Die wichtigsten Zutaten für moderne Softwareentwicklung in der Medizintechnik

Agile Entwicklung ermutigt jeden Einzelnen im Team, sich aktiv einzubringen

In der Softwareentwicklung verläuft der Weg zum erfolgreichen Software-Release in der Regel alles andere als linear. Laufzeitfehler und zu spät erkannte Sicherheitslücken, aber auch veraltete Tools und Methoden können zu Showstoppern werden. Zumindest vergrößern sie den Entwicklungsaufwand, führen zu Verzögerungen und können somit den Marktlaunch von Medizinprodukten gefährden. Um die Angreifbarkeit der Geräte im Internet of Medical Things zu minimieren, fordert der Markt zudem immer kürzere Zykluszeiten. Hinzu kommt die zunehmende Komplexität der medizinischen Systeme. Umso wichtiger ist es daher, moderne Sprachstandards sowie zeitgemäße Werkzeuge und Methoden in der Entwicklung zu nutzen, um robuste und stabile Releases mit vertretbarem Aufwand zu erzielen. Doch wie können zeitgemäße Werkzeuge und Methoden helfen, diese Risiken zu adressieren?

Autor:
Dr. Andreas Kuntz
ITK Engineering GmbH
www.itk-engineering.de

Medizinische Software muss sehr hohe Anforderungen erfüllen

Zunächst werden die Ziele und Randbedingungen medizinischer Softwareentwicklung betrachtet: Im Fokus stehen robuste Produkte für den medizinischen Einsatz. Hier zählen vor allem Stabilität und Fehlerfreiheit. Die Produktentwicklung ist in der Regel in einen festen Zeitrahmen eingebettet. Um Meilensteine einhalten zu können, muss die Parallelisierung der Entwicklung durch eine Vergrößerung des Teams zu einer effektiven Beschleunigung des Projektes führen. Ein beschränktes Budget macht kosteneffizientes Arbeiten essenziell. Gleichzeitig muss jedoch eine Verlagerung der Kosten von der Entwicklungs- in die Wartungsphase vermieden werden. Ein weiteres Ziel stellt die Reduktion der Komplexität dar. Die Entwickler sollten sich auf die wesentlichen Inhalte konzentrieren können, um die Geschwindigkeit im Verlauf eines langen Projektes beibehalten zu können. Wichtig ist zudem der Know-how-Transfer im Team, um Wissensinseln zu vermeiden und Mitarbeiter in zeitkritischen Phasen entlasten zu können.

Nachvollziehbarkeit und Reproduzierbarkeit

Im streng regulierten medizinischen Umfeld schreibt die IEC 62304 (Medizingeräte Software – Software-Lebenszyklus) zusätzlich Konfigurationsmanagement vor, woraus sich die Forderungen nach Nachvollziehbarkeit und Reproduzierbarkeit ergeben. Nachvollziehbarkeit meint Kontrolle der Konfigurationen, wie z. B. der Laufzeitumgebung, Kontrolle der Abhängigkeiten und Tracing von Änderungen. Reproduzierbarkeit verfolgt das Ziel, auch noch in vielen Jahren binär identische Artefakte erstellen zu können. Auch nach der Entwicklung sind regulatorische Anforderungen zwingend zu berücksichtigen. Mit der MDR sind die Anforderungen an die Post-Market Surveillance deutlich gestiegen. Bei der Beobachtung der Geräte im Feld stehen insbesondere Safety- und Security-Aspekte im Fokus. Beim Bekanntwerden von Fehlern und Angreifbarkeiten sollten Patches für die Produkte und alle Abhängigkeiten schnell ausgeliefert werden können, um die Gefährdung von Patienten und Angriffe auf Krankenhaussysteme zu vermeiden.

