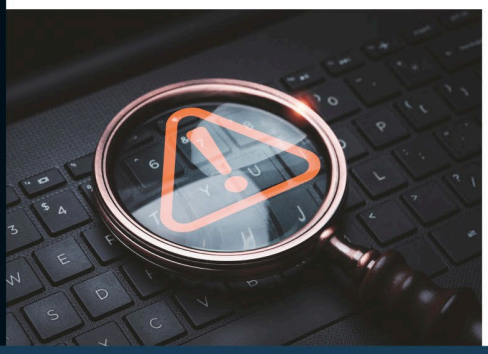
Derk Rembold

EXKLUSIVE LESEPROBE
technology research hub



Safety Engineering

Das Praxisbuch für funktionale Sicherheit

- Methoden für sichere und robuste Softwareentwicklung
- Betriebssicherheit herstellen und Fehler verstehen
- Gefahrenanalysen, Risikographen, Fehlerbaumanalysen, Zuverlässigkeitsblockdiagramme, Markov-Decision-Prozesse

Rheinwerk Computing

Kapitel 5
Softwaresicherheit

Software- und Hardwareentwicklung sollten in koordinierten Projektmanagementprozessen Hand in Hand gehen. Bei der Hardware kann durch die Architektur (z. B. Redundanz, Robustheit) ein System zuverlässig oder sicher gemacht werden. Bei der Software dienen dazu programmiertechnische Methoden. Es gibt beispielsweise Programmiermethoden, um Abstürze von Software zu vermeiden, z. B. durch den Verzicht auf die *Pointer*-Programmierung oder die Anwendung von *Smart-Pointern*.

Normen machen Vorschläge dazu, welche Methoden einzusetzen sind, damit eine Sicherheitsanforderung (Sicherheitsintegritätslevel, siehe Abschnitt 6.5) erreicht werden kann. Insgesamt beziehen sich die Methoden für eine sichere Softwareentwicklung stark auf die Projektmanagementprozesse – angefangen beim Prozess zu Entwicklung, Integration und Test über die Anforderungen an die Dokumentationen bis zur Softwareerstellung selbst. So üben die erfahrenen Softwareentwickler einen großen Einfluss bei *Codereviews* aus.

5.1 Fallbeispiel: Flight 965

Es soll nun am Beispiel der Katastrophe des Flugs 965, siehe Artikel [21], gezeigt werden, welche Auswirkungen Softwarefehler nach sich ziehen. Das Szenario wird grafisch in Abbildung 5.1 dargestellt. Im Dezember 1995 startete die Boeing 757 zum Flug von Miami, USA, nach Cali, Kolumbien. Im Flugzeug befanden sich über 150 Passagiere. In Kolumbien gab es damals viele Aktivitäten durch terroristische Gruppen. Deswegen war unter anderem die Radarstation des Flughafens von Cali beschädigt, und die Bodenstation konnte die Positionen der Flugzeuge nicht beobachten. So musste sie sich auf die Positionsweitergabe durch die Piloten verlassen. Den Piloten helfen dabei sogenannte *Beacons*, die Position des Flugzeugs zu ermitteln. Diese senden über Funk die Information ab, wodurch der Abstand des Flugzeugs zum *Beacon* ermittelt werden kann. Eine Höheninformation kann mit dieser Methode aber nicht ermittelt werden. Der Flug war verspätet, wodurch die Entscheidung der Bodenstation zustande gekommen war, das Flugzeug von Süden des Flughafens her landen zu lassen. Dadurch wäre eine Abfertigung schneller gewesen. Das machte aber eine Wende des Flugzeugs, das aus dem Norden kam, notwendig. So wurde in der Nähe des Flughafens von Cali der Landeanflug eingeleitet, und die Landeklappen wurden aus-

gefahren. Das Flugzeug verlor deswegen an Höhe. Damit der Autopilot die Kontrolle des Flugzeugs übernehmen kann, gibt der Pilot in das *Flight Management System* (*FMS*) einen Wegepunkt ein. Dieser Wegepunkt war für den Flughafen Cali durch das Kennwort *Rozo* codiert. Das kann mit dem Buchstaben *R* eingegeben werden, und das *FMS* schlägt automatisch Wegepunkte vor, die in der Nähe des Flugzeugs liegen. Das *FMS* kürzte *Rozo* hier aber nicht mit *R* ab, sondern schlug den falschen Wegepunkt *Romeo* vor, der durch den Piloten ausgewählt wurde.

Romeo ist ein Wegepunkt bei der Stadt Bogotá, die 200 km entfernt liegt. Der Autopilot steuert daraufhin das Flugzeug in Richtung Bogotá. Wegen der Landeklappen befand sich das Flugzeug weiterhin im Sinkflug in Richtung Berge. Die Piloten brauchten mindestens eine Minute, bis sie realisierten, dass der Kurs falsch war. Deswegen leiteten sie eine Korrektur in Richtung Cali ein. In der Zwischenzeit waren die Berge auf der Höhe des Flugzeugs in Richtung des Flughafens. Da es ein Nachtflug war, konnten die Piloten die Berge nicht sehen. Ein Warnsystem, das den Abstand des Flugzeugs zum Boden misst, löste dann aus. Die Piloten reagierten sofort darauf und zogen das Flugzeug in die Höhe. Die Landeklappen verhinderten jedoch den notwendigen Aufstieg des Flugzeugs. Das Flugzeug schlug daraufhin in einen Berg ein. Von über 150 Passagieren überlebten nur vier Passagiere. Es folgten Anklagen der amerikanischen Flugbehörden an den Hersteller des *FMS* wegen der fehlerhaften Programmierung des Systems.

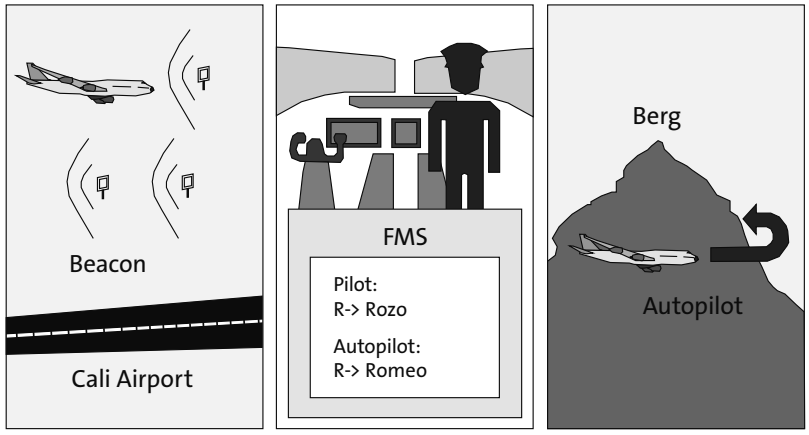


Abbildung 5.1 Flug 965

5.2 Softwareentwicklung

In dem Lebenszyklus von Softwareprojekten gibt es Phasen, die nacheinander und teilweise auch wiederholt durchlaufen werden. Abbildung 5.2 zeigt diese Phasen in einem V-Modell, das aus der Norm *IEC-61508* [6] entnommen wurde.

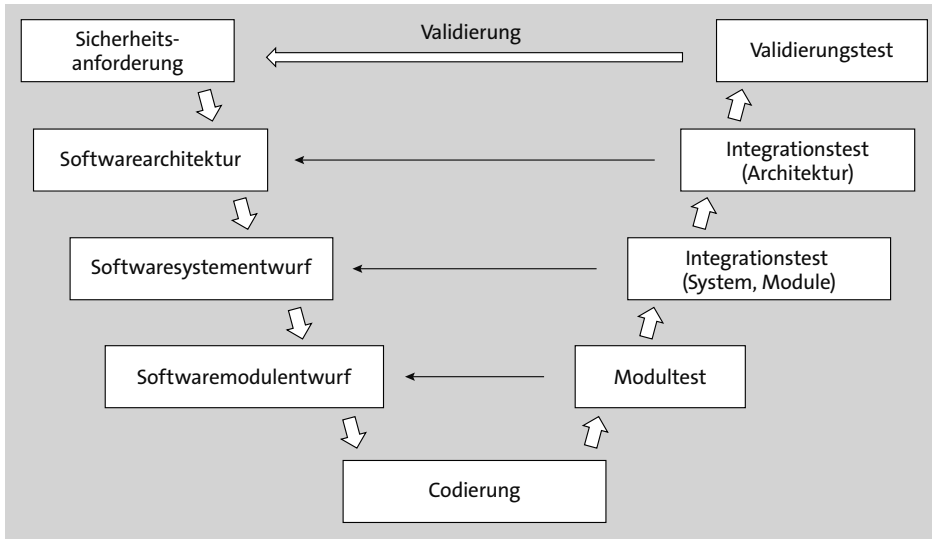


Abbildung 5.2 Phasen in der Softwareentwicklung

Dabei ist zu erwähnen, dass das V-Modell eine Erweiterung des Wasserfallmodells ist. In der Realität folgen die einzelnen Phasen nicht grundsätzlich nacheinander, sondern können durchaus für Teile der Projekte parallel laufen. In vielen Fällen muss eine Phase mehrmals durchlaufen werden, z. B. wenn nachgearbeitet werden muss. Nur grob kann eine agile Softwareentwicklung mit dieser Grafik beschrieben werden. Die Granularität der zu entwickelnden Softwareanteile muss dann entsprechend angepasst werden.

In der Regel erstellt der Safety Engineer ein Anforderungsdokument für die Sicherheit, um die erforderliche Funktion der Software (zusammen mit der Hardware) zu beschreiben. Bei sicherheitsgerichteten Systemen wird das Dokument in der Norm *IEC-61508* [6] *Software Safety Requirements* genannt. Nach der Beschreibung der Sicherheitsanforderungen erstellt der Safety Engineer ein Dokument zur Beschreibung des kompletten Systems. Dies beinhaltet in der Regel sowohl die Software als auch die Hardware. Genannt wird dieses Dokument bei sicherheitsgerichteten System *Software Architecture Design*-Dokument.

Eine weitere Detaillierung findet in der Phase des Softwaresystementwurfs statt. Das entstehende Dokument beschreibt hier die Softwareelemente, die die Sicherheitsanforderungen erfüllen sollen. Das Dokument heißt in der *IEC-61508* [6] *Software System Design*-Dokument. In klassischen Softwareprojekten kann dies einem *High-Level-Design*-Dokument entsprechen. Oftmals gibt es an dieser Stelle eine Übergabe vom Architekten an den Entwickler, der in einem *Software Module*-Dokument beschreibt, wie die Software zu entwickeln ist, um die Sicherheitsanforderungen zu erfüllen. In der Softwaretechnik wird dieses Dokument häufig auch *Low-Level-Design-*

Dokument genannt. Bei *Agile* wird dieser Detaillierungsgrad mit *User Stories* beschrieben. Im Fall von *Agile* werden *User Stories* Teil des sogenannten *Backlogs*.

Danach geht es in die Programmierphase über (Codierung im V-Modell), und Softwareentwickler programmieren die in den Dokumenten (*High-Level-Design*-Dokument, *User Stories* oder *Software Module*-Dokument) beschriebenen Anforderungen. In *Agile* wird in einem *Sprint Review Meeting* entschieden, welche *User Stories* in den *Sprint Backlog* kommen, die vom Entwicklungsteam selbstständig bearbeitet werden können.

Nach der Entwicklung der Softwaremodule werden diese durch das Entwicklungsteam selbst oder das Testteam getestet, abhängig z. B. von der Größe des Moduls oder der Komplexität. Ein Pfeil in der Abbildung zeigt, dass dabei gemeldete Fehler wieder zurück in die Entwicklung gehen. Dokumente und Software müssen stets anpasst werden. Bei *Agile* gibt es für *User Stories* Akzeptanzkriterien, die für das Softwarerelease erreicht werden müssen. Darüber hinaus gibt es aber noch Demonstrationen, um das Modul einem *Sprint Review*-Team zu zeigen. Gibt es einen Konsens, wird die Software in der Quellcode-Datenbank (*Repository*) gespeichert. Das Testteam integriert die Softwareteile, um diese komplett zu testen. Das Team folgt dabei dem *Software System Integration*-Dokument, in dem beschrieben wird, wann und wie die einzelnen Softwaremodule zusammengesetzt werden. Den Plan zum Test der kompletten Architektur liefert das *Software Architecture Integration*-Dokument.

Bei der agilen Softwareentwicklung bedienen sich die Entwickler einem Manifest, wobei einer der Punkte lautet: Funktionierende Software ist wichtiger als Dokumente. Dies steht mit der Nachweispflicht, die bei einer Beweisumkehrlast einzuhalten ist, im Konflikt. Hier muss das Entwicklungsteam einen guten Mittelweg finden. Prozessanpassungen sind aber durchaus bei *Agile* erlaubt. Anpassungen der Richtlinien in der Norm *IEC-61508* [6] sind bei guter Begründung ebenfalls möglich.

Der dritte Teil der Norm *IEC-61508* empfiehlt in den Anhängen eine Reihe von Maßnahmen zur Entwicklung von sicherer Software. Die aus dem Buch [1] und aus *IEC-61508* [6] abgeleitete Tabelle 5.1 zeigt einige Maßnahmen und gibt dabei Empfehlungen bezüglich der Sicherheitsintegritätslevels (*SIL*). Die *SIL* ergeben sich aus der Analyse der Risiken und werden nummeriert von 1 bis 4, wobei die Zahl aufsteigend eine Kennzeichnung für eine gesteigerte Sicherheitsanforderung ist. Die Bezeichnung + + gibt an, dass diese Maßnahme dringend empfohlen wird. + ist eine Angabe für die Empfehlung, und o ist keine Empfehlung. Eine Zuordnung von *SIL* und den geforderten Wahrscheinlichkeiten stellt die zentrale Tabelle 5.1 dar.

In der ersten Reihe von Tabelle 5.1 sehen Sie die Richtlinie zur Modularisierung und die strukturierte Programmierung von Quellcode. Diese wird bei allen *SIL* dringend empfohlen. Die Einhaltung von Entwurfs- und Codierungsrichtlinien wird zwar immer empfohlen, aber bei den höheren *SIL* dringend empfohlen. Rechnergestützte Entwurfswerkzeuge und ebenso defensive Programmierung werden vor allem bei

höheren *SIL* nahegelegt. In der Tabelle wird hier noch der Einsatz von semiformalen Methoden aufgeführt, die bei allen *SIL* empfohlen bzw. dringend empfohlen sind.

Bei Entwicklung und Test wird stets empfohlen (bzw. dringend empfohlen), dass ein Verweis zur Zurückverfolgung zwischen dem *Software Safety Requirements*-Dokument und den *Software Module*-Dokumenten stattfindet.

Richtlinie	SIL1	SIL2	SIL3	SIL4
Modularisierung und strukturierte Programmierung	++	++	++	++
Entwurfs- und Codierungsrichtlinien	+	++	++	++
Rechnergestützte Entwurfswerkzeuge	+	+	++	++
Defensive Programmierung	o	+	++	++
Semiformale Methoden	+	++	++	++
Verweise im *Software Safety Requirements*-Dokument auf die *Software Module*-Dokumente	+	+	++	++

Tabelle 5.1 Empfohlene Softwarerichtlinien bei Softwareentwurf und -entwicklung

5.2.1 Modularisierung und strukturierte Programmierung

Wichtig beim Entwurf von sicherer Software ist die Begrenzung der Komplexität, denn Komplexität ist immer ein Faktor für eine erhöhte Anzahl von Softwarefehlern. Eine Methode zur Begrenzung ist die Modularisierung, also die Aufteilung der Software in Komponenten, die überschaubar sind. Abbildung 5.3 zeigt ein Softwaremodul, das Schnittstellen für die Eingabe und die Ausgabe hat.

