Hardware-Obsoleszenz mit Architektur lösen

Lösungskonzepte für den rückwärtskompatiblen Austausch von Prozessor und Betriebssystem

Dr. Jörg-Volker Müller, Systemum GmbH & Co. KG
Roman Koch, GMC Instruments GmbH

Der Wert von „historisch gewachsener“ Embedded Software ist nicht zu unterschätzen. Eine vollständige Neuentwicklung ist in vielen Fällen wirtschaftlich und zeitlich nicht realisierbar. Allerdings erfordern abgekündigte Hardware und gestiegene Anforderungen oft den Wechsel von Prozessor und Betriebssystem. Damit einher gehen eine Portierung und Weiterentwicklung, sprich Modernisierung. Dennoch soll mit der modernisierten Software oft die bestehende Hardwarebasis weiter unterstützt werden. Eine Anforderung, die nur mit fundierten Architekturkonzepten und einem guten Variantenmanagement lösbar ist.

Die rasante Entwicklung der Technologie verändert ständig die Welt, in der wir leben. Doch während immer neue Produkte auf den Markt kommen, gibt es im Kern vieler erfolgreicher Produkte Softwarebestandteile, die seit Jahrzehnten existieren und ständig weiterentwickelt werden. Diese „historisch gewachsenen“ Softwarekomponenten bergen einen unschätzbaren Wert, der die Basis für aktuelle und zukünftige Entwicklungen bilden sollte.

Modernisierung von Software im Produkt

Aber was passiert, wenn die Welt um sie herum sich weiterentwickelt und die Software ihr nicht mehr gerecht wird? Die Modernisierung von Embedded Software im Produktgeschäft ist in diesen Fällen oft eine Notwendigkeit. Die Modernisierung von Software ist somit nicht nur eine strategische Entscheidung, sondern oft eine notwendige Reaktion auf verschiedenste interne und externe Faktoren (Abbildung 1).

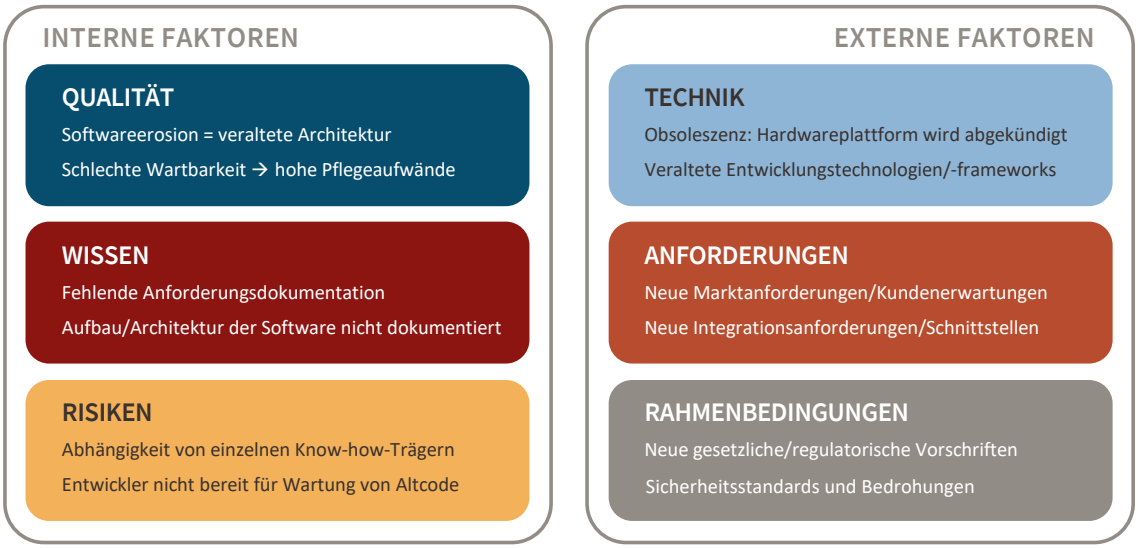

Abbildung 1 Gründe für eine Modernisierung

Es ist über den gesamten Produktlebenszyklus essenziell, sowohl die interne Softwarelandschaft als auch die externen Einflüsse regelmäßig zu bewerten, um zeitnah angemessene Modernisierungsentscheidungen treffen zu können.

Es stellt sich die entscheidende Frage: Wie können Unternehmen die Vorteile der vorhandenen, bewährten Software erhalten und gleichzeitig die Software so modernisieren, dass sie den aktuellen Anforderungen gerecht wird?

Modernisierungsstrategien

Eine Softwaremodernisierung von Embedded Software ist kein einfaches Unterfangen und die Wahl der richtigen Strategie hängt stark von den individuellen Anforderungen und Gegebenheiten des jeweiligen Produkts und seinem Produktumfeld ab. Zwei gegensätzliche Ansätze sind das **Reengineering** und die komplette **Neuentwicklung**. Beide haben ihre Vor- und Nachteile:

Bei einem **Reengineering** handelt es sich um eine schrittweise Verbesserung des bestehenden Systems. Dieser Ansatz ermöglicht in der Regel schnelle Erfolge, da durch gezielte lokale Refactorings oder größere Reengineerings unmittelbare Resultate erzielt werden können. Ein schrittweises Vorgehen hat zudem den Vorteil, dass nicht das gesamte System auf einmal überarbeitet werden muss, und bereits existierende Spezifikationen, Tests und andere Ressourcen weiterhin genutzt werden können.

