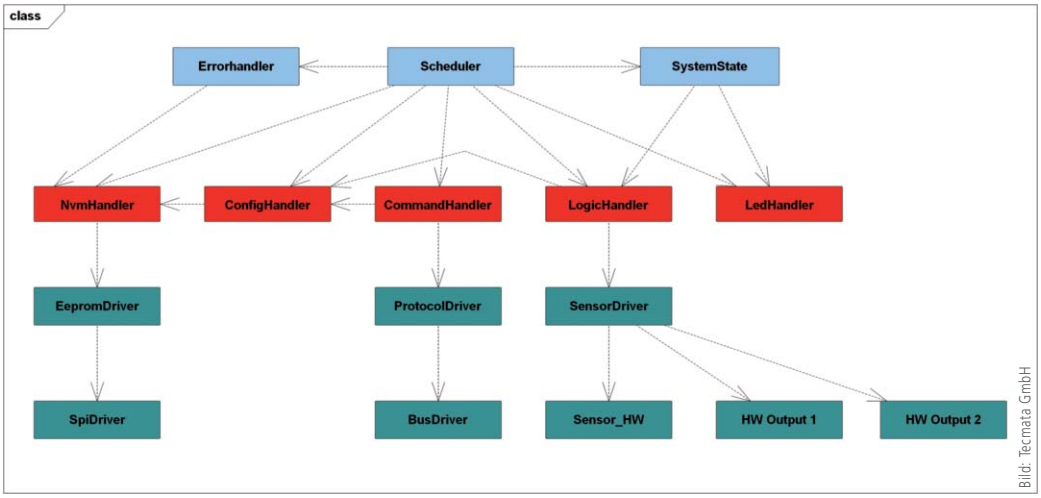
Autoren: Dipl. Math. Christian Geißelbach, Software Architect & Functional Safety Consultant und Dipl. Inf. (FH) Sascha Körner, Embedded Entwicklung & Functional Safety Professional, tecmata GmbH

Am Anfang steht die Entscheidung für ein Produkt und seine gewünschten Eigenschaften. Sie werden mittels eines Katalogs von Anforderungen dokumentiert, über Prozesse organisiert, entwickelt und anhand von Tests überprüft, sodass am Ende des Entwicklungsprozesses ein akzeptables Produkt entsteht. Neben den definierten Anforderungen, Prozessen und Tests nimmt die Software-Architektur einen wesentlichen Einfluss auf die Qualität eines Software-Produkts. Sie ist nicht nur Design-Artefakt und damit technisches Detail, sondern trägt wesentlich zur Produktqualität bei. Rein funktional ist der Wert einer guten Architektur dem fertigen Produkt nicht anzusehen – Software funktioniert oder nicht. Wie die Funktionalität intern zustande kommt ist für den Anwender nicht sichtbar, im Gegensatz zu fehlendem oder gar fehlerhaftem Verhalten. Architektur ist, über die technischen Inhalte hinaus, die Verbindung zwischen den Qualitätsfaktoren Codierung, Test und Dokumentation. Sie bestimmt die Komplexität des gesamten Entwurfs, und wirkt damit auf die gesamte Entwicklung ein. Eine gute Architektur teilt Software in kleine, unabhängige Einheiten ein, denen Anforderungen eindeutig zuzuordnen sind. Der offensichtlichste Einfluss der Architektur befindet sich während der Codierung. Je unabhängiger die einzelnen Teile sind, desto weniger Interaktion mit anderen Modulen ist zu beachten.

Anforderungen sicherheitskritischer Systeme

Eine derart gekapselte Funktionalität ist einfacher zu überblicken und deutlich weniger anfällig für Fehler während der Codierung. Insbesondere im Bereich sicherheitskritischer Embedded-Systeme ist eine geringe Komplexität ein Muss. Diese Systeme müssen besondere Anforderungen an Ausfallsicherheit und Zuverlässigkeit

Auszug aus der Fachzeitschrift Embedded Design, Ausgabe VI/2015.



2 Nach Anwendung der vier Designmuster entsteht eine übersichtliche Struktur mit deutlich geringer Komplexität.

erfüllen. Diese Robustheit kann nur erreicht und gegenüber Dritten – Kunden oder Prüfstellen wie dem TÜV – nachgewiesen werden, indem alle Abhängigkeiten und Auswirkungen zwischen Software-Modulen bekannt und vorhersagbar sind. Hohe Komplexität verdeckt dagegen oft subtile Unterschiede im Verhalten, mit der möglichen Konsequenz von Fehlverhalten bis hin zu Unfällen. In diesem Bereich sind vor Änderungen an der Software außerdem genaue Analysen der Auswirkungen der geplanten Modifikation Pflicht. Auch hier helfen Aufteilung und Kapselung, dass die Auswirkungen möglichst lokal begrenzt bleiben – was sich auch in der Zeit, welche für Codierung und Tests benötigt wird, positiv niederschlägt.