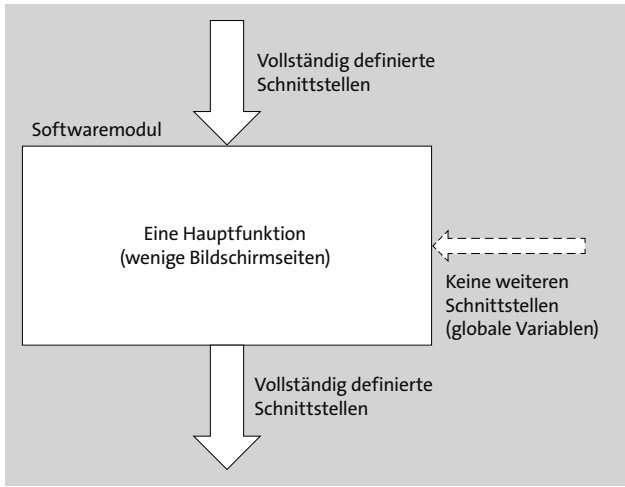


Abbildung 5.3 Modularisierung

In einem *High-Level-Design*-Dokument sollten die Schnittstellen der Softwaremodule vollständig definiert sein. Softwaremodule sollten tendenziell klein gehalten werden; z. B. sind wenige Bildschirmseiten ein guter Richtwert. Es gibt Softwarewerkzeuge, die die Modulgröße beschränken, um den Entwickler darauf aufmerksam zu machen. Grundsätzlich sollten globale Variablen vermieden werden, da diese Abhängigkeiten zu anderen Softwaremodulen erhöhen und somit die Komplexität steigern.

Bei der strukturierten Programmierung geht es um die einfache Gestaltung des Steuerflusses des Softwareprogramms. Abbildung 5.4 zeigt dazu eine schematische Darstellung. Idealerweise sollte hier ein Modul das nächste aufrufen. Zirkulare Abhängigkeiten (z. B. Modul A ruft Funktionen von Modul B auf, das wiederum ruft Funktionen von Modul A auf) sollten vermieden werden. Diese Art von Strukturierung verhindert somit einen komplizierten Programmablauf. Module dürfen auch, wie in der Abbildung gezeigt, seitwärts weitere Module aufrufen. Diese springen beim Ablauf aber wieder zurück in das Ausgangsmodul. Voraussetzung dafür ist die einfache Beziehung zwischen dem aufrufenden und dem aufgerufenen Modul. Hier muss es eine einfache Beziehung mit den Schnittstellen der Eingabe und Ausgabe geben.

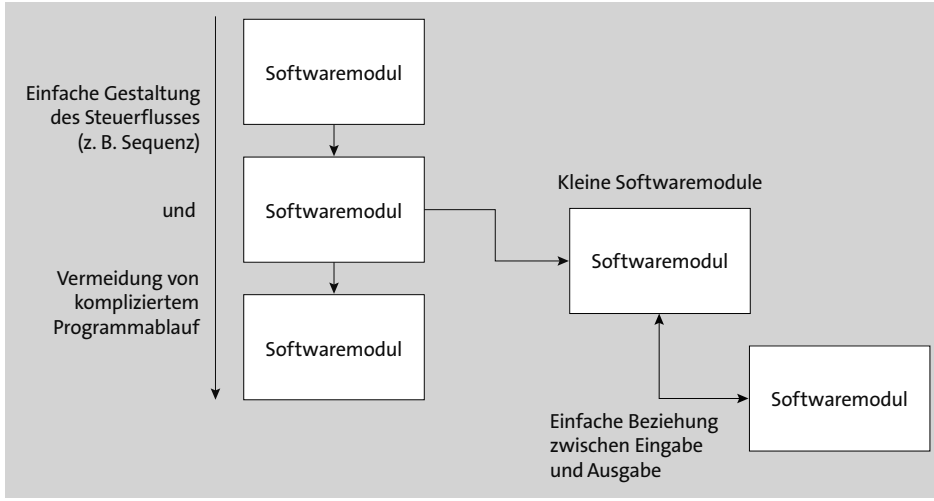


Abbildung 5.4 Strukturierte Programmierung

5.2.2 Entwurfs- und Codierungsrichtlinien

Entwurfs- und Codierungsrichtlinien werden in Dokumenten festgehalten und dienen dem Softwareentwicklungsteam als Richtlinie für einen einheitlichen Programmstil, um Softwarequalität, Lesbarkeit und Wartbarkeit zu verbessern. Diese Richtlinien werden über Abteilungsgrenzen hinweg von den Softwareentwicklern selbst entworfen.

Den Anfang macht z. B. die Verwendung einer einheitlichen Sprache wie Englisch. Vor allem Entwickler in deutschen Firmen neigen bei der Benennung von Bezeich-

nern dazu, eine Mischung aus Englisch und Deutsch zu verwenden. Sobald Mitarbeiter ohne Deutschkenntnisse an der Entwicklung beteiligt sind, wird die Lesbarkeit für sie schwierig.

Des Weiteren kann als Beispiel eine Regelung für den Einsatz von *Pointern* genannt werden, insbesondere wenn eine Sprache wie C++ eingesetzt wird. Pointer setzen die Lesbarkeit des Programms herunter und erhöhen gleichzeitig die Fehleranfälligkeit. Es ist daher gerechtfertigt, dass bei höheren *SIL*-Anforderungen die Pointer-Programmierung nur eingeschränkt eingesetzt werden soll. Eine ganz typische Regel bei der Pointer-Programmierung in C++ ist die *Null-Pointer-Abfrage*. Bei der Instanziierung bekommt dabei der Pointer stets einen NULL-Wert zugeordnet. Kurz vor der Dereferenzierung des Pointers sollte dann immer eine Abfrage auf den Wert erfolgen, sodass ein *Segmentation-Fault* (ein Zugriff in einem nicht instanziierten Bereich des Speichers) vermieden wird.

Eine weitere Regel kann z. B. der eingeschränkte Einsatz von *Interrupts* sein. Interrupt-Programmierung macht den Programmfluss kompliziert, da oftmals Variablen zwischengespeichert werden müssen.

Außerdem muss beachtet werden, dass sequenzielle und verschachtelte Aufrufe von Interrupt-Funktionen (also die Handhabung eines Interrupts innerhalb der *Interrupt*-Funktionen) wirklich vom Betriebssystem unterstützt werden können. Fehler in Interrupt-Funktionen sind schwer nachzuvollziehen, Fehler im Betriebssystemcode bei der Interrupt-Handhabung sind sehr schwer zu beheben.

Es gibt gute Gründe, Sprachen wie C++einzusetzen. Beispielsweise ist bei richtiger Programmierung die Performanz vergleichbar mit der Performanz von Assemblersprachen. Der Preis dafür ist aber die sehr hardwarenahe Programmierung. Als Beispiel kann man die Allokation von dynamischen Variablen sehen. Diese müssen nach dem Gebrauch wieder deallokiert werden, da ansonsten die Allokation im Speicher bestehen bleibt. In der Fachsprache wird dieses Problem *Memory Leak* genannt. Schlimmstenfalls kann der Speicher mit allokierten Daten überlaufen, und dies führt zu einem Programmabsturz. Daher sollten automatische Variablen eingesetzt werden oder Datencontainer, die als Bibliotheken in C++ zu Verfügung stehen. Zu nennen ist die *Standard Template Library* (*STL*) oder *Boost*. Alternativ dazu kann eine programmiertechnische Überwachung der dynamischen Daten z. B. mithilfe von Smart-Pointern eingesetzt werden. Hier erfolgt die Deallokation automatisch.

5.2.3 Rechnergestützte Entwurfswerkzeuge

Der dritte Teil der Norm *IEC-61508* empfiehlt nach Tabelle 5.1 den Einsatz von rechnergestützten Entwurfswerkzeugen. Dabei gibt es Entwicklungswerkzeuge, die direkt vom Softwareentwickler eingesetzt werden, und Entwicklungswerkzeuge, die in einem späteren Entwicklungsprozess ihre Verwendung finden, z. B. beim Bau der

Software. Es ist daher wichtig, den gesamten Softwareentwicklungsprozess zu betrachten und zu verinnerlichen. Es empfiehlt sich, den Entwicklungsprozess komplett zu beschreiben und dann die entsprechenden Prozesse zu implementieren. Dies ist keine einfache Aufgabe und kann abhängig von Team- und Abteilungsgröße mehrere Monate in Anspruch nehmen.

Der Prozess ist auch kontinuierlich, da er stets Änderungen bzw. Verbesserungen durchläuft. Erfolgen Prozessschritte der Softwareentwicklung automatisch, wird von *Continuous Integration* (*CI*) gesprochen. *CI* endet oftmals bei der Speicherung des Quellcodes in einer Quellcode-Datenbank, genannt *Repository*. Moderne Ansätze, wie z. B. *Continuous Deployment* (*CD*) oder *DevOps* (Kunstwort aus *Software Development* und *IT Operations*), gehen dabei noch einige Schritte weiter. *CD* beinhaltet die weitgehende Installation der Software aus dem Repository beim Kunden mithilfe von Orchestrierungstools, wie z. B. *Kubernetes*. *DevOps* bezieht auch das operative Team und den Kunden mit ein, um schnell Feedback zu erhalten, das in den Entwicklungsprozess zurückfließen kann. Eine Abhandlung über die *DevOps*-Methoden wird in Artikel [22] beschrieben.

Im Folgenden werden Beispiele für Entwurfswerkzeuge aufgezeigt, die beim automatischen Entwicklungsprozess von Software eine Rolle spielen können. Es werden bei Weitem nicht alle Werkzeuge aufgezeigt. Dies soll nur eine Beschreibung mit Beispielen dafür sein, welche Art von Werkzeugen eingesetzt werden können.

5.2.4 Statischer Quellcode-Analysator

Das Werkzeug *Sonarqube* ist ein Beispiel, das direkt vom Softwareentwickler bei der Entwicklung eingesetzt werden kann. Damit kann z. B. Java-Code oder Code anderer Sprachen nach *Sicherheitsaspekten*, *Bugs* oder *Good Practices* analysiert werden. Das Werkzeug weist auf bedenkliche Passagen hin und schlägt vor, wie der Entwickler diese korrigieren kann. Es gibt hier ebenfalls Möglichkeiten, das Werkzeug in den Entwicklungsprozess des Softwareentwicklers einzubinden, damit früh Probleme in der Software erkannt werden. So kann das Speichern des Quellcodes in ein Repository kontrolliert erfolgen.

Das Werkzeug *lint* ist ein statischer Quellcode-Analysator, ursprünglich entwickelt für C/C++ Quellcode. Abwandlungen davon gibt es auch für weitere Programmiersprachen. Das Werkzeug *lint* untersucht vor allem den Kontext des Quellcodes, aber nicht seinen Stil. Beispielsweise kann *lint* so konfiguriert werden, dass Zuweisungen von Initialwerten bei neu instanziierten Variablen erfolgen müssen, ansonsten werden Fehler ausgegeben. Ein konkretes Beispiel ist hier die Zuweisung von NULL-Werten in Pointern. Weitere Beispiele sind die Überprüfung von Fallunterscheidungen, fehlende Lizenz-Headers etc.

5.2.5 Dynamischer Quellcode-Analysator

Das Werkzeug *valgrind* ist für dynamische Tests von kompiliertem Code ausgelegt. Der kompilierte Code muss nicht besonders aufbereitet werden, da *valgrind* den binären Code in einer virtuellen Maschine ausführt. Insbesondere C++ ist anfällig für *Memory Leaks*. Das Werkzeug *valgrind* überprüft hier, ob nach Beendigung des Programms (oder von Funktionen bzw. Methoden) allokierte Daten im Speicher noch vorhanden sind. Weiter können Übertritte beim Schreiben und Lesen von Datengrenzen detektiert werden, die bei C++ wegen der Pointer-Programmierung einfach durchzuführen sind. *valgrind* lässt sich in die Entwicklungsumgebung einbauen, sodass nach dem Bau der kompletten Software (oder Teilen davon) eine Überprüfung stattfinden kann. Bei einem Fehler kann die Speicherung des Quellcodes in den Hauptast des Repository verhindert werden.

5.2.6 Quellcode-Speicher bzw. Repository

Unabhängig vom Entwicklungsprozess ist ein *Repository* eine Notwendigkeit bei der Softwareentwicklung. Es ist eine Datenbank mit Benutzerschnittstellen, um Quellcode zu speichern. Je nach Art lässt sich der Inhalt des Repository leicht duplizieren, um Backups zu erzeugen. Diese sind notwendig bei einem Ausfall des Servers oder bei korrumpierten Repository-Daten. So kann ein alter Softwarestand wiederhergestellt werden. Auch können Softwareentwickler selbst auf alte Softwarestände gehen für den Fall, dass fehlerhafter Quellcode gespeichert wurde.

Eine weitere wichtige Eigenschaft eines Repository ist die Möglichkeit, Konflikte zwischen Quellcodes mehrerer Entwickler zu lösen. Auch bei dieser sehr knappen Beschreibung des Repository sollte ersichtlich sein, dass seine Handhabung vom Entwickler geübt sein muss. Eine hohe Bereitschaft zur Kommunikation mit anderen Teammitgliedern (z. B. bei der Quellcode-Konfliktlösung) ist erneut hier Voraussetzung.

5.2.7 Quellcode-Beautifier

Artistic Style (kurz *ASTYLE*) ist ein Werkzeug zur Überprüfung von Codierungsrichtlinien, siehe auch Abschnitt 5.2.2. Es gibt hier die Möglichkeit, das Werkzeug so zu konfigurieren, dass *Best Practices* bei der Programmierung durch eine automatische Überprüfung eingehalten werden.

Beispielsweise kann das Entwicklungsteam fordern, bestimmte Einrückungsregeln im Quellcode einzuhalten. Das Werkzeug überprüft den Quellcode nach diesen Einrückungsregeln und gibt eine Fehlermeldung bei Nichteinhaltung mit dem Hinweis auf die Quellcode-Zeile aus. *ASTYLE* lässt sich in den Entwicklungsprozess einbauen,

sodass vor dem Einchecken in das Repository die Überprüfung erfolgt. Schlägt sie fehl, wird das Speichern des Quellcodes verhindert.

5.2.8 Quellcode-Reviewing

Ganz wesentlich bei der Verbesserung der Quellcode-Qualität ist der Einsatz von *Reviews*, z. B. nach dem Vieraugenprinzip. Es darf nämlich die Entwicklungsregel aufgestellt werden, dass nur Quellcode, der durch einen *Review* mit Teammitgliedern begutachtet worden ist, in das Repository gespeichert werden darf. Dieser *Review*-Prozess lässt sich oft ohne Werkzeuge leicht umgehen. Ein Werkzeug wie etwa *Codestriker* oder *Gerrit* kann hier Abhilfe schaffen.

Auch hier muss erwähnt werden, dass Quellcode-*Review*-Werkzeuge in vielen Produkten bereits vorhanden ist (z. B. in *GitLab* oder *GitHub*). So lassen sich Werkzeuge wie *Codestriker* oder *Gerrit* in den Entwicklungsprozess derart einbauen, dass Speichern von Quellcode nur dann erlaubt ist, wenn tatsächlich ein positives *Review*-Ergebnis durch die Teammitglieder erfolgt ist.

5.2.9 Defensive Programmierung

Bei den Codierungsrichtlinien sollte die Handhabung von Funktions- und Methodenaufrufen und die Handhabung der Rückgabewerte besonders betrachtet werden. Hier wird die Typüberprüfung und die Überprüfung von Eingabe- und Rückgabewerten behandelt. Abbildung 5.5 zeigt das Softwaremodul mit den Eingängen und Ausgängen.

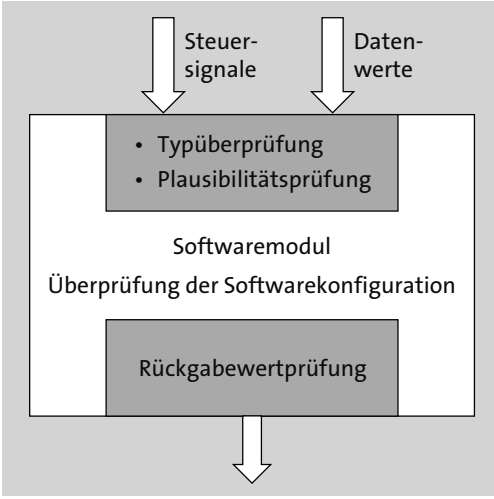


Abbildung 5.5 Defensive Programmierung

Moderne Sprachen wie z. B. Python besitzen die Möglichkeit, typunabhängig den Funktionen und Methoden Parameterwerte zu übergeben. Das bedeutet, dass der Typ des Werts beim Funktionsaufruf nicht geprüft wird. Das geht in den meisten Fällen gut. So liefern Softwaretests dem Entwickler dabei positive Bestätigungen.