Jedoch birgt diese Methode auch Risiken. So kann es passieren, dass schlechte Strukturen nicht grundlegend überarbeitet oder überflüssige Funktionen nicht beseitigt werden. Es besteht auch die Gefahr, dass man zu sehr an bestehenden Strukturen festhält, was die Implementierung von zukünftigen Anforderungen erschweren kann. Darüber hinaus gerät der Modernisierungsprozess oft ohne eine klare Zielsetzung und formulierte Strategie ins Stocken – die Modernisierung verläuft im Sande und bleibt wirkungslos.

Bei einer **Neuentwicklung** werden in der Regel alle bisher getroffenen Entscheidungen – von den fachlichen Funktionen und Features über die Architektur bis hin zur genutzten Toolwelt – kritisch hinterfragt. Diese Methode eröffnet die Möglichkeit, eine Architektur zu entwerfen, die auf Zukunftsanforderungen ausgelegt ist und sich gleichzeitig von veralteten Strukturen und Funktionen löst.

Doch mit solch einem Neubeginn gehen auch erhebliche Herausforderungen einher: Wenn fachspezifisches Wissen in Form von Anforderungsdokumentation fehlt, werden umfassende und kostenintensive Analysen der bestehenden Funktionalitäten notwendig. Oftmals ist eine "Softwarearchäologie" erforderlich, um Algorithmen zu entschlüsseln und für die Neukonzeption nutzbar zu machen. Dabei lauert die "100%-Falle", die den Ehrgeiz weckt, sämtliche Features des Originalsystems in die Neuentwicklung übernehmen zu wollen. Dies kann den Erneuerungsprozess erheblich verlängern, sodass die Ablösung des Altprodukts immer wieder aufgeschoben wird. Je länger die Modernisierung dauert, umso größer ist die Parallelentwicklung, weil Weiterentwicklungen des bestehenden (Alt-)Produkts auch im neuen integriert werden müssen.

Die Kunst besteht darin, zwischen den beiden Extremen – dem reinen Reengineering und der vollständigen Neuentwicklung – die ideale **Modernisierungsstrategie** zu finden. Hierbei sind zahlreiche Faktoren zu berücksichtigen: Angefangen beim vorhandenen Wissen über fachliche Funktionen, über die Qualität der bestehenden Softwareelemente, bis hin zu den verfügbaren Ressourcen, müssen diverse Einflussgrößen analysiert werden. Basierend auf diesen Kriterien wird sorgsam

abgewogen, welche Bestandteile der Altsoftware erhalten, modifiziert oder ganz neu entwickelt werden sollen.

Ein entscheidender Aspekt einer solchen Modernisierungsstrategie ist die **Zielarchitektur** des modernisierten Systems und der Pfad von der bestehenden Softwarearchitektur zu dieser neuen Struktur. Dabei erweist sich ein iterativ-inkrementeller Ansatz als vorteilhaft, da nach jedem Modernisierungsschritt eine Überprüfung und gegebenenfalls Anpassung des gewählten Vorgehens erfolgt (vgl. Abbildung 2).

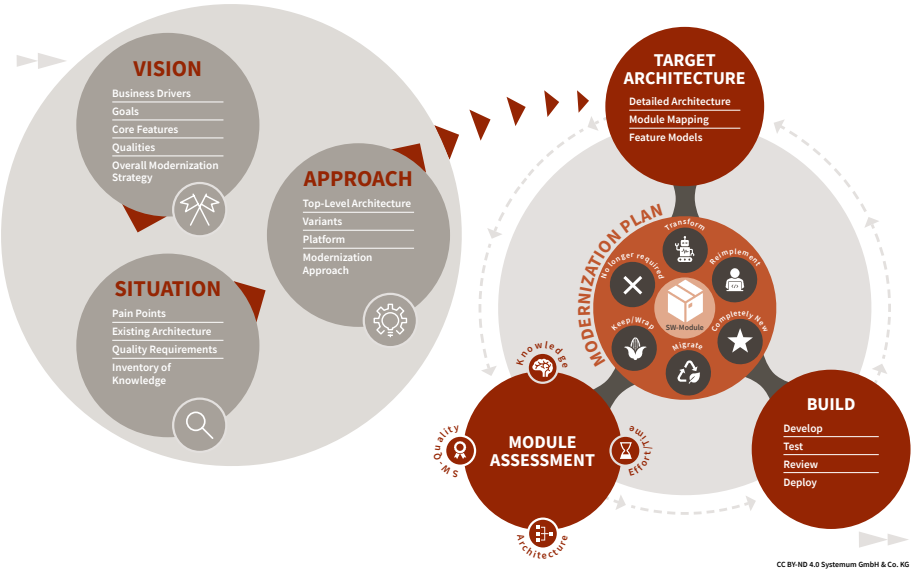

Abbildung 2 Systemum-Modernisierungsvorgehen

Modernisierung Profitest

Im Folgenden wird an einem realen Modernisierungsprojekt gezeigt, wie im Rahmen einer kosteneffizienten Teilmodernisierung einer Embedded Software im Bereich Messgeräte die bestehende Software zukunftsfähig gemacht wurde und gleichzeitig die Kompatibilität zu bestehenden Geräten im Feld gesichert wurden.

Betrachtet hier wird *ProfiTest*, eines der Prüfgeräte einer Gerätefamilie, das zur Prüfung elektrischer Installationen und Anlagen dient (s. [4], [5]).

Die Geräte sind langlebig und können 15 bis 20 Jahre im Einsatz sein. In der aktuellen Phase seines Lebenszyklus ist das Produkt seit mehreren Jahren auf dem Markt und erhält in der Regel weniger als fünf jährliche Software-Veröffentlichungen pro Gerät.