Zeitgemäße Methoden legen den Grundstein für kollaborative Softwareentwicklung

Wer heute an zeitgemäße Methoden denkt, dem wird sofort Agile einfallen. Hier gibt es zahlreiche Varianten. Ein gemeinsames Merkmal ist, dass die Arbeit in kurze Phasen gegliedert ist und die Entwickler in Entscheidungsprozesse eingebunden werden – dies fördert Eigenverantwortung und Committment. Die Arbeit wird mit Task Boards strukturiert, die ebenso Load Balancing erleichtern und für Transparenz und Nachvollziehbarkeit sorgen.

Vor der Übernahme eines jeden Feature-Branches in den Produktiv-Zweig wird in der Continuous Integration Pipeline ein Mindestreifegrad sichergestellt

Des Weiteren hat sich in der Open Source-Softwarewelt immer früheres Testen etabliert. Dies geht bis hin zu Test-First Ansätzen, wie z. B. Test-driven Development.

Die Vorzüge

liegen klar auf der Hand: Die Tests entstehen entwicklungsbegleitend und halten getroffene Annahmen fest. Die Einhaltung der Annahmen wird so zukünftig automatisiert geprüft. Dies hilft dabei, Fehler frühzeitig zu erkennen. Tests dokumentieren außerdem Schnittstellen und wichtige Anwendungsfälle und dienen so dem Wissensaustausch im Team. Die Verfügbarkeit von Git als Werkzeug zur Code-Versionierung hat die kollaborative Softwareentwicklung revolutioniert. Entwicklungsstränge können unabhängig voneinander vorangetrieben werden. Vor der Übernahme eines Feature-Branches in den Produktiv-Zweig kann ein Mindestreifegrad sichergestellt werden. Die Entwicklung kann in jedem Zustand vollständig an Kollegen übergeben werden, ohne dabei mit dem Produktiv-Zweig zu interferieren. Der Kontext zusammengehöriger Änderungen ist aus der Historie nachvollziehbar. So lässt sich neben dem Umfang auch der Grund der Änderungen ableiten. Auch Code Reviews sollten nicht nur „pro forma“ durchgeführt werden. Neben der offensichtlichen Qualitätssicherung sind sie ein wichtiges Mittel zum Know-how Transfer, zum Einbinden neuer Kollegen in das Entwicklerteam und unterstützen ebenso beim Mentoring. Reviews fördern außerdem die Kommunikation, den Austausch und die Zusammenarbeit im Team.

Aktuelle Werkzeuge helfen, Fehler frühzeitig zu erkennen

Aktuelle Compiler ermöglichen die Verwendung aktueller Sprachstandards. Diese sollen die Sprache vereinfachen und die Lesbarkeit und Verständlichkeit verbessern. Ein anschauliches Beispiel hierfür ist die Einführung des Drei-Weg-Vergleichsoperators „<=>“ in C++20, dem sog. Spaceship Operator. Mit dessen Hilfe leitet der Compiler automatisch einen vollständigen Satz von Vergleichsoperatoren für eigene Datentypen ab. Dies reduziert den Code und den Pflegeaufwand. Außerdem wurden die Optimierungsfähigkeiten der Compiler erheblich verbessert. Unter Umständen können heute Ergebnisse ganzer Funktionsaufrufe zur Compile-Zeit ausgerechnet werden. Die Verwendung aktueller SOUPS in der Entwicklung ist obligatorisch. Die neuste Version hat i.d.R. die wenigsten bekannten Fehler. Ihre Verwendung ist daher zum Erreichen der normativ geforderten Safety- und Security-Ziele unerlässlich. Automatische Code-Formatierung sollte heute ebenfalls eine Selbstverständlichkeit sein. Während statische und dynamische Codeanalyse bereits ein bewährtes Werkzeug ist, hat sich deren Verfügbarkeit deutlich verbessert:

Umfangreiche Compiler-Warnungen helfen dabei, potenzielle Fehler aufzudecken. Die OSS Compiler GCC und Clang bieten zahlreiche Sanitizer, die durch Instrumentierung schwer auffindbare Speicher- und Concurrency-Probleme oder auch undefiniertes Verhalten zur Laufzeit aufdecken. Durch die Aktivierung eines Sanitizers in allen Testläufen wird frühzeitig auf diese Probleme aufmerksam gemacht. Dies erspart dem Entwicklerteam viele Stunden kompliziertes Debugging.