Nun sind Softwaretests nicht perfekt, und es gibt immer die Möglichkeit, dass ein bestimmter Test ausgelassen wurde. Daher ist es besser, von vornherein sicherzustellen, dass ein passender Typ eines Parameters übergeben wird. Die Sprache C++ gilt hier als typsicher, solange nicht mit Pointern gearbeitet wird. Also besteht hier das Problem nur bedingt. Die Typsicherheit lässt sich durch Typumwandlung bei der Pointer-Programmierung umgehen. Bei der Sprache Python kann durch eine spezielle Programmierweise Typsicherheit hergestellt werden, entweder durch Angabe des Typs in der Funktionsdeklaration oder durch Abfragen des Typs am Anfang der aufgerufenen Funktion. Dies hat natürlich den Vorteil, dass eine entsprechende Behandlung (z. B. mit *Exceptions*) erfolgen kann. Die defensive Programmierung schreibt die Überprüfung der Parametertypen und der Parameterwerte vor. Die Sprache C++ ist eigentlich typsicher, es sei denn, es werden Parameter wie *Void-Pointer* übergeben. Hier sollte eine Überprüfung des Typs, aber auch des Inhalts in dem Code erfolgen, der direkt am Anfang der Funktion oder der Methode steht. Eine Fehlerbehandlung bei einem fehlerhaften Typ sollte sofort geschehen. Wie die Fehlerbehandlung abläuft (z. B. über Exceptions oder über die Rückgabe eines Fehlercodes), ist der Programmierkultur des Entwicklungsteams überlassen.

Ein weiterer Aspekt bei der defensiven Programmierung ist die Überprüfung des Inhalts bzw. der Werte der Übergabeparameter. Übergabeparameter haben normalerweise einen Wertebereich. Eine Überschreitung des Wertebereichs kann sofort als Fehler identifiziert werden. Also sollte der Code mit dem falschen Wert nicht weiter ausgeführt werden, stattdessen sollte die Funktion sich sofort beenden und eine entsprechende Fehlermeldung zurückliefern. Nahezu alle Sprachen besitzen das Konzept der Überprüfung mit der `assert`-Funktion. Die `assert`-Funktion erhält als Übergabeparameter eine Vergleichsoperation. Hier hat der Softwareentwickler die Möglichkeit, Übergabeparameter der Funktion direkt auf einen Wertebereich zu überprüfen. Beispielsweise ist ein Übergabeparameter α eine Winkelangabe und hat einen Wertebereich zwischen 0 und 180°. Tatsächlich ist es nicht falsch, 190° anzugeben, da dies auch einen regulären Winkel darstellt. Aber der folgende Quellcode der Funktion muss nicht notwendigerweise diesen Winkelwert handhaben können. Die `assert`-Funktion kann leicht diese Überprüfung ausführen. Falls die Vergleichsoperation positiv ist, hat die `assert`-Funktion keine Auswirkung, und der darunterliegende Code wird ausgeführt. Ist sie aber negativ, bricht die Ausführung des Programms ab mit Angabe der Zeile im Quellcode. Dies ist zwar ein erwünschtes Verhalten von ausgeliefertem Code auf dem Feld. Auf der Testfläche aber ist es durchaus willkommen, da der Tester dem Softwareentwickler schnell das Problem mit Zeilenangabe erklären kann. Dann ist es

oft ein für den Softwareentwickler schnell zu behebendes Problem, wenn er die Ursache eines Übertritts des Wertebereichs ermittelt. Die `assert`-Funktion kann sich leicht durch *Compiler*-Optionen ausschalten. So kommt es niemals vor, dass die `assert`-Funktion ein Programm im negativen Fall abbricht. Daher kann es von Vorteil sein, auch hier eine dedizierte Abfrage des Wertebereichs vorzunehmen, ähnlich wie bei der Typabfrage. Wird also der Wertebereich überschritten, wird eine Exception geworfen, oder ein Fehlercode kann zurückgeliefert werden.

Der letzte Aspekt ist die Überprüfung des Inhalts des Rückgabewerts. Ob nun Rückgabewerte über die `return`-Funktion oder über Parameter (Pointer, Referenzen) zurückgeliefert werden, ist der Programmierkultur der jeweiligen Firma oder der Abteilung zuzuordnen. Dennoch kann beides nach deren Inhalt überprüft werden. Ähnlich wie bei der Parameterüberprüfung kann auch hier die `assert`-Funktion eingesetzt werden, jedoch mit dem Nachteil, dass in einer Produktivumgebung (im Feld, also beim Kunden) keine robuste Fehlerbehandlung stattfindet. Auf der Testfläche dagegen hat die `assert`-Funktion durchaus seine Berechtigung und hat Vorteile.

Dennoch kann eine wirkliche Fehlerbehandlung der `assert`-Programmierung vorgezogen werden, und zwar indem der Wertebereich des Rückgabewerts überprüft wird. Bei Überschreitung sollte eine Exception bzw. die Rückgabe eines Fehlercodes erfolgen.

5.2.10 Semiformale Methoden

Eine Softwarerichtlinie ist die Beschreibung der Softwarefunktionen mithilfe von semiformalen Methoden, die im Anhang B der Norm *IEC-61508* [6] (siehe Tabelle 5.1) aufgeführt sind. Beispiele für semiformale Methoden sind Sequenzdiagramme, Datenflussdiagramme, Petri-Netze etc. Die *Unified Modeling Language* (*UML*) unterstützt dabei sämtliche Möglichkeiten zur Beschreibung von Software. Ursprüngliches Ziel bei *UML* war die Beschreibung der zu entwickelnden Software mithilfe von grafischen Mitteln oder durch die Sprache *UML* selbst. Auch diese konnte prozessiert werden, und es entsteht daraus C++, Java und anderer Quellcode. Allerdings hat sich die grafische Programmierung kaum durchgesetzt, da deren Vorteile für den Softwareentwickler nicht offensichtlich sind. Dennoch hat *UML* Vorteile, insbesondere bei sicherheitsrelevanter Software. Der Architekt kann die zu entwickelnde Software grafisch und einfach darstellen, sodass das Team darüber diskutieren kann, ohne eine Quellcode-Zeile oder eine schriftliche Beschreibung der Anforderung anschauen zu müssen. Der Aufbau der Software kann über Klassendiagramme dargestellt werden, die Funktionsweise der Klassen über Sequenzdiagramme. Die Grafiken lassen sich leicht in Dokumente einbinden, wie z. B. in ein *High-Level-Design*-Dokument für ein sicherheitsrelevantes System. Dieses Dokument dient natürlich auch als Sicherheitsnachweis.

Abbildung 5.6 zeigt ein einfaches Klassendiagramm und ein Sequenzdiagramm. Klassendiagramme beschreiben lediglich die Klassen und deren Verbindung zueinander. Sie zeigen die Vererbungshierarchie, aber auch die Bindungen der Klassen mit unterschiedlicher Stärke, genannt *Assoziation*, *Komposition* und *Aggregation*. Die Sequenzdiagramme zeigen den dynamischen Verlauf eines Programms. Klassen werden also instanziiert und rufen ihre Methoden auf, die durch Pfeile dargestellt werden. Die Instanzen sind Rechtecke, Lebenslinien führen von oben nach unten und stellen den zeitlichen Verlauf dar.

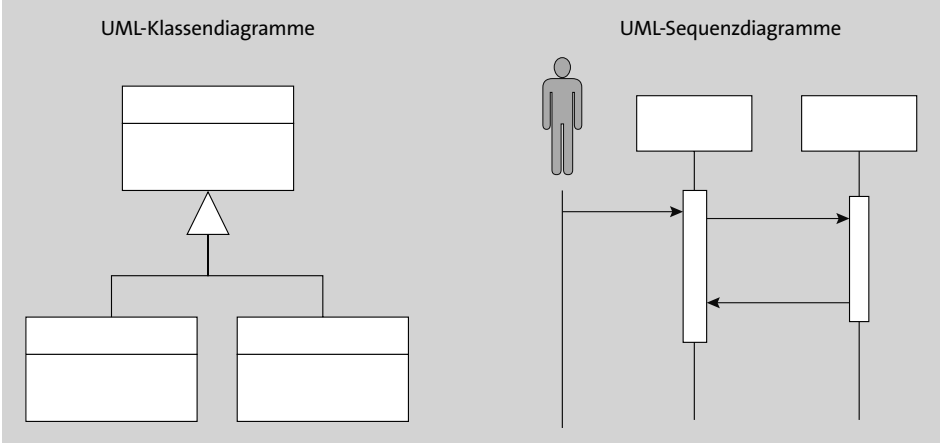


Abbildung 5.6 Semiformale Methoden zur Entwicklung

5.2.11 Verweise im Dokument Software Safety Requirements

Die letzte hier aufgeführte Softwarerichtlinie ist nicht programmiertechnischer Natur, sondern hat mit der Dokumentationspflege bei der Entwicklung zu tun. Softwareentwicklungsdokumente bilden für den Entwickler die Grundlagen zur Programmierung. Sie entstehen aus dem Dokument *Software Safety Requirements*. Eine Referenzierung ist demnach bei den höheren *SIL* notwendig. Dies hat den Vorteil, dass bei der Entwicklung des Quellcodes stets das *Software Safety Requirements*-Dokument und die übrigen Entwicklungsdokumente angepasst werden müssen. So kann eine Überprüfung dahin gehend stattfinden, ob die Anforderungen technisch erreicht werden.

5.3 Modul- und Integrationstests

Nach dem *V-Modell* ist die erste Phase die Entwicklung der Sicherheitsanforderungen oder des Lastenhefts, und danach folgt die Entwicklung der Architektur, die sowohl die Hardware als auch die Software beinhaltet. Daran schließen sich weitere Phasen

an, z. B. der System- und der Modulentwurf, die z. B. im *High-Level-Design* und im *Low-Level-Design* dokumentiert sind. Hat die Entwicklung stattgefunden, wird überprüft, ob die Anforderung durch das Softwaresystem erfüllt wurde. Es wird nun zwischen Verifikation und Validierung unterschieden.

5.3.1 Verifikation

Nach *DIN-9000* (aber auch nach *IEC-61508*) ist die Verifikation die Bestätigung durch Bereitstellung eines objektiven Nachweises, dass festgelegte Anforderungen erfüllt sind. Ein objektiver Nachweis bedeutet, dass es eine konkrete Beschreibung einer Anforderung in einem der Softwareentwicklungsdokumente (Architektur, System, Modul) gibt. Das Testteam muss zeigen, dass diese erfüllt sind. Eine Anforderung kann z. B. sein, dass die Auslastung einer CPU nur maximal 60 % erreicht. Das Testteam wird dann mithilfe von Testwerkzeugen beim Betrieb unter Volllast belegen müssen, dass dieser Maximalwert niemals erreicht wurde.

5.3.2 Validierung

Die Definition der Validierung unterscheidet sich dabei leicht. Die Validierung ist nämlich die Bestätigung durch Bereitstellung eines objektiven Nachweises, dass die Anforderungen für einen spezifischen beabsichtigten Gebrauch oder eine spezifische Anwendung erfüllt worden sind, siehe *IEC-61508*. Es können nämlich im *Software Safety Requirements*-Dokument Anforderungen in Form von Gebrauchsmustern beschrieben werden. Als Beispiel kann hier genannt werden, dass ein Fahrzeug, das auf trockener Straße bei 60 km/h einen Bordstein streift, nicht außer Kontrolle gerät. Das Gebrauchsmuster ist also, das Fahrzeug bei den angegebenen Parametern an den Bordstein zu steuern. Ein Testteam wird diese Fahrsituation auf der Straße rekonstruieren und so das beschriebene Gebrauchsmuster validieren.

5.3.3 Modul-Logging

Eines der wichtigsten Werkzeuge des Softwareentwicklers bleibt der *Debugger*, mit dem Softwarecode Schritt für Schritt durchlaufen werden kann, um den Verlauf des entwickelten Codes zu verstehen. Es ist empfehlenswert, jede Zeile des Quellcodes zu *debuggen*, bevor dieser in das Repository gespeichert wird. Dadurch wird dem Entwickler die Funktionsweise noch einmal richtig bewusst. Falls das Programm aber schon operativ auf der Testfläche und im Feld ist, wird der Debugger dem Entwickler nur eingeschränkt etwas nützen, wenn er nicht die Ausgangssituation und die Fehlersituation komplett versteht, um diese an seinem Schreibtisch zu rekonstruieren. Problembeschreibungen der Tester liegen in Form von Beschreibungen in *Tickets* aus einem Ticketmanagementsystem vor, die für den Tester oder den Kunden eine Mo-

mentaufnahme der Fehlersituation darstellen. Der Entwickler weiß nach Zuweisung des Tickets, dass ein Problem vorherrscht. Meist wird ihm diese Information allein nicht helfen, das Problem zu beheben. Für diese Fälle helfen *Logging-Dateien*. Logging-Dateien enthalten Einträge, die bei der Ausführung des Programms geschrieben wurden, und zwar in der Regel bei Eintritt in eine Funktion oder Methode und bei Austritt der Funktion oder Methode. Bei Eintritt werden oftmals der Funktionsname und die Werte der Parameter in die Logging-Datei eingefügt. Bei Austritt einer Funktion werden z. B. der Funktionsname und der Rückgabewert hineingeschrieben, siehe auch Abbildung 5.7. Optional können weitere Informationen hinzugefügt werden, z. B. die Uhrzeit oder die Dauer der Ausführung der Funktion. Sehr oft helfen die Logging-Dateien dem Entwickler, Probleme zu lösen, da der Verlauf nachvollziehbar wird. Nun hat Logging den Nachteil, dass die Dateien nach einiger Zeit sehr groß werden können und dadurch die Speicherkapazität der Festplatte an ihre Grenzen kommt.

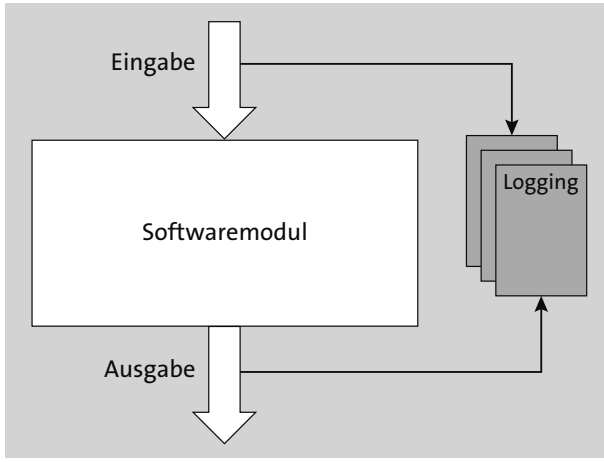


Abbildung 5.7 Modultest mit Logging

Aus diesem Grund haben bessere *Logging-Bibliotheken* die Möglichkeit, Dateien zu komprimieren oder alte Einträge mit dem *Round-Robin-Verfahren* zu überschreiben. Beim Round-Robin-Verfahren wird nur eine konfigurierte Anzahl von Zeilen in die Logging-Datei geschrieben. Beim Erreichen dieser Anzahl werden die alten Einträge überschrieben.

Eine weitere Möglichkeit ergibt sich durch das Setzen von Prioritäten. So kann die Logging-Bibliothek so konfiguriert werden, dass z. B. auf der Testfläche alle Funktionen ihre Informationen in die Logging-Dateien speichern, während beim Kunden im Feld nur Informationen mit hoher Priorität gespeichert werden. Es sei noch angemerkt, dass nicht nur auf der Testfläche oder beim Kunden das Logging Sinn ergibt, sondern auch beim Arbeiten innerhalb des Teams, hier kann es der Kommunikation

oder Problemfindung dienen. Auch lassen sich beim Bauprozess Softwareanalysatoren einsetzen, um für automatisierte Tests Kontrollfunktionen einzubauen. So können Rückgabewerte, die Dauer der einzelnen Funktionen und die Testabdeckung aus den Logging-Dateien automatisch ausgelesen werden.