Das Produkt besteht konstruktiv aus einem beweglichen Bedienteil und einem Basisteil (Abbildung 3). Das Bedienteil ist verantwortlich für die Benutzerführung, Abläufe, Datentransfer und Datenspeicherung. Das Basisteil (liegt hier nicht im Fokus) handhabt die gerätespezifische Messfunktionalität.

Das Bedienteil ist verantwortlich für die Benutzerführung, Abläufe, Datentransfer und Datenspeicherung. Das Basisteil (liegt hier nicht im Fokus) handhabt die gerätespezifische Messfunktionalität.

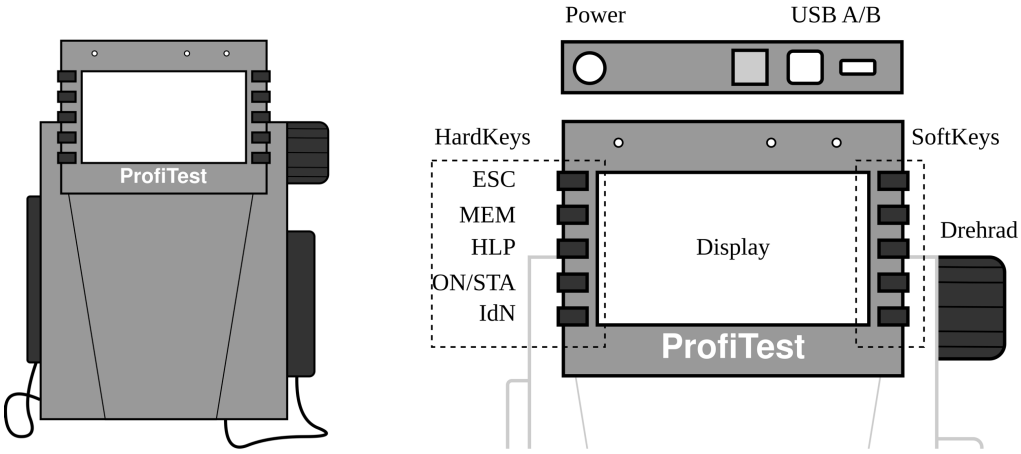

Abbildung 3 ProfiTest (links) / Bedienteil und Bedienelemente (rechts).

Randbedingungen

Neben der Erfüllung der technischen Anforderungen wurden für die modernisierten Messgeräte die folgenden Rahmenbedingungen festgelegt:

A) Die kontinuierliche Lieferfähigkeit und fortwährende Kompatibilität mit dem bestehenden Gerät sind Voraussetzungen für eine Modernisierung bzw. Weiterentwicklung von Software und Hardware.

B) Die vollständige Funktionalität der aktuellen Geräte, einschließlich der tatsächlichen Messfunktionen, muss beibehalten werden.

C) Die modernisierte Anwendung soll auf verschiedenen Hardware-Plattformen lauffähig sein, einschließlich der Legacy-Hardware (gealterte Technologie) und einer Linux-Distribution.

D) Die Grafikausgabe muss plattformübergreifend sein und sich an die Anzeige anpassen, einschließlich Auflösung und Farbtiefe.

E) Die aktuellen Eingabemethoden müssen aufgrund ihrer hohen Bekanntheit und Erkennbarkeit weiterhin unterstützt werden.

F) Die Benutzerführung und das grundlegende, ausgereifte Bildschirmlayout sollen beibehalten werden, um einen schnellen Einstieg zu ermöglichen.

Modernisierungsstrategie

Abhängig von den Entwicklungsstadien des Produkts und den neuen Anforderungen sowie Randbedingungen galt es nun, die passende Modernisierungsstrategie zu wählen. Es standen drei Optionen zur Software-Anpassung zur Auswahl, von denen jede verschiedene Risiken in sich barg.

- **Neuentwicklung:** Die offensichtlichste Option, insbesondere für ambitionierte Berufseinsteiger, war das vollständige Neuschreiben des Codes.
- **Weiterentwicklung Seitenzweig:** Eine alternative Option war die Anpassung eines verwandten, aber parallel entwickelten Code-Zweigs, der etwas jünger ist und bereits bekannte Anforderungen erfüllt.
- **Teilmodernisierung:** Die dritte Option bestand darin, den vorhandenen Code zu bewerten und eine Teilmodernisierung auf der Basis des vorhandenen Systems durchzuführen.

Die ersten beiden Optionen waren in Bezug auf Risiken ähnlich problematisch, da sie voraussichtlich sehr ressourcenintensiv wären und die gegebenen Randbedingungen nicht erfüllen würden.

Nach einer gründlichen Analyse der Risiken und Chancen wurde die Entscheidung getroffen, eine schrittweise Modernisierung des Legacy-Systems zur nächsten Entwicklungsstufe hin durchzuführen (Teilmodernisierung).

Die Kompetenz des Teams spielte hierbei entscheidende Rolle: Erfahrene Entwickler sind von Vorteil für den Erfolg einer Modernisierung.

Die Modernisierung wurde begünstigt, da die internen Entwicklungsprozesse bereits auf die Handhabung von Produktvarianten, sowohl in Bezug auf die Software als auch die Hardware, ausgerichtet waren.

Schnittstellen

Die umfassenden Schnittstellen des Geräts (Abbildung 3 und Verweise [4] und [5]) blieben weitgehend unverändert.

Allerdings erfolgte eine Aktualisierung durch den Austausch des monochromen Displays gegen eine farbige, hochauflösende Alternative. Die RS232-Buchse wurde durch eine USB-Host-Schnittstelle ersetzt. Die modernisierten Geräte erhielten zudem erweiterte Vernetzungsmöglichkeiten.