Effektives Testen

Der Softwaretest wird ebenfalls durch die Komplexität maßgeblich beeinflusst. Effektives Testen wird erst durch die klare Zuordnung von Anforderungen möglich. Eine gut gekapselte Softwarestruktur ermöglicht darüber hinaus, Teilsysteme frühzeitig zu testen. Damit können funktionale Fehler früher erkannt und behoben werden. Eine aufwendige und kostenintensive Fehlerbehebung nach Fertigstellung des Produkts wird damit minimiert. Geringe Abhängigkeiten helfen, Korrekturen lokal zu halten und somit den Aufwand für die Fehleranalyse zu reduzieren. Was macht nun eine gute Architektur aus? Eine etablierte Methode, um die Qualität von Architekturen und Design zu beurteilen sind Metriken wie z.B. von McCabe und Halstead. Sie versehen die Eigenschaften der Struktur mit Werten und machen diese so vergleichbar. Sie messen insbesondere, wie stark Funktionalitäten in abgeschlossenen Teilen der Software gekapselt

sind und, wie z.B. die Maße nach Briand, ob Kopplung und Kohäsion der Komponententeile wie gewünscht vorliegen. Der Nachteil dieser Metriken ist, dass bereits eine Architektur vorhanden sein muss, die analysiert wird. Außerdem bleibt die Frage, wie ein Design geändert werden muss um zu besseren Werten der Metriken zu gelangen. Was dazu notwendig ist, sind Richtlinien und Entwurfsmuster, die von vorneherein zu einer guten Architektur führen. Dieselben Muster können auch verwendet werden, um bestehende Systeme zu beurteilen und für den Einsatz im Bereich funktionaler Sicherheit fit zu machen.

Vier Merkmale qualitativer Software

Aus der Erfahrung heraus wissen wir, dass gute Software ganz konkret vier Merkmale aufweist: einfach, unabhängig, singulär und disjunkt. Mit der Einfachheit ist hier die inhaltliche Zuordnung von Aufgaben an ein Modul (c- und h-File) gemeint. Oftmals wird der Fehler begangen, zu viele Aufgaben einem Modul zuzuordnen. Als einfache Faustformel kann hierbei angestrebt werden, dass die Aufgabe eines Moduls mit zwei bis drei Sätzen beschrieben werden kann. Daraus ergibt sich, dass ein Modul lediglich für eine (Kern-) Aufgabe zuständig sein sollte. Wird dies berücksichtigt, bestehen nur wenige Abhängigkeiten zu anderen Modulen und die Integration ist sehr einfach zu realisieren. Die Unabhängigkeit geht mit der Abstraktion einher. Hierbei werden Module in klare Zuständigkeiten eingeteilt und die Funktionalität in kleine Teilaufgaben aufgeteilt. Jedes Modul für sich ist dabei für eine Teilaufgabe zuständig und muss bzw. darf nicht wissen, wie und was andere Module machen. Somit wird die Wiederverwendbarkeit stark erhöht. Ein Beispiel ist die Speicherung nicht flüchtiger Daten der Applikation. Es empfiehlt sich hierbei, das Datenmanagement von der verwendeten Technik zu trennen. Dazu dient in erster Linie ein Modul zum Daten-Mnagement, dessen Schnittstelle die Bedürfnisse der Applikation bedient. Das nächste Modul stellt eine Schnittstelle für den Zugriff auf den Speicherbaustein, z.B. ein EEPROM, zur Verfügung. Diese Schnittstelle weiß ihrerseits nichts über die Inhalte, sondern kann lediglich Daten an Adressen schreiben und von dort lesen. Als letztes Modul kommt ein Treiber ins Spiel, der die Kommunikation mit dem Speicher über eine konkrete Schnittstelle realisiert. Dadurch, dass die Module jeweils klar umrissene Aufgaben haben, sind sie verhältnismäßig klein und einfach zu realisieren. Zudem wird das Wissen über die Anwendung gekapselt, sodass Änderungen entweder die Technik oder die Logik betreffen – aber nicht beides zugleich. Das Prinzip der Singularität adressiert besonders die Variantenvielfalt, welche eingegrenzt werden sollte. Insbesondere soll verhindert werden, dass dieselbe Entscheidung an mehreren Stellen der Applikation redundant getroffen wird. Ein gutes Beispiel ist, wenn eine Software mit mehreren unterschiedlichen Sensoren für eine Messgröße umgehen können soll. Hier wäre es ein zu komplexes Design, direkt auf die konkreten Treiber der Sensoren zuzugreifen. Besser ist eine Zwischenschicht, die die Besonderheiten der Hardware kapselt und intern entscheidet, welcher Sensortreiber verwendet wird. Die restliche Applikation muss so nicht mehr wissen, dass es Hardwarevarianten gibt, und auf welcher sie aktuell ausgeführt wird. Bei der Disjunktheit geht es um die Verteilung von Daten. Meistens müssen Invarianten, d.h. immer gegebene Eigenschaften, auf Daten garantiert werden. Wenn mehrere Module ungeregelt auf Daten zugreifen, können diese Invarianten nur mit großem Aufwand aufrechterhalten werden. Wenn für die Modifikation der Daten dagegen nur genau ein Modul mit seiner Schnittstelle zuständig ist und ansonsten nur lesende Zugriffe erlaubt, ist die Konsistenz der Daten leicht zu gewährleisten. Ein Beispiel hierfür ist ein zentraler Fehlerspeicher, der die Datenhaltung organisiert. Würden alle Module ihre Fehler direkt selbst speichern, wäre die Gefahr von Inkonsistenzen größer.

Fazit

Wenn diese Prinzipien beachtet werden, gelangt man fast automatisch zu einem robusten und testbaren System. Auch existierende Systeme profitieren von diesen Regeln. Beim Scheitern von Erweiterungen liegt es meistens daran, dass die Software mindestens eines dieser Prinzipien missachtet. Mit Fokus auf diese Regeln kann der entsprechende Teil jedoch meist mit wenig Aufwand entzerrt und damit wieder erweiterbar gemacht werden. ■

www.tecmata.com

Auszug aus der Fachzeitschrift Embedded Design, Ausgabe VI/2015.