5.3.4 Testabdeckung

Testabdeckungswerkzeuge sind weitere wichtige Werkzeuge. Testabdeckung ist die Analyse der Durchläufe der Funktionen und Methoden bei der Ausführung des Programms. Bei vielen Entwicklern herrscht die begründete Einstellung, dass Quellcode, der nicht beim Test durchlaufen worden ist, nicht funktionsfähig ist. Nach dieser Einstellung ist an den Kunden gelieferter Quellcode, bei dem nicht alle Funktionen getestet worden sind, ungetesteter Quellcode. Dies soll unbedingt vermieden werden. Das gilt besonders bei sicherheitsgerichteter Software.

Zur Erklärung der Testabdeckung eignen sich Kontrollflussgraphen. Abbildung 5.8 stellt eine einfache `if`-Anweisung als Kontrollflussgraph dar. Den Quellcode der entsprechenden `if`-Anweisung zeigt Listing 5.1. Eine Variable `a` wird auf ihre Größe geprüft. Ist diese kleiner als 3, wird der Variablen `result` der Wert 20 zugewiesen. Danach wird die Variable `result` inkrementiert.

```
if a<3:
    result = 20
result += 1
```

Listing 5.1 if-Anweisung in Python

Abbildung 5.8 zeigt den entsprechenden Kontrollflussgraphen. Der oberste Knoten entspricht der `if`-Anweisung. Es gibt hier eine Entscheidung, die über die zwei Transitionen ausgedrückt wird. Falls also `a` kleiner ist als 3, gibt es einen Übergang in den Knoten `result = 20`, ansonsten in den Knoten `result+= 1`.

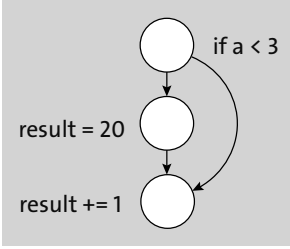


Abbildung 5.8 Kontrollflussgraph der if-Anweisung

Wie oben erwähnt, gibt es also ein allgemeines Interesse bei Projektleitung, Entwicklern und Testern, jede einzelne Zeile des Quellcodes vor der Auslieferung auszufüh-

ren. *Debugger* können diese Aufgabe erfüllen, allerdings hat der Tester zu wenig Kenntnis über den Quellcode, um jede Codezeile zu testen. Es ist auch ein erheblicher manueller Aufwand. *Coverage*-Werkzeuge helfen dem Tester dabei, Kennzahlen zu erhalten. Ein Beispiel ist eine Prozentzahl, die angibt, wie viel Prozent der Codeanteile tatsächlich beim Test durchlaufen wurde. Auch können genau die Codestellen zurückgeliefert werden, deren Ausführung nicht getestet wurde. Das Verhältnis des Anteils von ausgeführtem Quellcode zum gesamten Quellcode wird *Statement Coverage* genannt, kurz *C0*. Nun ist an dem einfachen Beispiel in Abbildung 5.8 schon ersichtlich, dass die Variable `a` immer kleiner sein muss als 3, damit die *C0*-Kennzahl 100 % ist, da alle Codestellen durchlaufen worden sind. Was ist nun, wenn `a = 3` ist? Der Programmablauf ist ein anderer mit einem anderen Ergebnis. Dieser Test wäre bei der *Statement Coverage* nicht notwendig, da *C0* bereits mit `a` kleiner 3 erfüllt ist.

Aus diesem Grund erfolgt die Erweiterung, und zwar die Zweigabdeckung, auf Englisch *Decision Coverage*, kurz *C1*. Bei der *C1*-Bedingung besteht nun das Ziel darin, alle Verzweigungen zu testen. Listing 5.2 zeigt einen einfachen Quellcode zur Veranschaulichung. Die Funktion `calc` liefert den Rückgabewert von `result` zurück, der abhängig von den beiden Eingangsparametern `x` und `y` gesetzt wird.

```
def calc(x,y):
    result = 0
    if x > 0 and y > 0:
        result = 30
    if x > 10 and y < 10:
        result += 30
    else:
        result += 10
    return result
```

Listing 5.2 Die Funktion calc in Python

Der Kontrollflussgraph aus Listing 5.2 wird in Abbildung 5.9 dargestellt, links aus der Sicht der *C0*-Abdeckung und rechts aus der Sicht der *C1*-Abdeckung. Die Struktur ist bei beiden Graphen gleich, aber der Fokus ist unterschiedlich. Bei der *C0*-Abdeckung stehen die Knoten im Fokus, bei der *C1*-Abdeckung die Transitionen. Die Funktion `calc` startet mit der Initialisierung der Variablen `result`, die bei beiden Graphen durch den ersten Knoten dargestellt wird. Danach kommt die erste `if`-Anweisung. Abhängig von den Eingangsparametern `x` und `y` wird die Anweisung `result = 30` ausgeführt, welcher wiederum als Knoten dargestellt wird. Es folgen `if-else`-Anweisungen, die im Kontrollflussgraphen durch einen Knoten mit zwei Verzweigungen dargestellt werden. Abhängig von der `if`-Bedingung werden die Codestellen `result += 30` oder `result += 10` ausgeführt. Beide Codestellen sind als hervorgehobene Knoten im Kontrollflussgraphen dargestellt. Der letzte Knoten im Kontrollflussgraphen bildet die

`return`-Anweisung ab, bei dem der Wert in `result` dem Aufrufer der Funktion `calc` zurückgeliefert wird. An diesem Quellcode ist wiederum ersichtlich, warum eine Überprüfung der *C0*-Abdeckung in vielen Fällen nicht ausreichend sein kann. Zur Wiederholung: Alle Codestellen müssen ausgeführt werden, damit *C0* erfüllt ist. Die Ausführung der Anweisung `result = 30` ist aber nicht verpflichtend. Wie kann der Tester bei *C0* überhaupt wissen, dass die Initialisierung `result = 0` richtig ist, da sie stets mit `result = 30` überschrieben wird? Es sollten also alle Verzweigungen getestet werden und nicht ausschließlich die Knoten.

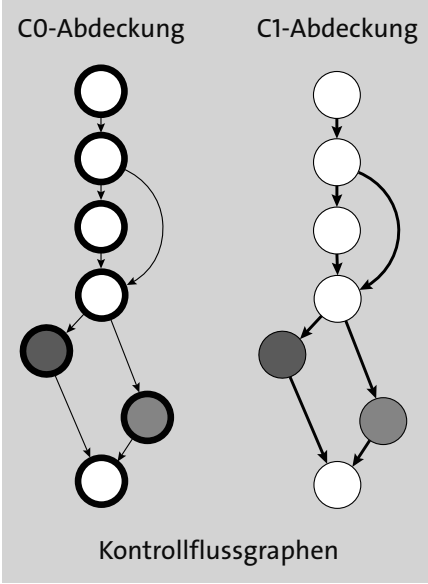


Abbildung 5.9 Kontrollflussgraphen der Funktion calc mit C0- und C1-Abdeckung

Die Problematik mit *C0* und deren Lösung mit *C1* wird durch das Beispiel oben ausreichend beschrieben. Allerdings kann der Verlass auf *C1* weiterhin problematisch sein. Denn die erste `if`-Anweisung besteht aus zwei Bedingungen (engl. *Conditions*): `x > 0` und `y > 0`. Man nennt jede einzelne Bedingung atomare Bedingung, da sie sich nicht durch weitere boolesche Operatoren (z. B. `and`, `or`, `not`) weiter zerlegen lässt. Die Bedingungen `x > 0` und `y > 0` der ersten `if`-Anweisung haben als Ergebnis vier Möglichkeiten:

1. $x > 0$ is `True` und $y > 0$ is `True`: insgesamt `True`
2. $x > 0$ is `True` und $y > 0$ is `False`: insgesamt `False`
3. $x > 0$ is `False` und $y > 0$ is `True`: insgesamt `False`
4. $x > 0$ is `False` und $y > 0$ is `False`: insgesamt `False`

Um *C1* komplett zu erfüllen, müssen nicht notwendigerweise alle Fälle getestet werden. Es wäre möglich, beide Bedingungen insgesamt positiv und negativ zu testen

(z. B. 1. und 2.). Damit wäre *C1* erreicht. Kombinationen von atomaren Bedingungen jeweils positiv und negativ zu testen, ist bei *C1* nicht mehr notwendig. Wer stellt nun sicher, ob die Möglichkeiten 3. und 4. die Absicht des Entwicklers ist?

Für diese Fälle gibt es die sogenannte *Condition Coverage*. Hier ist die Vorgabe, alle Kombinationen der durch boolesche Operationen zusammengesetzten atomaren Bedingungen zu testen. Es ergibt sich aber das Problem, dass die Anzahl der Kombinationen mit der Anzahl der atomaren Ausdrücke exponentiell wächst. Ein Test nach *Condition Coverage* kann nur praktikabel sein, wenn Abstriche bei den Kombinationsmöglichkeiten gemacht werden.

Das führt zur sogenannten *Modified Condition*. Abbildung 5.10 soll deutlich machen, dass eine *MC/DC* eine Erweiterung der *C1*-Abdeckung ist. *MC* steht für *Modified Condition* und *DC* für *Decision Condition*, wobei das Letztere genau *C1* entspricht.

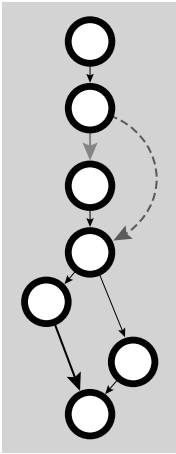


Abbildung 5.10 Modified Condition and Decision Coverage

Die folgende Liste zeigt Eingangsparameter für der Funktion `calc`, um die *Modified Condition* zu erfüllen:

1. ($x = 1$ und $y = 1$) is `True`
2. ($x = 0$ und $y = 0$) is `False`
3. ($x = 1$ und $y = 0$) is `False`

Das erste Element (1.) und zweite Element (2.) in der Liste sind Bedingungen, die *C1* bereits erfüllen. Hier zeigt sich, dass *C1* eine Notwendigkeit ist, um *MC/DC* zu erfüllen. *MC* aber untersucht auch Kombinationen von atomaren Bedingungen. Bei *MC* ist die Vorgabe, dass jede atomare Bedingung sowohl positiv als auch negativ sein muss und mindestens einmal unterschiedlich, z. B. $x = 1$ *is True* und $y = 0$ *is False*. Deswegen wurde dies als dritte Element (3.) zum Test hinzugefügt.

Bei `x = 0` bricht der Code der ersten `if`-Anweisung sofort ab und weist der kompletten Bedingung den Wert `False` zu, ohne dass die zweite Bedingung eine Rolle spielte. Es kommt dabei nicht zur Abfrage von `y > 0`. Somit wurde die `if`-Anweisung mit dem Eingangsparameter `y = 0` nicht durchlaufen.

Bei dem dritten Element (3.) der Liste vergleicht die erste atomare Bedingung `x > 0` den Eingangsparameter `x = 1`, bricht aber nicht ab, da die atomare Bedingung gleich `True` ist. Der Code überprüft die zweite atomare Anweisung `y > 0`, und mit dem Eingangsparameter `y = 0` stellt er fest, dass sie `False` ist. Das Ergebnis ist zwar wie bei (2.) gleich False, aber bei (3.) werden zwei atomare Bedingungen durchlaufen.

Bei der Norm *IEC-601508* und bei der Norm *ISO-26262* gibt es Empfehlungen für die Testabdeckungen bezogen auf die *SIL* oder die Automotive Sicherheitsintegritätslevels (*ASIL*). Bei den unteren Stufen von *SIL* und *ASIL* wird die *C0*-Abdeckung empfohlen. Bei mittleren *SIL* und *ASIL* wird die Zweigabdeckung *C1* empfohlen. Bei Funktionalitäten, die nach *SIL* 4 und *ASIL* 4 eingestuft werden, soll die *MC/DC* erreicht werden, siehe auch Tabelle 5.2.

Testabdeckung	SIL, ASIL 1	SIL, ASIL 2	SIL, ASIL 3	SIL, ASIL 4
C0	++	++	++	++
C1	+	++	++	++
MC/DC	+	+	+	++

Tabelle 5.2 Empfohlene Testabdeckungen

5.3.5 Blackboxtest

Das Testen von Software sollte nicht dem Softwareentwickler überlassen werden, denn dieser hat auf die selbst geschriebene Software einen anderen Blick als der Softwaretester. Der Softwareentwickler hat zu seinem Werk eine ähnliche Beziehung wie die Eltern zu ihrem Kind, was zu einer Verblendung führen kann.

Vermieden wird das, wenn eine unabhängige Person testet. Diese Person ist meist einer anderen Abteilung zugeordnet, sodass eine Nähe nicht unmittelbar gegeben ist. Natürlich hat die fehlende Nähe zwischen Entwickler und Tester den Nachteil, dass der Tester den Entwicklungsprozess der Software nicht direkt mitbekommt. Allerdings gibt es bei Software, die nach einem Prozess entwickelt wurde, Softwareentwicklungsdokumente, z. B. das *Low-Level-Design*-Dokument, auf den sich der Tester beziehen kann. Weitere Dokumente sind das *Software Safety Requirements*-Dokument, das Lastenheft etc. Der Tester wird aus diesen Dokumenten in Kooperation mit dem Safety Engineer auf höherer Ebene zunächst Testpläne und Integrationspläne entwickeln, z. B. den *Software Safety Validation*- und den *Software Architecture Integration Test*-Plan. Auf niedriger Ebene werden zusammen mit dem Entwickler die

Pläne *Software System Integration Test* und *Software Module Integration Test* erstellt. Im Allgemeinen wird in den Testplänen beschrieben, wer testet sowie wann und was getestet wird. Da nun der Tester die Codefunktionalität meist nicht kennt, muss die Software für ihn als eine Unbekannte betrachten werden.

Im Englischen wird diese als *Blackbox* bezeichnet (siehe auch Abbildung 5.11). Dem Tester bleibt nun nichts anderes übrig, als die Software, die Module und die Funktionen aus den oben genannten Dokumenten zu verstehen und Testmuster zu definieren, die sich aus den Dokumenten ergeben. Testmuster sind eine Folge von Handlungen, Anweisungen oder Befehlen, die eine bekannte Antwort der Software, der Module oder Funktionen erzeugt. Ein Testmuster ergibt also eine Testantwort, und die Gesamtheit der Testmuster ergibt eine Menge von Testantworten. Die Verifikation der Testmuster sind Arbeitspakete, denen nun ein Termin, eine Dauer und ein Tester zugeordnet werden. Durch die Dokumentation der Arbeitspakete entstehen die oben genannten Testpläne. Diese Testpläne sind ein Bestandteil des Sicherheitsnachweises und müssen deswegen dementsprechend abgelegt werden.

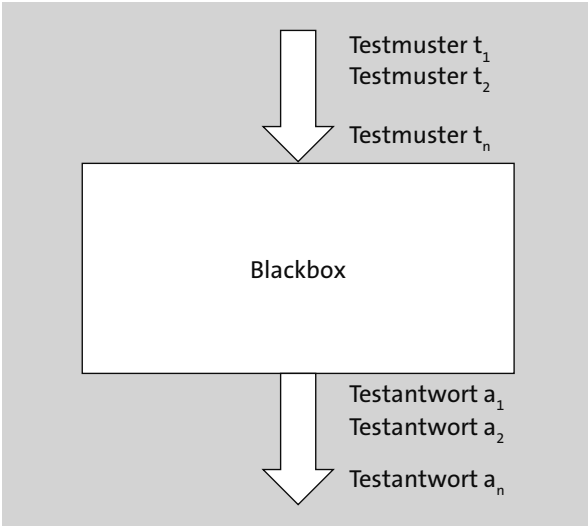


Abbildung 5.11 Blackboxtest

5.3.6 Leistungstest

Weitere Tests, die sich unter anderem aus dem *Software Safety Requirements*-Dokument und dem *Software Safety Validation*-Plan ergeben, sind Leistungstests. Hier wird unter anderem der Test der Softwareperformanz zusammen mit der benötigten Hardware, also z. B. den Rechnersystemen, Sensoren und Aktoren, geplant. Software benötigt Ressourcen, z. B. CPU, Speicher oder Kommunikationsschnittstellen. In vielen Fällen stellt der Ausfall eines Rechnersystems eine Gefahr dar, ein entsprechendes Sicherheitsziel nicht zu erreichen. Ausfälle können durch das Überstrapazieren

der Ressourcen verursacht werden. Demnach kann durch Überdimensionierung der benötigten Ressourcen Ausfälle vermieden werden. Der Tester muss nun ermitteln, ob diese Überdimensionierung ausreicht (siehe auch Abbildung 5.12). Es gibt z. B. in der Automobilindustrie die Vorgabe, dass bei kritischen Rechnersystemen nur maximal 60 % der Ressourcen (also CPU, Speicher) beansprucht werden darf. Durch entsprechende Werkzeuge können die tatsächlichen Auslastungen ermittelt werden. Nur wenn das System unter einem Schwellenwert (wie z. B. 60 %) liegt, gilt der Test als bestanden. Wie bei allen Testaktivitäten werden die Ergebnisse in Reports dokumentiert und sind somit Bestandteil des Sicherheitsnachweises.