Hardware

Die Legacy-Hardware umfasste eine selbst entwickelte Hauptplatine mit einem SAM7-Prozessor, einem RAM von wenigen Megabytes und verschiedenen Steuerungselementen. Mehrere Module waren an den seriellen Schnittstellen des Prozessors angeschlossen.

Die alte Hardware hatte ihre Leistungsgrenze erreicht, und das Potenzial für Softwaremodernisierungen auf diesem Hardwarestand war erschöpft. In der nächsten Gerätegeneration wurde ein Prozessormodul mit ausreichender Leistung für ein Linux-Betriebssystem eingeführt.

Das veraltete monochrome Display mit geringer Auflösung und kritischer Verfügbarkeit wurde durch ein modernes Farbdisplay mit hoher Auflösung ersetzt.

Die Erweiterung der Konnektivität mit USB-Host, Bluetooth, WiFi/Ethernet war auf der Legacy-Hardware mit vertretbarem Aufwand nicht umsetzbar. Die neudesignte Hauptplatine mit Schnittstellen zur Integration eines System-on-Modules (SoM) und einer erneuerten Stromversorgung bietet zusätzliche Schnittstellen für Datenkommunikation und Vernetzung.

Das neue Design ermöglicht iterative Verbesserungen und Weiterentwicklungen der Hauptplatine mit vorhersehbaren Auswirkungen auf die Anwendung.

Software

Die Anwendungssoftware wurde vor über 30 Jahren in Assembler entwickelt und später in den Programmiersprachen C und teilweise C++ weitergeschrieben. Im Laufe der Jahre wurden mehrere funktionale Erweiterungen und teilweise Modernisierungen durchgeführt, sodass die Anwendungssoftware heute etwa 100.000 Zeilen Code umfasst und sehr ausgereift ist. Trotz ihres Alters besitzt die Software noch genug Potenzial für Modernisierungen.

Die Softwareentwicklung wurde von den Anforderungen der Fachdomäne beeinflusst. Die hierarchisch strukturierten Softwaremodule wurden entwickelt, um spezifische Aufgaben innerhalb der Fachdomäne zu erfüllen, wobei die Anforderungen direkt in den Code umgesetzt wurden. Die interne Struktur der Bedienteilsoftware wurde stark von den Anforderungen der Benutzerführung und den Benutzerinteraktionen geprägt.

Die klar erkennbare Funktionalität und Fachlogik erleichterten den Einstieg in den Code. Allerdings führte die geringe Abstraktion auch zu großen Dateien mit problematischen Abschnitten im Code und direkten Hardware-Zugriffen in der Anwendung. Eine geringe Testabdeckung und eine gewachsene Softwarearchitektur stellten ebenfalls eine Herausforderung für die anstehende Modernisierung dar.

Die Abläufe waren in Tasks organisiert und wurden von einem "Hamsterrad-Scheduler" gesteuert, der in einer festen Abfolge in der *main*-Funktion die Tasks aufrief. Diese Tasks, zusammen mit der *while*-Schleife, bildeten eine Art "State-Machine-Schlinge", die die Software nach mehreren Durchläufen des Schedulers in den gewünschten Zustand brachte.

Der Übergang zu Embedded Linux war eine strategische Entscheidung mit weitreichenden Konsequenzen. Da Embedded Linux bis dato nicht zu den Kernkompetenzen gehörte, erforderte dies externe Unterstützung, um das erforderliche Know-how aufzubauen. Daher war die Beauftragung eines externen Dienstleisters zur Einrichtung und Pflege einer Embedded Linux-Distribution ein sinnvoller Schritt.

Abbildung 4 illustriert den Übergang von der Legacy- zur Linux-Plattform. Die hauptsächlichen Anpassungen betrafen die Einführung einer Abstraktionsebene für den Zugriff auf die Display-Hardware.

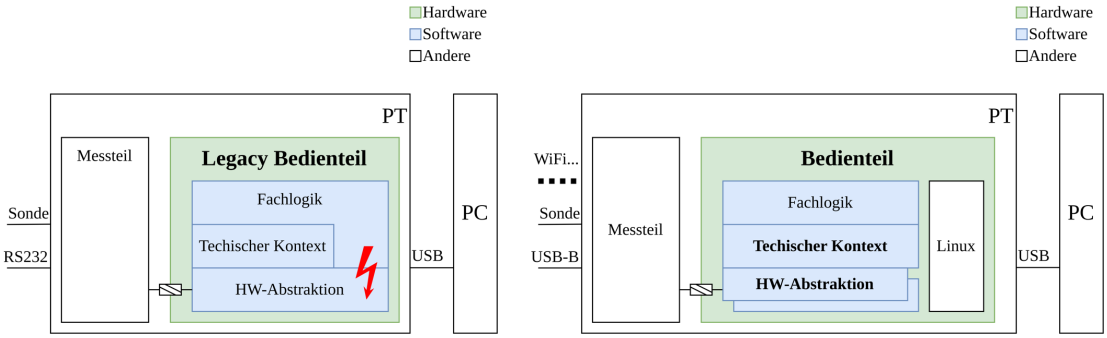

Abbildung 4 Bedienteil Legacy (links) und modernisiert (rechts).

Die farblosen Bereiche wurden nicht modernisiert und werden daher nicht weiter betrachtet. Die grünen Bereiche links und rechts entsprechen der Legacy- und modernisierten Hardware. Die blauen Bereiche links und rechts repräsentieren die Legacy- und die modernisierte Anwendungssoftware.

In beiden Fällen wurde die Software in Schichten modelliert. Die Unterschiede zwischen den Modellen illustrieren die Softwareanpassungen.

Migration

Die Einführung eines toolbasierten Coding-Regelwerks half, ein gemeinsames Verständnis für die Code-Hygiene zu entwickeln und sicherzustellen, dass diese eingehalten wird.