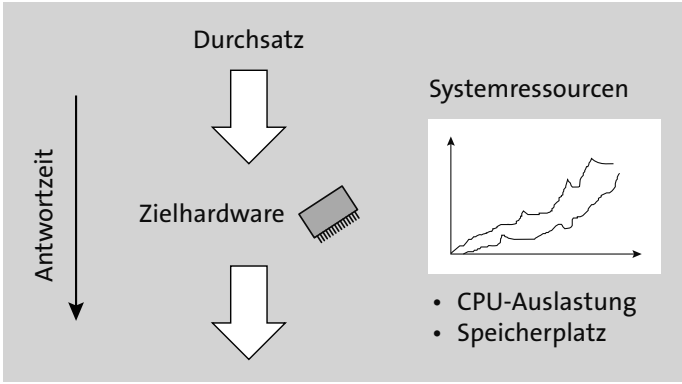


Abbildung 5.12 Leistungstest

5.3.7 Software und Hardwareintegration

Im Folgenden soll auf die Testpläne näher eingegangen werden. Der Inhalt der Pläne besteht aus Arbeitspaketen mit Terminen, Dauern und Namen der Tester, die den Test ausführen sollen. Arbeitspakete sind oftmals Softwaremodulen und deren Funktionalitäten zugeordnet.

In Abschnitt 5.2 wurde bereits auf die Phasen in der Softwareentwicklung eingegangen. Die Phasen werden in einem *V-Modell* dargestellt, das eigentlich in dieser Art in Projekten selten vorkommen. Dennoch ist das Modell nützlich zur Beschreibung der Prozesse und deren Abhängigkeiten. Nach der Codierung folgt die Modulintegration und der Modultest. Nicht alle Entwicklungsteams arbeiten zur selben Zeit am selben Projekt, auch wenn sie Arbeitspakete für das aktuelle Projekt zugeordnet bekommen. Projektplanung ist in der Regel der Programmplanung unterworfen. Das bedeutet, dass es Teams gibt, die an vorherigen oder an externen Projekten arbeiten. Dies erzeugt ein zeitlich versetztes Fertigstellen von Arbeitspaketen.

Auch innerhalb des Projekts sind einem Team normalerweise mehrere Arbeitspakete zugeordnet, wodurch das Team nicht alle Arbeitspakete gleichzeitig bearbeiten kann. Zudem beansprucht die Bearbeitung von Prozessanforderungen, z. B. Architektur-

arbeiten, Reviews oder Audits, Zeit. Für das Integrations- und Testteam bedeutet das, dass nicht alle Arbeitspakete zur selben Zeit eintreffen. Auch das Testteam ist Ressourceneinschränkungen unterworfen. In der Regel besteht bei größeren Firmen eine Testabteilung aus mehreren Testteams, und nicht alle Teams arbeiten gleichzeitig an einem Projekt. Dafür benötigt es die Planung des Projekt- und Programmmanagements . Diese geht ein auf die Testspezifikation, *Software Architecture Integration Test*-Plan, *Software System Integration Test*-Plan und *Software Module Integration Test*-Plan.

Nicht nur die Tests der Arbeitspakete wird in den Plänen dokumentiert, sondern auch deren Abhängigkeiten und die Integration der Arbeitspakete in ein Gesamtsystem. Die ersten Schritte werden auf Modulebene beschrieben (angefangen mit dem *Software Module Integration Test*-Plan). Modul A und Modul B beispielsweise haben Abhängigkeiten, z. B. ruft Modul A das Modul B auf. Der Projektmanager der Testabteilung muss nun von der Entwicklungsabteilung einfordern, ein integriertes Modul (ein neues Modul C, das die Module A und B enthält) zu einem bestimmen Termin zu liefern, da er weiß, dass er zu diesem Zeitpunkt Ressourcen für den Test frei hat. Die Entwicklungsabteilung ihrerseits muss überprüfen, ob es Ressourcen für die Zusammensetzung der Module gibt, da die Integration der beiden Module Zeit braucht. Die Projektleitung, die Entwicklungsabteilung und die Testabteilung einigen sich in Planungsmeetings und Workshops auf einen Termin. Diese ersten Integrationsschritte werden somit als Ergebnis der Meetings in der Testspezifikation und im *Software Module Integration Test*-Plan dokumentiert. Die Planung der nächsten Integrationsschritte folgt darauf, bis ein weitergehender Integrationsplan für die gesamte Software entsteht. Termine, Dauer und Verantwortliche werden in dem *Software Module Integration Test*-Plan dokumentiert.

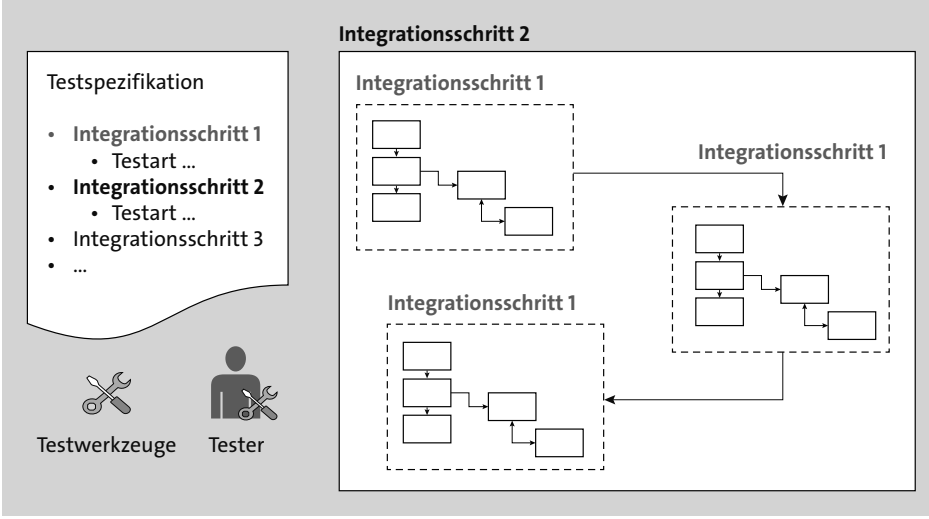


Abbildung 5.13 Integrationstest mit Software

Abbildung 5.13 veranschaulicht dieses Vorgehen. Wie bei den zuvor erwähnten Dokumenten ist die Testspezifikation Teil des Sicherheitsnachweises.

Eine weitere Komplexität tritt auf, wenn Hardware in das Gesamtsystem integriert wird. Hier steigt der Aufwand der Planung, da es jetzt weitere Teams gibt (z. B. Softwareteam, Hardwareteam und Testteam), die sich auf einen Gesamtplan einigen müssen. Oben gab es zwischen den Teams nur zwei Kommunikationsrichtungen, diese sind jetzt bei drei Teams auf sechs Richtungen angewachsen. Um die Abhängigkeiten zu verringern, verwendet vor allem das Softwareteam Simulationen, die die Hardware zumindest bei der Entwicklung vorübergehend ersetzt. Simulationen, die auf Rechnersystemen laufen, können von Softwareteams erstellt werden. Dies braucht ein Verständnis der Hardware, das aber ohnehin benötigt wird. Somit hat die Entwicklung der Simulation auch den Vorteil, dass das Verständnis der Hardware verbessert wird. Auch das Testteam hat davon einen Nutzen, da es sich von der Hardwareabteilung anfangs entkoppelt und zum Testen der Software die Simulation nutzt. Früh können auf diese Weise Softwarefehler gefunden und behoben werden, ohne dass eine einzige Hardwarekomponente zur Verfügung steht.

Für diese Arbeitsweise sollten natürlich Arbeitspakete für die Erstellung der Simulation eingeplant werden. Ein Meilenstein (z. B. *Integration mit Simulation*) kann dabei nützlich für die Projektleitung sein. Sobald die Hardware zur Verfügung steht und die Integration der Hardware zusammen mit der Software erfolgt ist (Integrationstest Architektur im *V-Modell*), wird der Meilenstein *Integration mit Hardware* erreicht. Die Testspezifikation beschreibt daher Integrationsschritte mit Simulation und Integrationsschritte mit Hardware. Abbildung 5.14 zeigt schematisch den beschriebenen Zusammenhang.

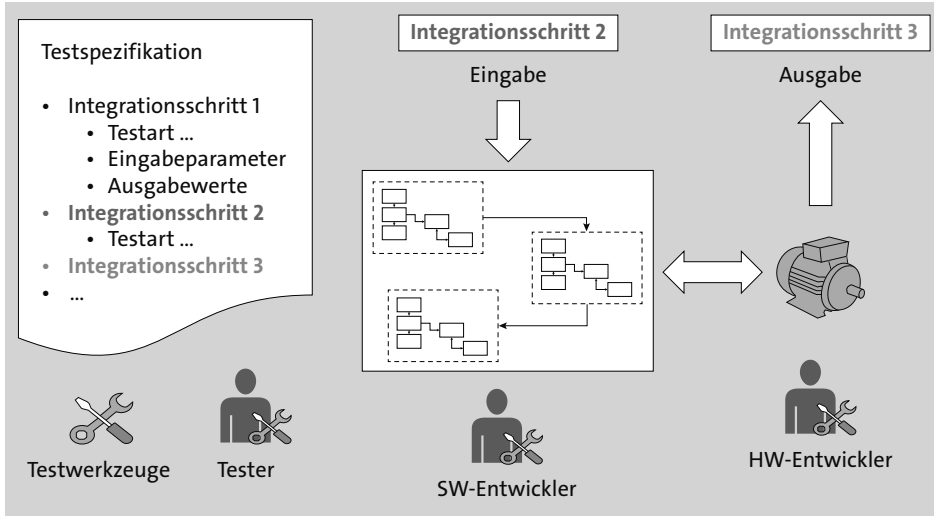


Abbildung 5.14 Integrationstest mit Hardware

Bezüglich des Sicherheitsnachweises ist es notwendig, den Gebrauch des Systems durch das Testteam zu validieren. Die Planung wird in dem *Software-Safety-Validation*-Plan dokumentiert. Eine Validierung ist der Nachweis von Gebrauchsmustern. So wird der Gebrauch des sicherheitstechnischen Systems getestet und mit den Anforderungen verglichen, um herauszufinden, ob diese erfüllt sind. Diese Anforderungen sind in den *Software Safety Requirements* dokumentiert. Die Validierung bedarf dabei einer Testvorbereitung, z. B. müssen eine Testumgebung und deren Werkzeuge zur Validierung definiert sein. Der Tester dokumentiert die Art des Verfahrens zur Validierung und gibt die Eingangsparameter an. Als Resultat der Validierung erhält der Tester Ergebnisse, die mit den Anforderungen verglichen werden. Die Beurteilung des Testers zum Ausgang der Validierung wird in Reports dokumentiert. Abbildung 5.15 zeigt mögliche Dokumentationspunkte für die Validierung der Sicherheitsanforderungen.

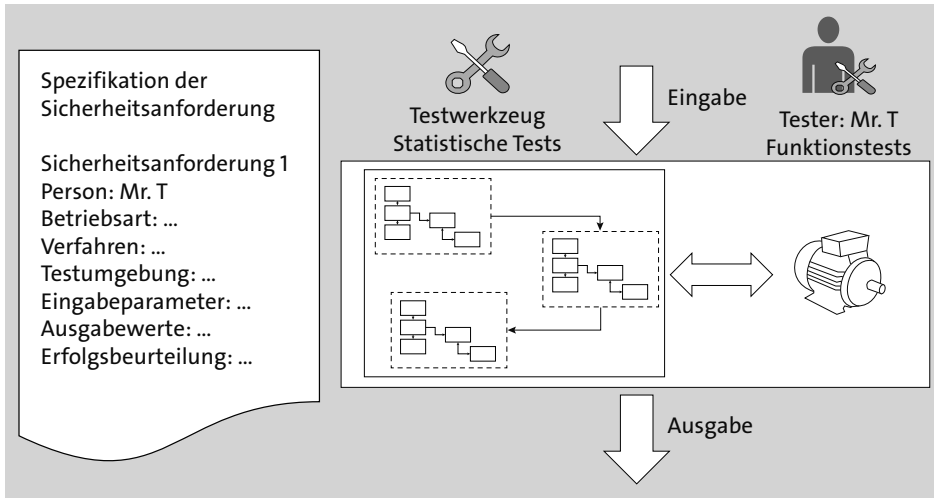


Abbildung 5.15 Sicherheitsvalidierung

5.3.8 Ticketmanagementsystem

Wesentlich beim Test ist die Dokumentation der Testfortschritte. Bei einer inoffiziellen Kommunikation der Fehler zwischen Tester und Entwickler, z. B. über E-Mail, geht schnell der Überblick verloren. Auch ist die Möglichkeit des Nachweises kaum gegeben. Der verantwortliche Safety Engineer, das Entwicklungsteam und die Personalmanager brauchen den Überblick über den Stand des Projekts. Abhilfe schafft da ein Ticketmanagementsystem. Hier haben Tester und Entwickler (mit Abhängigkeiten zu anderen Entwicklern) die Möglichkeit, über eine Benutzerschnittstelle Datenbankeinträge (genannt *Tickets*, *Bugs* oder *Issues*) zu erzeugen, um ein Problem in der Software zu adressieren. Diese *Tickets* sind immer an eine Person gerichtet, die für das

beschriebene Problem verantwortlich sein soll. An die Tickets sind stets Problembeschreibungen angehängt, aber auch Daten, wie z. B. Logging-Dateien, um das Problem weiter analysieren zu können. Fühlt sich der Entwickler nach der Analyse der Problembeschreibung und der Dateien verantwortlich, behebt er das Problem mit einer Softwarekorrektor (oft *Fix* genannt). Dieser durchläuft den Softwareentwicklungsprozess (*CI-* oder *DevOps*), damit die Problembehebung bestätigt werden kann. Der Fix geht in den Hauptentwicklungszweig des Repository. Der Erzeuger des Tickets kann dem Problem immer eine Priorität zuweisen, abhängig davon, wie sehr seine Arbeit andere behindert.

Hoch prioritäre Tickets erhalten in der Regel die Aufmerksamkeit des Safety-Verantwortlichen oder Personalmanagers. Dies hat eine sehr wichtige Funktion. Der Entwickler weiß beim Erhalt des Tickets, wo seine Priorität liegt, und der Personalmanager weiß, wie weit die Entwickler seiner Abteilung mit Problemlösungen beschäftigt sind. Die Idee dahinter sollte nicht die Kontrolle des Entwicklers, sondern die Einschätzung sein, ob der Entwickler Hilfe bei der Problemlösung benötigt. Hilfe kann z. B. das Bereitstellen von Freiraum (oder Ruhe) sein, das Wegräumen von Hindernissen (Urlaubsvertretungen) oder das Beschaffen von dringend benötigter Information.

Abbildung 5.16 zeigt schematisch ein Ticketmanagementsystem mit seinen Benutzern und deren Interaktion.

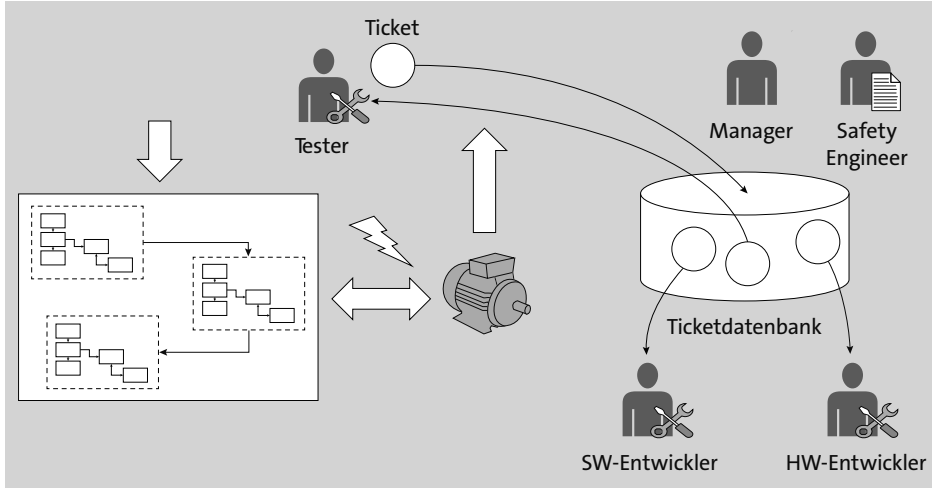


Abbildung 5.16 Dokumentation von Fehlern im Ticketmanagementsystem

Nicht in allen Fällen hat das Hardwareteam Zugang bzw. geübten Umgang mit Ticketmanagementsystemen, da Repositories in der Regel aus der Softwaretechnik stammen. Dennoch gibt es klare Vorteile, wenn über die Abteilungsgrenzen hinweg ein

einheitliches Ticketmanagementsystem verwendet wird. Wichtig für den Sicherheitsnachweis ist es, die Tickets zu dokumentieren, mit denen Funktionen, Module, Software und Architektur getestet bzw. validiert wurden. Da diese Information im Ticketmanagementsystem vorhanden ist, können automatisch Reports aus den Ticketidentifikationsnummern erzeugt werden.

5.3.9 Konfigurationsmanagementsystem

In dem oben beschriebenen Prozess entstehen Dokumente, Quellcode, Hardwarebeschreibungen, Simulationswerkzeuge etc. Sie sind Teil des Sicherheitsnachweises, wenn sie Beschreibungen über sicherheitstechnische Funktionen enthalten. Es stellt sich die Frage, wo diese Dokumente aufbewahrt werden sollen. Spätestens der Assessor will sie bei einem Audit schnell zur Hand haben, wenn dieser unangekündigt ist. Das verzögerte Heraussuchen und Zusammenstellen der Dokumente kann ein erheblicher Arbeitsaufwand sein. Anzahl und Art der Dokumente können eine komplizierte Verwaltung hervorrufen. Denn in der Laufzeit eines Produkts gibt es immer wieder Änderungen der Anforderungen bei Software und Hardware. Nach der Implementierung der Änderungen strebt die Testabteilung die Ausgabe eines Produkt-Updates an. Die Änderungen werden so weit es geht nach den alten, aber adaptierten Testplänen getestet.

Ein umfassender Test kann sehr teuer sein, da Zeit und Personal dafür aufgebracht werden müssen. Dennoch ist es bei einem Produkt-Update die Regel, dass auch neue Anteile von Software entstehen. Für die neuen Softwareanteile müssen wiederum Architekturdokumente, Testpläne, Testreports auf den neuesten Stand gebracht und derart abgelegt werden, dass sie einer neuen Version zugeordnet werden können. Es entstehen also nicht komplett neue Versionen eines Produkts, sondern sie basieren meistens auf Software- und Hardwarebeständen der alten Produkte. Dadurch kann es eine große Anzahl von Produkt-Updates und Versionen geben. Die Informationsflut der Dokumente muss deswegen verwaltet werden, wofür ein Konfigurationsmanagementsystem hilfreich sein kann. Die Situation wird in Abbildung 5.17 gezeigt.

Ein Konfigurationsmanagementsystem ist eine Art Datenbank mit entsprechender Benutzerschnittstelle, um Dokumente und Werkzeuge nach Versionen und Produkten zu speichern. Die Funktionalität entspricht einem Repository, wobei dieses meist nur für Quellcode-Dateien eingesetzt wird. Konfigurationsmanagementsysteme verwalten etwa die Architektur- und Sicherheitsnachweisdokumente, aber auch Referenzen von Versionen der eingesetzten Softwarewerkzeuge. Sie verwalten und versionieren z. B. Word-Dateien (also nicht nur Textdateien), Softwarewerkzeuge (Compiler), Simulationswerkzeuge oder komplette Testumgebungen (z. B. das *Image* eines Betriebssystems). Das Ziel ist hier, alle Dokumente und Werkzeuge einer kompletten

Entwicklungs- und Testumgebung wiederherstellen zu können, um bei einem Audit schnell alle Dokumente zur Verfügung zu haben. Auch bei Auftritt eines Problems beim Kunden sollen schnell die notwendigen Entwicklungswerkzeuge zur Verfügung stehen.

Ein Begriff aus der Softwareentwicklung ist *Infrastructure as Code* (*IaC*), bei der die gesamte Infrastruktur mit ihren Entwicklungswerkzeugen über Konfigurationsdateien definiert werden können. So kann über entsprechende Werkzeuge die komplette Infrastruktur versioniert und wiederhergestellt werden. Ein Vertreter dieser Werkzeuge ist *Terraform*.

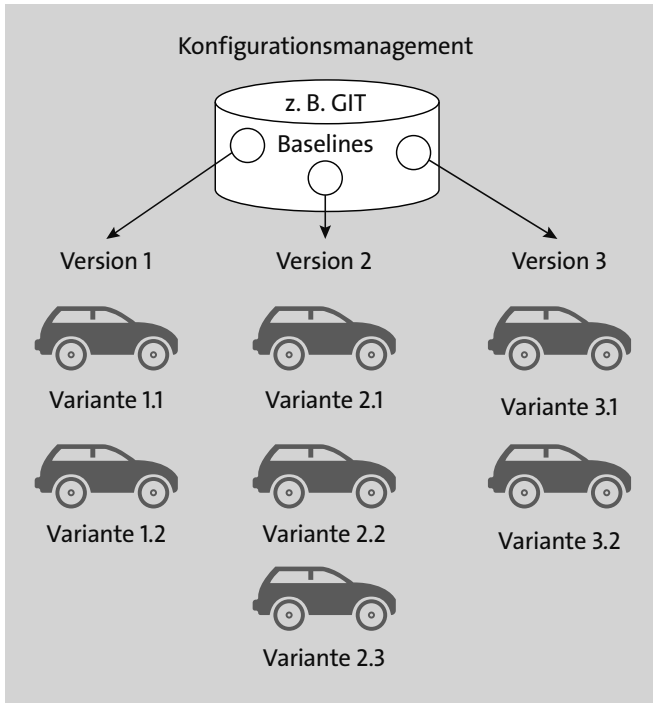


Abbildung 5.17 Konfigurationsmanagement

5.4 Überblick über Entwicklungspläne und Testpläne

In Tabelle 5.3 erhalten Sie einen Überblick über Dokumente zur Entwicklung von sicherheitsgerichteter Software, die Teil des Sicherheitsnachweises sind. Diese beziehen sich auf die Ausführungen innerhalb dieses Kapitels. Eine vollständige Liste finden Sie in *IEC-61508*.

Lebenszyklus	Dokument
Planung	*Software Safety Requirements*-Dokument
Entwicklung	*Software Architecture Design*-Dokument *Software System Design*-Dokument *Software Module*-Dokument
Integration	*Software Module Integration Test*-Plan *Software System Integration Test*-Plan *Software Architecture Integration Test*-Plan Reports
Validierung	*Software Safety Validation*-Plan Reports

Tabelle 5.3 Dokumente für sicherheitsgerichtete Systeme bei der Softwareentwicklung

5.5 Softwareentwicklungsprozess und Bauplan

Die Softwareentwicklung durch einen einzelnen Programmierer kann sehr einfach sein. Diese Person hat die komplette Kontrolle über seine Software, und Absprachen mit anderen Teammitgliedern kommen nicht vor. Da Software viel zu komplex oder umfangreich ist, um nur von einer Person hergestellt zu werden, entspricht dies nicht der Realität. Softwareentwicklung entsteht überwiegend innerhalb eines oder mehrerer Teams. Das bedeutet, dass Absprachen und Koordinierungen einen wesentlichen Anteil bei der Entwicklung haben. Unmittelbar sichtbar ist dieser vermeintlich unproduktive Anteil im Endprodukt nicht.

Es kommt natürlich ständig vor, dass Teammitglieder Fehler machen und sich nicht an Absprachen halten. Auch sind Teile der Software nicht gleichzeitig fertig und müssen in Zwischenschritten in die integrierte Software eingebunden werden. Das liegt daran, dass Ressourcen meist nicht ausreichend vorhanden sind und Teile der Software nur sequenziell von den Entwicklern bearbeitet werden können. Die Koordinierung mit den Teammitgliedern wird umso wichtiger, je stärker die Größe des Teams steigt. In der Softwareentwicklung gibt es bereits Werkzeuge, um die Teammitglieder bei der Koordinierung zu unterstützen. Die Werkzeuge werden normalerweise in einen Softwareentwicklungsprozess eingebettet, um so ein Regelwerk für die Zusammenarbeit zu etablieren. Die Werkzeuge des Softwareentwicklungsprozesses müssen auf Servern installiert werden, damit alle Teammitglieder darauf zugreifen können. Die logische und physikalische Einordnung der Werkzeuge auf die Server wird Bauplan genannt.

5.5.1 Softwareentwicklungsprozess

Zur umfassenden Einarbeitung in dieses Thema ist Artikel [23] geeignet. Er zeigt umfassend den aktuellen Stand der Technik und stellt die Unterschiede zwischen *CI*, *CDE* und *CD* gegenüber. Darauf aufbauend wird in diesem Kapitel ein möglicher Softwareentwicklungsprozess beschrieben, der bei *CDE* oder *CD* anzusiedeln ist. Die Technologie (oder das eingesetzte Softwarewerkzeug) spielt hier nur eine Nebenrolle. Zum Beispiel wird Git als Repository oft erwähnt, weil es in der Softwareentwicklung weit verbreitet ist. Allerdings gibt es auch andere Werkzeuge, die Git ersetzen können.

Auf die Beschreibung zur Anwendung der Werkzeuge, um den Softwareentwicklungsprozesses zu implementieren, wird weitgehend verzichtet, da hier nur die Grundideen aufgezeigt werden sollen. Wenn ein Prozess für die Softwareentwicklung konkret aufgebaut werden soll, empfehle ich Ihnen, sich auf den Webseiten von *GitHub Actions* und *Azure Pipelines* oder in der weitergehenden Literatur darüber zu informieren.

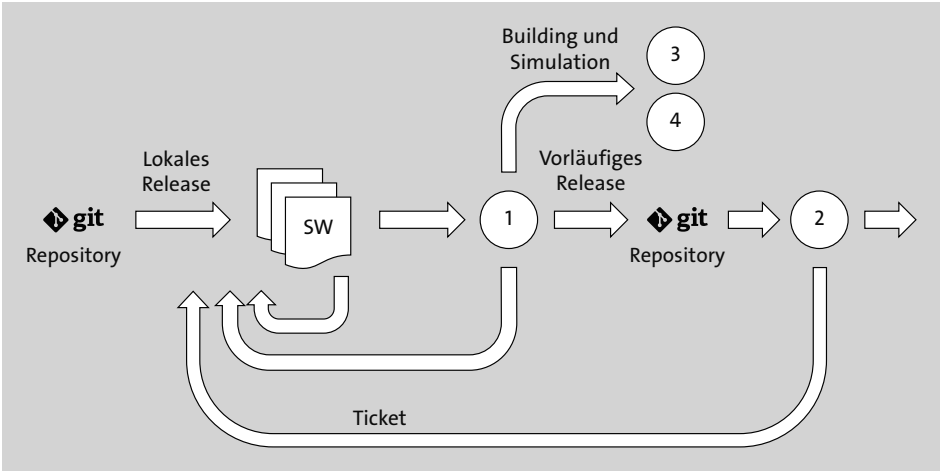


Abbildung 5.18 Softwareentwicklungsprozess beim Entwickler

Abbildung 5.18 zeigt den Anfang des Softwareentwicklungsprozesses. Einer der ersten Schritte ist das Aufsetzen einer Versionsverwaltung (in der Informatik wird diese meist *Repository* genannt). Das ist eine Datenbank mit Benutzerschnittstellen (und anderen Funktionalitäten), um Quellcodes der Entwickler zu speichern und zu versionieren. Ein prominentes Beispiel für ein Repository ist Git. Der Name ist kein Akronym, sondern bedeutet im Englischen so etwas wie »törichter Mensch«.

Es gibt natürlich unzählige käuflich zu erwerbende Repository, aber darauf soll hier nicht weiter eingegangen werden. So kann am Anfang der Entwicklungsphase das Repository ohne Inhalt sein. Es entstehen oftmals aber auch Folgeprojekte aus früheren Projekten, sodass das Repository mit dem Quellcode aus einem früheren Projekt be-

füllt ist. Am Anfang der Entwicklungsphase entsteht entweder neuer Quellcode, oder bestehender Quellcode wird von den Softwareentwicklern angepasst. Das Testteam ist in dieser frühen Phase noch nicht involviert (es hat in der Regel in dieser Phase noch mit aktuell laufenden Projekten zu tun).

Sie stellen sich vielleicht an dieser Stelle die Frage, woher der Softwareentwickler weiß, was er zu tun hat. Eine abschließende Antwort auf diese Frage gibt es nicht. Es kommt darauf an, wie die Firmenkultur dies handhabt. Der gelebte Projektmanagementprozess in der Firma ist dabei maßgeblich. So gibt es klassische, aber auch agile Projektmanagementmethoden. Ausgangspunkt sind immer die Anforderungsdokumente, aus denen die Arbeitspakete hervorgehen. Zum Beispiel kann der Architekt (bei agiler Entwicklung das Entwicklungsteam) *Tickets* für die zu entwickelnden Module schreiben und sie einem Entwickler oder einem Team zuweisen. So sind die Arbeitspakete und ihre Zuweisungen an die Verantwortlichen nicht nur in den Dokumenten, sondern auch im Ticketmanagementsystem dokumentiert. Tickets sind für Entwickler tägliches Geschäft, Anforderungsdokumente dagegen bei Weitem nicht in diesem Maße. Außerdem sind Tickets für Entwickler, Projektmanager und Manager einsehbar, um den aktuellen Entwicklungsstand zu bestimmen.

Die Entwicklung von Software fängt meist damit an, dass der Entwickler den kompletten Quellcode des Projekts – oder einen Teil davon – vom Repository auf die lokale Festplatte herunterlädt (der Begriff dazu heißt in der Softwaretechnik *Auschecken*), um darauf die Entwicklungsaktivitäten auszuführen. Entscheidet der Softwareentwickler, einen Teil seines Quellcodes im Repository abzulegen (in der Informatik wird dafür der Begriff *Einchecken* verwendet), sollten idealerweise erst Werkzeuge zur Kontrolle der Qualität eingesetzt werden. In der Liste sind mehrere Werkzeuge zur Qualitätskontrolle bzw. zur Qualitätsverbesserung angegeben, die zum Teil bereits in Abschnitt 5.2.3 beschrieben wurden.

- Quellcode-Analysator (*Sonarqube*, *lint*)
- *Coverage*-Analysator (*coverage*, *sourceforge*)
- Quellcode-*Beautifier* (*atype*)
- *Unit-Tests* (*cppunit*, *junit*)
- etc.