Der vorhandene Code wurde mithilfe eines Analyse-Tools formell überprüft, wodurch Schwachstellen (Vulnerabilities) identifiziert und Hinweise auf technische Schulden im Code gefunden werden konnten. Bei der Planung der erforderlichen Refactorings, wie beispielsweise der Beseitigung von “Magic Numbers”, wurde ein pragmatischer Ansatz verfolgt. Dabei lag der Fokus auf minimalinvasiven strukturellen Änderungen, die das Systemverhalten weitgehend unverändert ließen.

Die monolithisch wirkende Software wurde durch die Anwendung mehrerer Modelle aufgebrochen und strukturiert. Dabei wurde das kaum erkennbare MVC-Muster (Model-View-Controller) gestärkt, um die Fachlogik von der Steuerung und Darstellung zu trennen. Zudem wurden in einem Schichtenmodell zusätzliche Schichten in der Software modelliert, um die notwendigen Abstraktionen für die Migration zu beschreiben.

Die Vorgehensweise bei der Portierung der Softwarekomponenten im Bedienteil variierte von Modul zu Modul. Wie hoch die Abstraktion im Code eingesetzt werden sollte, hing von verschiedenen Faktoren ab. Manchmal wurden in der Praxis bewusst auch “schlechte” Entscheidungen getroffen. Letztendlich war nicht das Modell entscheidend, sondern der erforderliche tatsächliche Aufwand für die Modernisierung.

In einigen Fällen war es möglich, die Implementierung vollständig zu verwerfen und ausschließlich auf den Bestandscode zurückzugreifen (z. B. Firmware-Update Module).

Für einige Komponenten konnte die Abstraktionsebene sehr hoch angesetzt werden. Das Dateisystem ist ein Beispiel, bei dem eine aufwendige, proprietäre Legacy-Implementierung mithilfe eines Adapters durch Linux-Systemaufrufe ersetzt wurde.

In anderen Fällen waren aufwendigere, plattformspezifische Anpassungen erforderlich. Ein gutes Beispiel hierfür war das 10-Tasten-Keypad. Dieses Keypad, das spezielle Kundenkomfort- und Benutzerführungsfunktionen erforderte, musste auf aufwendig adaptierte Linux-Implementierungen aufgebaut werden.

In einigen wenigen Fällen mussten Hardwarezugriffe von einzelnen Pins abstrahiert werden, um die Anpassung zu ermöglichen.

Grafisches Benutzerinterface

Eine weitere Herausforderung war die stark in die Fachabläufe integrierte grafische Benutzeroberfläche. Diese Oberfläche wurde im Eigenbau erstellt und folgte einem “Slide-Show”-Muster, bei dem Bildschirmmasken aus gespeicherten Bitmaps übereinandergelegt wurden, um eine gültige Darstellung zu erzeugen. Die Verflechtung der Slide-Show-Maschine mit der Benutzerführung und der Fachlogik machte einen Austausch der Grafik-Engine in diesem Modernisierungsschritt aufwändig und riskant.

Die Lösung hatte auch positive Aspekte wie gut durchdachte Szenarien, klare Layouts, statische Maskendarstellungen und einen optimierten Ressourceneinsatz. Schließlich wurde entschieden, die Slide-Show-Maschine nach einer Displayabstraktion in ihrer aktuellen Form in den modernisierten Zustand zu übernehmen.

Während der Modernisierung wurden implizite Annahmen bezüglich der Positionen, Größe der Elemente, Auslösung und Farbtiefe aufgelöst. Die portierte Anwendung kann nun konfiguriert werden, um grafische Elemente auf verschiedenen Bildschirmen darzustellen.

Die Slide-Show-Engine verwendete insgesamt etwa 3000 manuell erstellte Bilder, darunter Masken, Maskenelemente und Hilfeseiten für alle Produktvarianten. Bei der

Modernisierung war es erforderlich, einen Satz von Bildern in einer geeigneten Größe für den neuen Bildschirm bereitzustellen.

In der ersten Iteration wurden alle notwendigen monochromen Legacy-Bilder, die für das Legacy-Produkt erforderlich waren, auf die Größe des neuen Bildschirms skaliert und der angepassten Engine zur Verfügung gestellt.

Parallel dazu wurden im Quellcode der Legacy-Geräte die direkten Verweise auf die grafischen Elemente, sowie ihre Positionen und Größen, durch indirekte Referenzen auf Datenstrukturen ersetzt. Alle “Magic Numbers” wurden identifiziert und durch Referenzen auf Konstanten ersetzt. Die Darstellungsfunktionen wurden um eine Skalierungsformel erweitert.

Mit den durchgeführten Änderungen konnten erste Erfahrungen mit der “neuen” GUI gesammelt werden. Es zeigte sich schnell, dass die Qualität der skalierten Grafiken für ein Produkt nicht akzeptabel war.

In der nächsten Entwicklungsiteration wurde ein Bildgenerator entwickelt. Alle grafischen Elemente, darunter Layouts, Masken, Maskenfragmente und Positionsboxen, wurden geräteunabhängig mit JSON formal beschrieben. Der Generator erzeugt zur Kompilierzeit displayspezifische Inhalte wie Bilder und Datenstrukturen mit Positionen, Größen und anderen Attributen basierend auf diesen Beschreibungen, skalierbaren Vektorgrafiken und Textfragmenten.

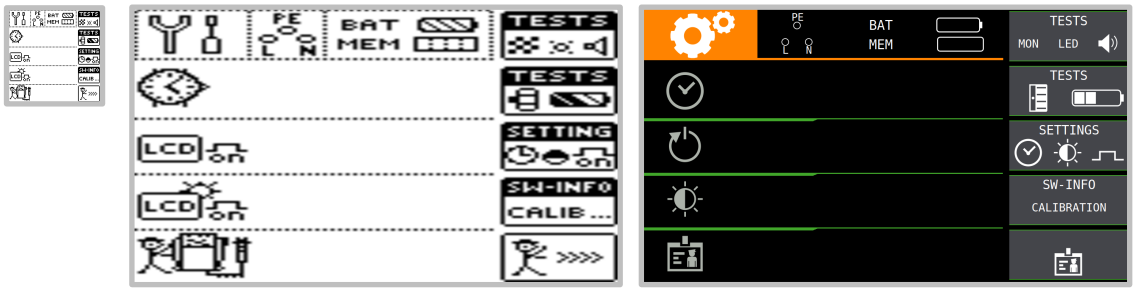

Abbildung 5 Grundmaske “Setup” in Originalgröße (links), skaliert und modern

Etwa 1400 Dateien mit Maskendefinitionen, Farbpaletten, Konfigurationen und skalierbaren Vektorgrafiken werden verwendet, um etwa 1090 Pixelbilder und rund 800 Code-Dateien mit Datenstrukturen zu generieren. Die Generator-Logs dienen zur Verifikation der Ergebnisse. Abbildung 6 zeigt schematisch die Generator-Pipeline.

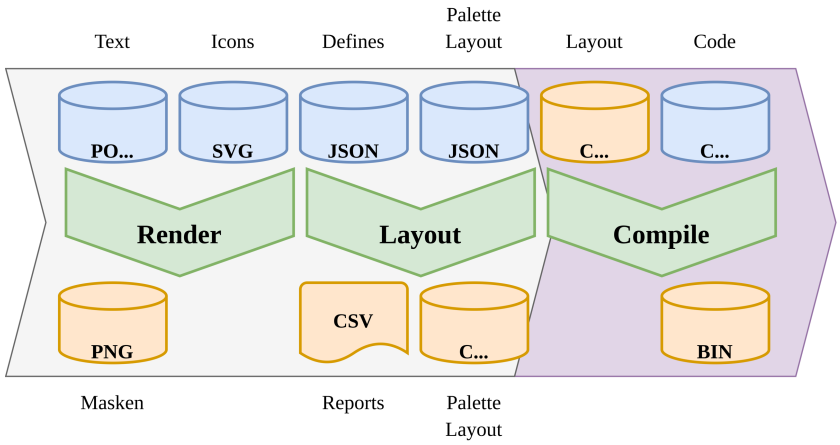

Abbildung 6 Generator-Pipeline.

Regression

Bei einer Software-Migration sollte die Regression vermieden werden, das bedeutet, dass unabsichtlicher Verlust spezifizierter Verhaltensmuster aufgrund der Codeanpassungen vermieden werden sollte.

Legacy-Code zu ändern, ohne das Verhalten in den Tests festzuhalten (die “change und pray”-Methode) ist riskant [2]. Durch einen Mangel an Ressourcen konnte die ohnehin begrenzte Testabdeckung nicht wesentlich erweitert werden. Das Risiko, dass die Codeanpassungen zu einer noch geringeren Wartbarkeit und zu noch mehr technischen Schulden führen können, wurde in Kauf genommen.

Die umfangreichen minimalinvasiven Änderungen zeichneten sich durch eine geringe Softwarekomplexität aus und konnten nahezu mechanisch durchgeführt werden. In komplexeren Fällen wurden vorhandene Tests oder das Debugging verwendet.

In vielen Fällen wurden die potenziellen Regressionen auf spätere Abnahmetests verschoben. Um Regressionen auf Produktebene schnell zu erkennen, wurde eine neue Test- und Bedienschnittstelle sowie eine Testsuite entwickelt und verwendet.

Migrationspfad

Der erste Schritt umfasste die Migration einer älteren, aber funktional vollständigen Lösung auf eine neue, leistungsfähigere Hardware. In den folgenden Iterationen wurden einige nicht mehr rückportierbare Erweiterungen hinzugefügt. Auf diese Weise war es möglich, sich schrittweise dem Ziel zu nähern. Basierend auf vorhandenen, gealterten Lösungen konnten neue Hardware, Software und Produkte entwickelt werden.

Abbildung 7 zeigt schematisch die wesentlichen Schritte bei der Migration der Legacy-Software auf Linux.

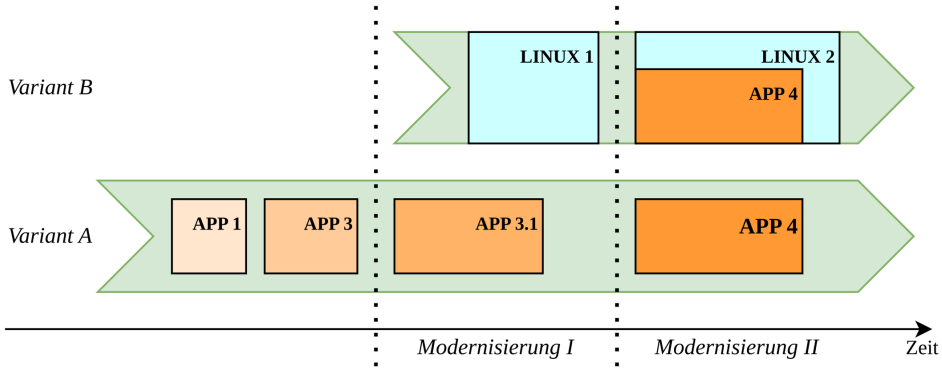

Abbildung 7 Softwaremigration

Die grünen Pfeile repräsentieren die Entwicklungslinien, während die farbigen Rechtecke die verschiedenen Softwarestände darstellen.