Es gibt Werkzeuge (z. B. *lint, atype*), die keinen lauffähigen Quellcode benötigen. Für andere Werkzeuge, z. B. für die Ausführung von *Unit-Tests*, müssen Teile des Quellcodes kompiliert und lauffähig sein. Wie der Entwickler seine Unit-Tests aufsetzt, obliegt ihm selbst. Hier werden Funktionen des Quellcodes in kleinen Testroutinen mit fixen Eingangsparametern aufgerufen, und dann werden die Rückgabewerte kontrolliert. Vollständig sind die Unit-Tests erst dann, wenn alle Funktionen aufgerufen und überprüft wurden. Die Entwickler können die Testroutinen der Unit-Tests in zentrale Dateien eintragen. Die zentralen Dateien werden nach der Kompilierung des Quell-

codes aufgerufen, um nicht nur die Unit-Tests des einzelnen Entwicklers, sondern die des gesamten Entwicklungsteams aufzurufen. Gibt es einen Fehler, wird das Einchecken ins Repository verhindert. Der Entwickler muss also nachbessern, bis alle Unit-Tests erfolgreich bestanden sind.

Ähnlich kann dies mit dem Quellcode-*Analysator* durchgeführt werden. Nur wenn dieser fehlerfrei auf den Quellcode anzuwenden ist, erhält der Entwickler die Freigabe zum Ablegen seines Quellcodes in das Repository. Ein weiteres Werkzeug, das vor dem Einchecken angewendet werden kann, ist der Quellcode-*Beautifier*. Dieses Werkzeug trägt zur Qualitätsverbesserung bei, indem die Lesbarkeit verbessert wird. Es kann automatisch aufgerufen werden und verändert den Quellcode selbstständig. Da der Softwareentwicklungsprozess das Werkzeug selbstständig aufrufen kann, ist die Freigabe zum Einchecken nicht erforderlich.

Eine weitere Kategorie von Werkzeugen beinhaltet die, die nur bei einem lauffähigen Programm angewendet werden. Ein lauffähiges Programm ist in der Regel nicht direkt am Anfang des Projekts vorhanden, da die Entwickler erst eine Quellcode-Grundsubstanz, die aus den Anforderungsdokumenten hervorgeht, entwickeln müssen. Sie arbeiten aber so lange darauf zu, bis Werkzeuge wie der *Coverage*-Analysator zum Einsatz kommen können. Wie bei den Unit-Tests werden in zentralen Dokumenten Eingaben definiert, um zu kontrollieren, ob jede Stelle bzw. Verzweigung im Quellcode durchlaufen wurde. Auch hier lässt sich in vielen Fällen vor dem Einchecken ein Test durchführen, der die Freigabe erteilt. In Abbildung 5.18 wird die Freigabe durch Punkt 3 angedeutet. Den Prozess kann der Entwickler bestenfalls lokal selbst anstoßen, um zu kontrollieren, ob sein Quellcode die geforderte *Coverage* hat.

Bei sicherheitsgerichteten Systemen kann Hardware ein wichtiger Bestandteil des Systems sein. Software wird zur Kontrolle oder Steuerung der Hardware genutzt. Dem Entwickler steht aber die Hardware nicht von Anfang an zur Verfügung – zum Teil weil sie noch nicht entwickelt wurde, zum Teil weil der Entwickler keinen Zugang zu ihr hat. Allein auf Anforderungs- und Architekturdokumenten die Entwicklung aufzubauen, ist sehr schwierig. So kann, wie im Fallbeispiel in Abschnitt 5.1 beschrieben wurde, der Autor des Anforderungsdokuments Begriffe verwenden, die dem Softwareentwickler nicht bekannt sind, und das kann zu Missverständnissen und fehlerhaftem Quellcode führen. Vorteilhaft ist deswegen eine Simulation der Hardwareumgebung, an der der Softwareentwickler seine entwickelte Software austesten kann. Dies benötigt natürlich Simulationssoftware, die in vielen Fällen durch ein eigenes Team entwickelt werden muss. Dieses Team gilt dann als Schnittstelle zwischen Hardwareteam und Softwareteam. Idealerweise bestehen die Teammitglieder aus beiden Bereichen.

Die Simulationssoftware kann mit Testeingaben parametrisiert und zusammen mit dem entwickelten Quellcode anstelle der Hardware getestet werden. Die Ausgaben der Simulation werden mit vordefinierten Ergebnissen verglichen. Wird neue Soft-

ware des Entwicklers durch die vorhandenen Testmuster nicht angesprochen, muss der Entwickler diese in zentralen Dokumenten erweitern. Ein Testlauf der Simulation kann also ein erfolgreiches und ein nicht erfolgreiches Ergebnis haben. Abhängig davon wird die Freigabe für das Einchecken des neuen Quellcodes in das Repository erteilt. Die Simulation wird in Abbildung 5.18 durch Punkt 4 dargestellt.

Die Einrichtung von Simulationen kann eventuell sehr aufwendig und zeitintensiv sein. Beispielsweise wird in der Halbleiterindustrie die Logik von Halbleiterelementen, bevor die Masken zur Belichtung entwickelt werden, in Simulationen getestet. Da bei Halbleiterelementen Millionen von Gattern simuliert werden müssen, braucht es spezielle Simulationsrechner, die eigens dafür entwickelt werden. Solche Rechner stehen nicht unbegrenzt zur Verfügung, und die Simulationen können unter Umständen sehr lange dauern. Deswegen wird die Simulationssoftware in mehreren Abstufungen bzw. Simulationstiefen angeboten, aus denen die Entwickler wählen können. Die Einbindung von zeitintensiven Simulationen in den Softwarebauprozess kann unter Umständen wenig praktikabel sein.

Zusammenfassend, können folgende Punkte Bedingungen sein, damit der Entwickler seinen Quellcode in ein vorläufiges Release des Repository einchecken darf.

- Bauprozess (Kompilierung)
- Softwareanalysen, siehe Liste oben
- Simulationen
- etc.

Diese Punkte lassen sich mit technischen Mitteln innerhalb des Softwareentwicklungsprozesses umsetzen. Es verhindert schlampiges Arbeiten seitens der Entwickler und trägt in erheblichem Maße zur Qualitätsverbesserung bei.

Wurde der neu entwickelte Quellcode in einem vorläufigen Release eingecheckt, folgt ein sehr wichtiger Schritt. Der neu eingecheckte Quellcode sollte von weiteren Entwicklern begutachtet werden (auch oftmals *Review* genannt). Dazu kann der Softwareentwicklungsprozess automatisch dem Entwickler zwei oder drei Reviewer vorschlagen.

Reviewer sind in der Regel erfahrene Softwareentwickler aus unterschiedlichen Teams der gleichen Abteilung. Der Entwicklungsprozess fordert automatisch die Reviewer auf, über ein Werkzeug den Quellcode zu begutachten. Diese Werkzeuge sind in der Lage, die Unterschiede des Quellcodes im Vergleich zum vorherigen Stand hervorzuheben, damit die Reviewer eine vereinfachte Sicht auf die neuen Quellcode-Stellen erhalten. Das Werkzeug erlaubt den Reviewern, Problemstellen im Quellcode durch Einträge hervorzuheben. Der Entwickler muss diese Problemstellen korrigieren, was aber wieder bedeutet, dass die Schritte in der Liste oben nochmals durchlaufen werden müssen, damit der Quellcode ein weiteres Mal in das vorläufige Release

eingecheckt und der Review-Prozess wieder angestoßen wird. Bei Entwicklern mit wenig Erfahrung kann sich dies mehrere Iterationen wiederholen, und zwar so lange, bis alle Reviewer ihr Einverständnis gegeben haben. Es gibt mehrere Werkzeuge, die diesen Review-Prozess unterstützen. Zum Beispiel hat *GitHub* bzw. *GitLab* bereits diese Funktionalität. Das Werkzeug *Gerrit* wird ausschließlich dafür verwendet, hat aber vereinfachte Einbindungsmöglichkeiten für *Git*.

Nach Erhalt der Freigabe von allen Reviewern wird das vorläufige Release des Entwicklers zusammen mit vorläufigen Releases anderer Entwickler zu einem Softwarestand für die Komplettsimulation integriert. Diese wird dann an den Test mit der Simulation (Punkt 4) weitergereicht, siehe Abbildung 5.19.

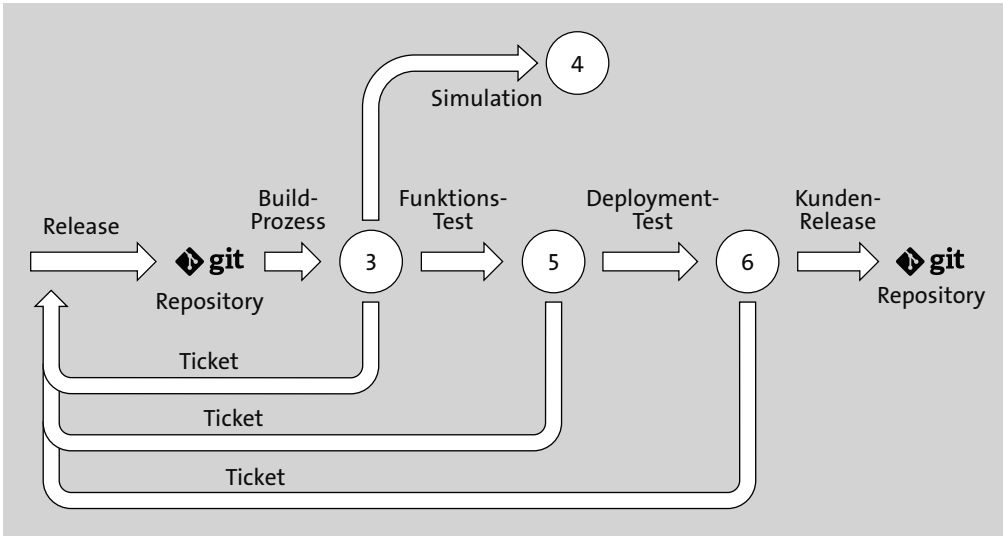


Abbildung 5.19 Softwareentwicklungsprozess bei Integration und Test

Bei der Softwareintegration wird zuerst der neueste Softwarestand gebaut (Punkt 3), was unter Umständen wieder Fehler hervorrufen kann. Das liegt weniger an dem eingecheckten Code der einzelnen Entwickler, sondern vielmehr an den möglichen Abhängigkeiten der vorläufigen Releases anderer Entwickler, die nicht ausreichend berücksichtigt wurden. So kann das Softwareintegrationsteam die Entscheidung treffen, problembehaftete Releases nicht in den neuesten Softwarestand aufzunehmen. Der verantwortliche Entwickler des problembehafteten Release erhält ein Ticket mit der Aufforderung, seinen Quellcode nachzubessern. In der Abbildung wird das durch den Rücksprung in Punkt 3 dargestellt. Bei konsequenter Befolgung des Prozesses muss der Entwickler alle Schritte zuvor wiederholen!

Während bzw. nach dem Bau der Software werden die Analyseschritte, beschrieben in der Liste oben, wiederholt. Auch die Simulationen, beschrieben durch den Punkt 4, kann nochmals durchgeführt werden. Erst nach erfolgreichen Tests der Werkzeuge

entsteht ein Release für das Testteam. Mit diesem Softwarestand kann es dann seine zuvor dokumentierten Tests durchführen. Oftmals werden auch hier Fehler entdeckt, die über Tickets an die Entwickler adressiert werden. Als kurzfristige Übergangslösung kann der Entwickler aber einen Quellcode-Fix dem Testteam zur Verfügung stellen. Damit jedoch der Quellcode dauerhaft repariert wird, muss der Prozess vom Entwickler über die Verwendung des Tickets von vorne wieder angestoßen werden.

Nachdem der Softwarestand vom Testteam eine Freigabe erhält, kann er an das *Deployment*-Team weitergereicht werden, siehe Punkt 6 in der Abbildung. Das Deployment-Team kann in das Testteam integriert sein, kann aber auch unabhängig von diesem agieren. Dieses Team zeichnet sich durch Kundennähe und Produktionsnähe aus. Seine Verantwortung liegt meist in der Kundenbetreuung, zu der auch die Installationen von neuen Softwareständen beim Kunden bzw. in der Produktion gehört. Auch dieses Team führt Tests durch, wobei sich diese hauptsächlich auf Installation (*Deployment*) und Anwendung beschränken. Das Deployment-Team kann Tickets an die Entwickler bei Auftreten von Fehlern schreiben, die sehr oft eine hohe Priorität haben.

In vielen Fällen wird bei bestandener Prüfung durch alle Instanzen der Softwarestand freigegeben und durch ein Kundenrelease dem Kunden zur Verfügung gestellt (siehe auch Abbildung 5.20). So kann der Kunde entweder selbstständig den Softwarestand auf sein System aufspielen (z. B. nach einer Benachrichtigung), oder er nutzt den Service des Deployment-Teams als Dienstleistung.

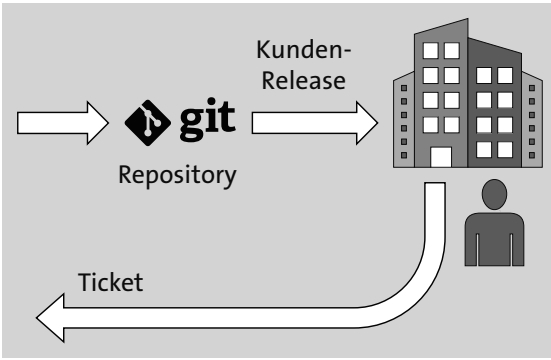


Abbildung 5.20 Softwareentwicklungsprozess zum Kunden

Auch der Kunde selbst kann unter Umständen Fehler in der Software finden, den er mit einem Ticket adressieren kann, falls er an das Ticketmanagementsystem angebunden ist. Der Entwickler wird auch hier in diesem Fall den ganzen Prozess wieder durchlaufen müssen. Oftmals sind vom Kunden adressierte Tickets durch eine hohe Priorität gekennzeichnet.

5.5.2 Bauplan

In den letzten zwei Jahrzehnten hat das *Cloud-Computing* in der Form von *Infrastructure as a Service* (*IAAS*), *Platform as a Service* (*PAAS*) oder *Software as a Service* (*SAAS*) Einzug gehalten. Bei *IAAS* wird die Hardware durch einen Serviceanbieter gegen eine Gebühr zur Verfügung gestellt. Damit muss sich der Nutzer nicht mehr mit Update- und Skalierungsaktivitäten der Hardware und Betriebssysteme beschäftigen. Dennoch wird der Nutzer nach wie vor seine Softwarekomponenten selbst installieren und verwalten. Bei *PAAS* erweitert der Anbieter das Angebot durch Plattformen wie z. B. Datenbanken, Ticketmanagementsysteme, Konfigurationsmanagementsysteme etc. Der Nutzer bekommt den Zugriff über das Internet und muss sich nicht mehr um Updates und Skalierungen der installierten Plattformen kümmern. So stehen bereits Sicherheitspakete dem Nutzer zur Verfügung, die sich direkt einsetzen lassen. Lediglich der Softwareentwicklungsprozess muss vom Nutzer definiert und implementiert werden. Dafür stehen Werkzeuge wie z. B. *Azure Pipelines*, *GitHub Actions* etc. zur Verfügung. Bei *SAAS* kann der Nutzer die komplette Infrastruktur eines Softwareentwicklungsprozesses anwenden. Er muss sich also nicht mehr um die Implementierung kümmern, da der Anbieter diese Aufgabe übernimmt. Weitere Dienstleistungen des Anbieters können z. B. Installation, Wartung, Sicherheit, Skalierung und Prozessdefinition sein.

So hat der Nutzer die Wahl, eine der oben genannten Serviceleistungen anzunehmen oder die komplette Infrastruktur für den Softwareentwicklungsprozess in der Firma aufzubauen. Letzteres hat die Konsequenz, dass die Firma Ressourcen benötigt, den Betrieb am Laufen zu halten.

Der Softwareentwicklungsprozess, beschrieben in Abschnitt 5.5.1, benötigt für die notwendigen Werkzeuge diese Infrastruktur. Im Folgenden soll es keine Rolle spielen, ob eine eigene Implementierung oder ein *IAAS* bzw. *PAAS* eingesetzt wird. Ein Softwareentwicklungsprozess, realisiert durch ein *SAAS*, wird hier nicht betrachtet und ebenso kein System zur Authentifizierung von Benutzern (z. B. *Lightweight Directory Access Protocol*, kurz *LDAP*).