In der ersten Modernisierungsphase entlang der Linie "Variante A" wurde die Anwendungssoftware schrittweise modernisiert (Refactoring) und notwendige Abstraktionen für die Legacy-Hardware und -Komponenten eingeführt. Zusätzlich wurde eine Embedded-Linux-Distribution eingerichtet und die Anwendung auf eine nicht-produkteigene Plattform portiert (Tweak-Portierung), um frühzeitig mögliche Portierungsprobleme zu identifizieren.

In der zweiten Modernisierungsphase wurden Abstraktionen für die Linux-Plattform entwickelt und in ein Linux-Release integriert.

Abbildung 8 veranschaulicht die schrittweise Erweiterung der Anwendung, einschließlich der Abstraktion der hardware- und plattformabhängigen Komponenten sowie der Implementierung neuer Code-Fragmente für die neue Hardware und die Linux-Plattform.

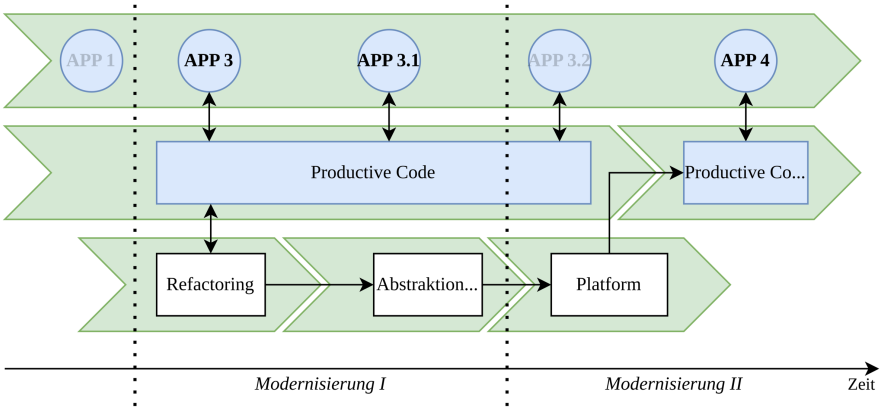

Abbildung 8 Erweiterung der Anwendung.

Die Hauptentwicklungslinie enthielt den produktiven Code der Anwendung. Mit dem Release *APP 3* begann die erste Migrationsphase mit dem Refactoring der Legacy-Software. Diese Phase endete mit der Einführung von Abstraktionen in den Legacy-Code.

Während minimalinvasive Änderungen und Erweiterungen am Bestandscode durchgeführt wurden, hinderte dies die Entwicklung nicht daran, weitere Features im Produktivcode zu implementieren und Software auszuliefern (*APP 3.2*).

Mit der Fertigstellung der Linux-Plattformkomponenten wurde die Softwaremigration abgeschlossen (*APP 4*). Bevor die implementierten Code-Teile in den Produktivcode integriert und veröffentlicht wurden, wurde sichergestellt, dass das Laufzeitverhalten der Software fehlerfrei war.

Timing

Bei der Portierung von langsamer auf schnellere Hardware kann die “innere Uhr” in der Anwendung schneller laufen, was zu Laufzeitfehlern führen kann. Im umgekehrten Fall kann eine sich selbst überlassene *while*-Schleife unnötige Systemlast verursachen.
Um sicherzustellen, dass die portierte Anwendung auf dem leistungsfähigeren System die gleiche Ausführungsgeschwindigkeit wie auf dem alten System beibehält, wurde der "Hamsterrad-Scheduler" gedrosselt.

Im Code konnten keine weiteren Stellen gefunden werden, die implizite Annahmen über die Ausführungsgeschwindigkeit der Anwendung gemacht haben.

Zeitaufwand

Die in Abbildung 9 aufgeführten relativen Zeitaufwände decken sowohl die Modernisierung der Hardware als auch der Software ab. Die Aufwände für Vorhaben wie Konstruktionsmaßnahmen, Fremdleistungen wie die Arbeit eines Designers oder Linux-Spezialisten und allgemeine Verwaltungskosten sind in diesen Zahlen nicht enthalten.

Das Vorhaben “Projekt” umfasst die gesamte für das Projekt aufgewendete Zeit, die keiner der anderen Gruppen zugeordnet werden kann.