Die Zuordnung der Werkzeuge zu physikalischen oder logischen Einheiten wird in der Fachsprache oftmals Bauplan genannt.

Abbildung 5.21 zeigt auf eine karikative Art einen Bauplan. In den Ausführungen aus Abschnitt 5.5.1 soll gezeigt werden, dass das Ticketmanagementsystem von zentraler Bedeutung ist.

Die Kommunikation zwischen Entwicklern, Testteam, Softwareintegrationsteam und Deployment-Team muss über *Tickets* erfolgen, damit eine Verfolgbarkeit der anstehenden Probleme gewährleistet ist. Tickets werden immer mit Identifikationsnummern versehen, die eine Referenz auf bestimmte Probleme möglich machen.

Des Weiteren sind Tickets bestens geeignet für die Dokumentation und den Nachweis der Verifikation von Anforderungen, Tests, Entwicklung und Arbeitspaketen des Projektmanagers. Zusammengefasst lässt sich sagen, dass ohne Ticketmanagementsystem keine Softwareentwicklung stattfinden sollte.

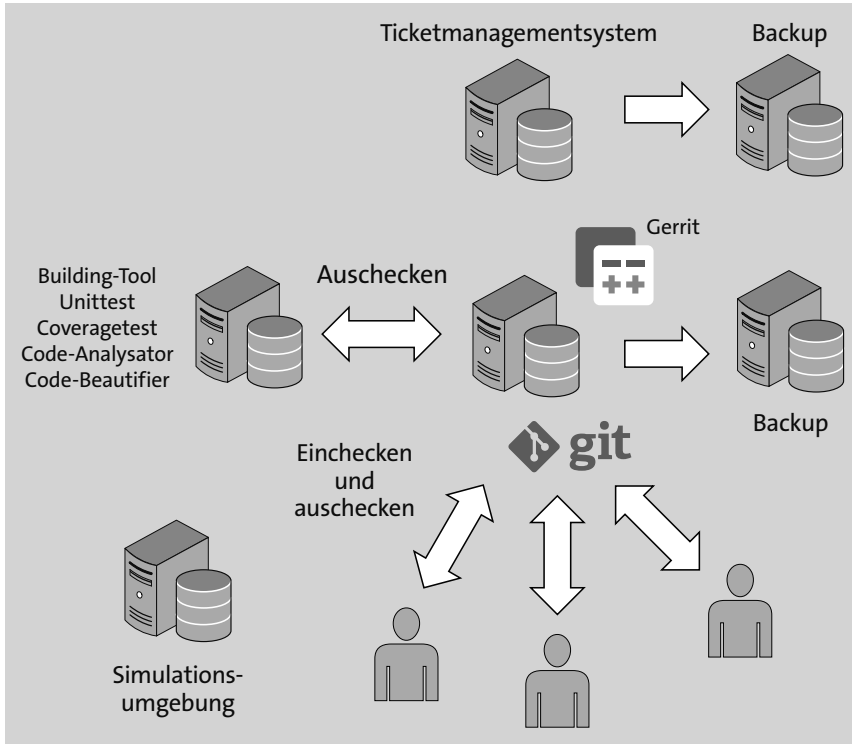


Abbildung 5.21 Bauplan der Entwicklungswerkzeuge

In Abbildung 5.21 wird das Ticketmanagementsystem ganz nach oben gesetzt. Jeder Entwickler, Tester, Projektmanager, Manager und eventuell auch Kunde sollte projektbezogenen Zugriff haben. Da die Dokumentation und Verfolgbarkeit der Tickets derart wichtig ist, darf kein Datenverlust auftreten. Deshalb muss ein funktionierendes Backup-System eingerichtet werden. Das heißt, dass die gespeicherten Tickets auf einem Zweitsystem gesichert sein sollten. Damit die Funktion der Wiederherstellung einwandfrei klappt, sollte diese in regelmäßigen Abständen getestet werden. Unter dem Ticketmanagementsystem in der Mitte der Abbildung ist das Repository (in diesem Fall *GIT*) auf einem weiteren Server eingerichtet. Da auch das Repository eine zentrale Komponente ist, sollte sein Inhalt in regelmäßigen Abständen (z. B. täglich) auf ein Backup-System zwischengespeichert werden. Auch hier gilt wie beim Ticketmanagementsystem: Das Wiederherstellen muss geübt sein.

Jeder einzelne Entwickler ist so in der Lage, einen Softwarestand abhängig von der Version vom Repository herunterzuladen (*auszuchecken*). Auf den individuellen Rechnern kann der Entwickler den Quellcode beliebig erweitern, verbessern oder fixen. Um den aktualisierten Softwarestand wieder zurück auf das Repository einzuchecken, benötigt der Entwickler ein Ticket mit Identifikationsnummer, das er von sich selbst oder von einer anderen Person (Entwickler, Tester, Projektmanager etc.) zugewiesen bekommt. So kommt der Softwarestand des Entwicklers in ein vorläufiges Release. Für den Review-Prozess wird ein Werkzeug benötigt, das Entwicklern die Möglichkeit gibt, Quellcode aus dem vorläufigen Release anzuschauen und diesen zu kommentieren. Dabei erlaubt das Werkzeug dem Reviewer, das Einchecken des Quellcode zu blockieren, wenn er der Auffassung ist, dass dieser wegen Auffälligkeiten nicht in das endgültige Release abgelegt werden soll. Ein Vertreter diese Werkzeugs ist *Gerrit*. Es gibt weitere Ticketmanagementsysteme, die bereits Review-Werkzeuge integriert haben. Das Softwareintegrationsteam sammelt die vorläufigen Releases, freigegeben durch das Review-Team, und baut daraus einen neuen Softwarestand. Dieser Schritt kann unter Umständen auch automatisch erfolgen.

Ein Werkzeug, das diese Funktion erfüllt, ist *Jenkins* (in Abbildung 5.21 links, Mitte). Das Werkzeug kann nach entsprechender Konfiguration automatisch einen Softwarestand auschecken, bauen, testen und analysieren. Tritt bei diesem Prozess ein Fehler auf, wird das Softwareintegrationsteam benachrichtigt, und nach entsprechender Untersuchung wird ein Ticket an den Verursacher geschrieben. Um den Softwarebauprozess nicht aufzuhalten, kann das vorläufige Release des Verursachers aus dem Bauprozess herausgenommen werden.

Die Implementierung von Simulationen kann sehr umfangreich sein, sie sind oftmals auf eigenem Server installiert (in Abbildung 5.21 unten links). Ein Prozessmanagement auf diesem Server kann für den Entwickler einen eigenen Simulationsprozess bei Bedarf starten. Schnittstellen erlauben es, die Software des Entwicklers gegen die Simulation zu testen.

Auch der Bauprozess kann die Simulation nach dem Bau von Software nutzen. Beim Auftreten eines Fehlers kann der Bauprozess abgebrochen werden, damit das Softwareintegrationsteam die Fehlerursache untersuchen kann.

Abschließend soll noch erwähnt werden, dass Installation und Betrieb der Infrastruktur für den Softwareentwicklungsprozess einen großen Aufwand bedeuten. Der Softwareentwicklungsprozess ist häufiger auch Veränderungen ausgesetzt. Neue Werkzeuge tauchen auf und müssen in den Prozess integriert werden. Der Prozess selbst wird immer wieder angepasst, und das Entwicklungsteam muss darauf geschult werden. Bei größeren Entwicklungsmannschaften gibt es wegen des Entwicklungsaufwands für diese Aufgaben freigestellte Teams.

5.5.3 Bezug zum Fallbeispiel

Welchen Softwareentwicklungsprozess die Software des *FMS* aus Abschnitt 5.1 hatte, ist mir nicht bekannt. Allerdings ist der beschriebene Prozess zum Zeitpunkt der Entstehung des Buchs allgemeine Praxis in vielen Firmen. So kann angenommen werden, dass der Softwareentwicklungsprozess nicht die Vernetzung und Automatisierung hatte, die oben beschrieben wurde. Die Kollision des Flugzeugs konnte auf eine fehlerhafte Eingabe und eine inkorrekte Textvervollständigung zurückgeführt werden. Möglicherweise hätten vollständige Unit-Tests an Schnittstellen oder Tests mit Simulationen den Fehler entdeckt. Dann hätte dieser früh in der Entwicklungsphase erkannt und behoben werden können. Natürlich ist es aus heutiger Sicht schwierig, zu behaupten, dass wirklich fatale Fehler gefunden worden wären. Aber es sollte Ihnen jetzt bewusst sein, dass die Kontrolle durch *Unit-Tests*, *Reviews*, *Simulationen* etc. engmaschig geworden ist, wenn ein stringenter Softwareentwicklungsprozess eingehalten wird.

5.6 Abschließende Bemerkungen

Im Fallbeispiel aus Abschnitt 5.1 wurde ein Unglück beschrieben, das durch Software hervorgerufen wurde. Hierbei hat das Autopilotsystem Eingaben akzeptiert, die nicht vom Piloten beabsichtigt waren. Dadurch kam es zu einer ungewollten Kurskorrektur des Flugzeugs. Eine Kontrolle der Eingabe erfolgte nicht durch den Piloten, da er wohl großes Vertrauen in das System hatte. Der Fehler kann auf eine fehlende Rückfrage der Software des Autopilotsystems zurückgeführt werden. Deshalb sind die darauffolgenden Streitigkeiten zwischen den Flugbehörden und dem Hersteller des Autopilotsystems nachvollziehbar. Die tatsächliche Ursache sollte aber vordatiert werden, und zwar in die Zeit der Erstellung von Anforderungsdokumenten und deren Validierung. Es traten nämlich bei der Entwicklung des Autopilotsystems mindestens zwei Parteien auf, die aus unterschiedlichen Bereichen kamen.

Auf der einen Seite gab es den Experten für Autopilotsysteme, der nicht unbedingt technisches Verständnis in der Softwaretechnik hatte. Und auf der anderen Seite gab es den Softwareexperten, die begrenztes Wissen über Autopilotsysteme hatte. Dieser war durchaus fähig, sich in die Thematik einzuarbeiten, aber hatte er den Wissensstand des Experten niemals umfänglich erreicht. Es ist deshalb nachvollziehbar, dass ein Anforderungsdokument zu Missverständnissen führen kann. Die Kommunikation (das Anforderungsdokument ist eine Art von Kommunikation) zwischen dem Ersteller der Anforderungsdokumente und dem Entwickler gehört deswegen zu den größten Fehlerquellen. Abbildung 5.22 zeigt diese Interaktion zwischen den beiden Experten.

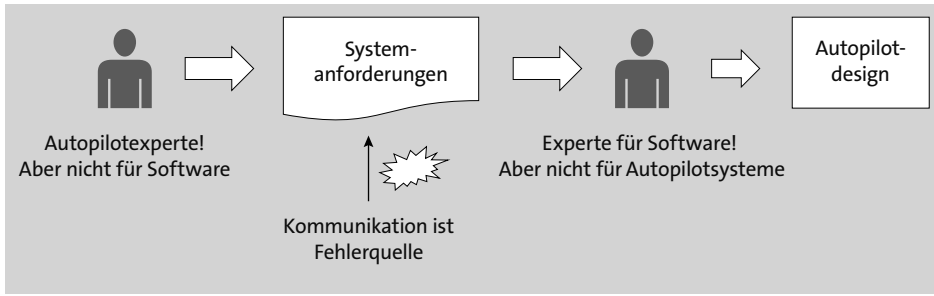


Abbildung 5.22 Beispiel für Kommunikationsfehler

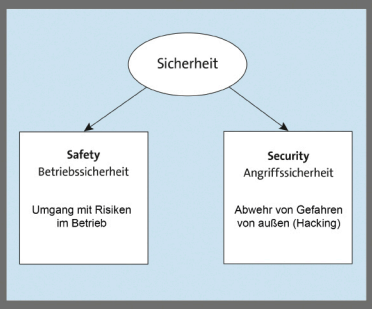
Es ist deswegen umso wichtiger, dass sich eine dritte Partei der Validierung des Anforderungsdokuments annimmt. Es sollte ein Team sein, dass nicht komplett von der einen oder anderen Seite stammt. Daraus folgt sogar, dass eine Person zur Validierung nicht notwendigerweise aus dem Softwarebereich kommen muss, wobei eine Affinität zur Software bestehen sollte. Es sollte genug Verständnis vorhanden sein, um ein Anforderungsdokument zu verstehen, und genug technisches Verständnis, um die Softwaretests durchführen zu können.

Safety Engineering

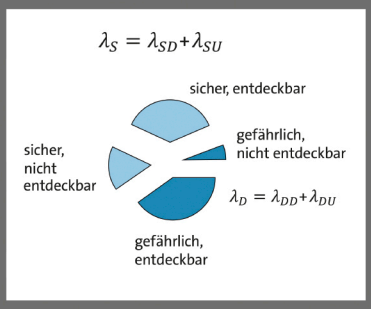
»Safety doesn't happen by accident«

Sichere und zuverlässige Systeme erstellen

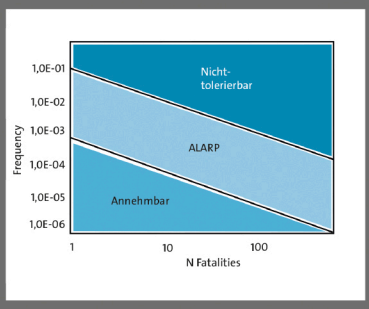
Dieses Lehrbuch vermittelt Ihnen die Grundlagen sicherer Softwareentwicklung und die Prinzipien der Betriebssicherheit in der Hardwareentwicklung. Sie lernen, wie Sie die Risiken komplexer Systeme einschätzen, Fehlerbaumanalysen durchführen, Risikographen gestalten und die essenziellen Methoden der sicheren Systementwicklung beherrschen.



Safety und Security



Fehler erkennen und verstehen



Modelle und Simulationen

Gute Programmierung

Unit-Tests, Code Reviews, defensive Programmierung: Schon mit einfachen Schritten können Sie die Qualität und Sicherheit Ihrer Software spürbar erhöhen. Hier erfahren Sie, was wirklich einen Mehrwert bietet und worauf Sie achten müssen.

Für Studium und Beruf

Von der Risikoidentifikation bis hin zu fortgeschrittenen Themen wie Fehlerbaum- und Markov-Analysen erhalten Sie einen Überblick über die Techniken der funktionalen Sicherheit. Fallbeispiele erläutern Sicherheitsprobleme, damit Sie aus den Fehlern beim Design und der Umsetzung sicherheitskritischer Systeme lernen können.

Sicher, robust und zuverlässig

Je komplexer Systeme werden, desto anfälliger sind sie für Ausfälle und Fehler. Dieses Lehrbuch zeigt Ihnen, mit welchen Methoden Sie systemrelevante Risiken qualitativ und quantitativ abschätzen können, um fehlerarme und wartbare Systeme zu entwickeln.

Aus dem Inhalt

- Normen und Sicherheitsrichtlinien
- Sicherheit in der Software- und Hardwareentwicklung
- Fehler analysieren und verstehen
- Verfügbarkeit, Schaden, Risiko, Fehlertoleranz
- Kenngrößen: Zuverlässigkeit, Ausfallrate, Lebensdauer
- Gefahrenanalyse
- Fehlerbaumanalyse
- Risikograph
- Layer of Protection Analysis
- Zuverlässigkeitsblockdiagramm
- Markov-Decision-Prozess
- Binary-Decision-Diagrams

Prof. Dr. Derk Rembold leitet das Institut für Echtzeitsysteme und Softwaretechnik an der Hochschule Albstadt-Sigmaringen. Er beschäftigt sich mit Betriebssicherheit, der qualitativen Überwachung von Produkten und Software in automatisierten Prozessen.

Rheinwerk Computing

Gedruckt in Deutschland
Papier aus nachhaltiger Waldwirtschaft
Mineralölfreie Druckfarben

ISBN 978-3-8362-XXXX-X

€ 59,90 [D] € 61,60 [A]
Informatik
ISBN 978-3-8362-9928-2