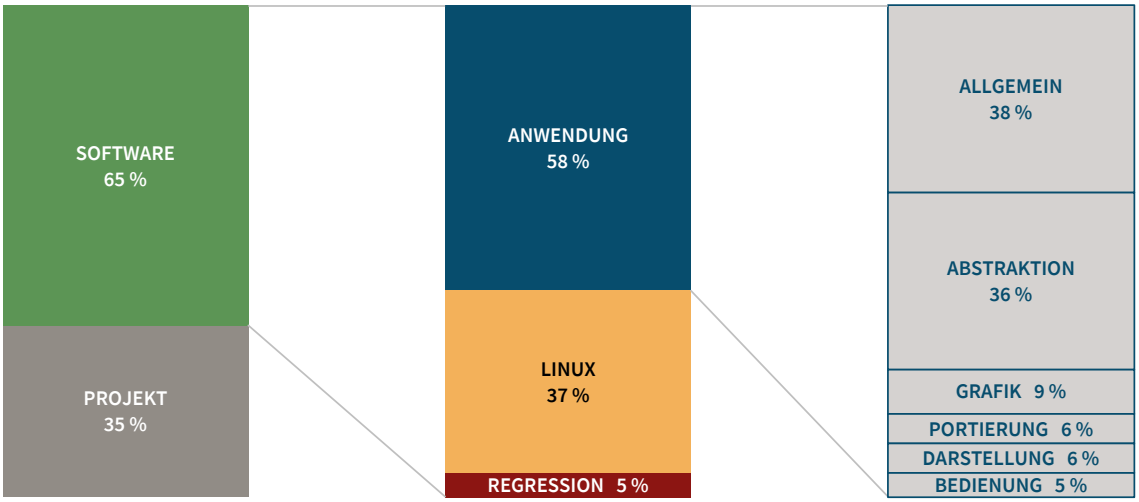

Abbildung 9 Verteilung der Aufwände für die Modernisierung

Die Aktivitäten im Zusammenhang mit dem Linux-Betriebssystem (“Linux”) umfassen das Einrichten eines Workflows, die Konfiguration und den Bau der Distribution sowie die Einrichtung des Entwicklungskits.

Im Vorhaben "Allgemein" sind die anwendungsbezogenen Aufgaben enthalten, darunter erforderliche Refactorings, die Entwicklung des Code- und Bild-Generators sowie die Erstellung der Maskenbeschreibungen.

Andere Vorhaben sind im Text ausführlich erläutert.

Zusammenfassung

In der Embedded-Softwareentwicklung befindet man sich selten auf der vielzitierten "grünen Wiese". Die Alterung von Hardware bis hin zur Obsoleszenz sowie das Leben mit geerbter Software sind übliche Herausforderungen.

Die Entscheidung für zwischen einer moderaten Teilmodernisierung und der vollständigen Neuentwicklung der nächsten Produktgeneration kann nicht immer frei gewählt werden. Randbedingungen und wirtschaftliche Faktoren schränken die Wahl der Modernisierungsstrategie in der Regel ein. Vor- und Nachteile, Chancen und Risiken müssen genauso bewertet werden wie die erwarteten Aufwände.

Die Teilmodernisierung, wie hier dargestellt, hat ihre Grenzen und ist nicht universell anwendbar. Der Umfang der Veränderungen sollte auf ausgewählte Aspekte und einen begrenzten Teil des Codes beschränkt sein. Hierbei spielen die Entstehungsgeschichte und die Qualität des bestehenden Codes eine entscheidende Rolle.

Der vorgestellte Modernisierungspfad basiert auf einer Fallstudie und ist nicht vollständig frei von einer gewissen Subjektivität. Da es nur wenige dokumentierte Fälle in der Literatur gibt, können die gezeigten Ideen und Ergebnisse nicht ohne Weiteres verallgemeinert werden.

In dem vorgestellten Projekt führte die gewählte Teilmodernisierungsstrategie zu einem risikoarmen und kosteneffizienten Wechsel der Hardware- und Betriebssystemplattform. Gleichzeitig wurde die Abwärtskompatibilität zur alten Hardware ohne wesentliche Abstriche bei der Funktionalität sichergestellt. Dies verdeutlicht den praktischen Nutzen einer wohlüberlegten Teilmodernisierung.

Verweise

[1]
Manuel Gerst & Claudia Eckert, John Clarkson, Udo Lindemann
Innovation in the tension of change and reuse
International Conference on Engineering Design, 2001

[2]
Michael C. Feathers
Effektives arbeiten mit Legacy Code
mitp, Frechen, 2011

[3]
Martin Fowler, Kent Beck
Refactoring – Improving the Design of Existing Code
Addison-Wesley, Boston, MA, 2000

Die untenstehenden URLs zeigen die Modernisierungsobjekte, einmal mit einer modernisierten Bedienteil-Software auf der Legacy-Hardware (ProfiTest MXTRA QI) und einmal mit modernisierter Software auf der neuen Hardware (ProfiTest MF XTRA):

[4]

GMC Instruments GmbH, ProfiTest MXTRA QI, 2023
https://www.gossenmetrawatt.de/produkte/mess-und-prueftechnik/prueftechnik/prueftechnik-fuer-e-mobility/pruefgeraete-fuer-e-ladekabel-mit-6-ma/profitest-mxtra-iq/

[5]
GMC Instruments GmbH, ProfiTest MF XTRA, 2023
https://www.gossenmetrawatt.de/produkte/mess-und-prueftechnik/prueftechnik/pruefung-elektrischer-installationen-und-anlagen/pruefungen-nach-din-vde-0100-600-0105-100-iec-60364-en-50110/profitest-mf-serie/profitest-mf-xtra/

Über die Autoren

Dr. Jörg-Volker Müller ist Geschäftsführer der Systemum GmbH & Co. KG. Er hat in leitender Funktion in zahlreichen Projekten erfolgreich IT-Systeme, Frameworks und Softwareprodukte realisiert. Seine fachlichen Schwerpunkte sind Software Engineering, Modernisierungen, Architekturen, Produktlinien und Varianten in Embedded Software und IT. Dr. Müller hat Informatik an der TU Braunschweig studiert und dort promoviert. Seit 2012 hält er regelmäßig Vorlesungen zum Thema Softwareengineering an der Hochschule Harz.

Roman Koch ist Software Architekt für eingebettete Systeme in der Entwicklung der GMC Instruments GmbH in Nürnberg. Herr Koch hat Informatik an der FAU Erlangen-Nürnberg studiert (Programmiersprachen und -methodik) und anschließend am Fraunhofer IIS geforscht (Kommunikationsnetze und -protokolle). Nach mehrjähriger Erfahrung in der hardwarenahen Softwareentwicklung und Testautomatisierung liegen fachliche Schwerpunkte aktuell im Umgang mit Bestandscode, Portierung und Überarbeitung.