Die Minimierung der eigenen Quellen spart Zeit und Aufwand in der Pflege

Jede Code-Zeile, die gelöscht wird, muss nicht gepflegt werden. Dies konsequent umgesetzt hat in der Praxis enorme Auswirkungen. Eine Möglichkeit, die Code-Basis zu verkleinern, liegt in der Nutzung externer Open Source-Bibliotheken. Denn für viele Standardprobleme gibt es etablierte und wiederverwendbare Standardlösungen. Beispielsweise kann durch konsequente Verwendung von Komponenten der Bibliotheken STL, Boost, Poco, u.v.m. die Trennung der Zuständigkeiten verbessert werden. Der Entwickler erhält eine gute Abstraktion und ausgereifte Schnittstellen. Darüber hinaus sind diese Bibliotheken in der Regel sehr gut getestet. Bekannte Security Probleme (CVEs) werden in öffentlichen Datenbanken erfasst und es ist nachvollziehbar, welche Versionen betroffen sind.

So viel wie möglich auslagern

Des Weiteren sollte aus den eigenen Quellen ausgelagert werden, was auslagerbar ist. Der Fokus sollte ausschließlich auf projektspezifischen Ergänzungen liegen. Eigenentwicklungen sollten nur stattfinden, wenn noch keine pas-

Ziele und Randbedingungen werden gemeinsam mit dem Kunden festgelegt

Anschauliche Darstellung zur Optimierung der Prozesse

sende Lösung verfügbar ist. Dann ist jedoch ein Refactoring dringend empfohlen, sobald sich ein Standard etabliert hat. Jede eingesparte Zeile reduziert den langfristig dominanten Wartungsaufwand und erhöht die Schnelligkeit des Entwicklerteams. Gemäß dem Motto „Don't repeat yourself" sollte auch nicht neu implementiert werden, was von anderen bereits verfügbar ist. Neben wiederverwendbaren Standardlösungen ist ein Dependency Management System ein weiteres Schlüsselelement. Viele Sprachen verfügen über leistungsfähige Paket-Manager, wie z. B. pip für Phyton, Cargo für Rust oder Conan für C++.

Paket-Manager

Conan ist seit einigen Jahren für C++ verfügbar und könnte sich zum defacto-Standard etablieren, da die Verwaltung von Abhängigkeiten wesentlich erleichtert wird. Der Paket-Manager erhält eine deklarative Liste der Abhängigkeiten und stellt die benötigte Bibliothek für das Zielsystem unter Beachtung der verwendeten Toolchain zur Verfügung, entweder vorkompiliert aus dem Archiv oder unter Beachtung der spezifischen Anforderungen lokal kompiliert und lokal gecached. Mit der Verwendung eines Paket-Managers geht ein weiterer Vorteil häufig einher: Es können alle SOUPs und Tools aus den Quellen gebaut werden. Dies ist selbst ohne die Verwendung eines Paket-Managers ratsam und verringert die Hürden, wenn die Entwickler in die Quellen schauen müssen. Die korrekte Version ist so lokal verfügbar. Der Entwickler hat außerdem die volle Kontrolle, falls gepatcht werden muss. So kann direkt die Ursache des Problems beseitigt und ein Workaround vermieden werden. Wird der Patch anschließend upstream den Bibliotheksverantwortlichen zur Verfügung gestellt, unterstützt dies einerseits die Community und reduziert langfristig den eigenen Pflegeaufwand der Patches.

Automatisierung

reduziert die Komplexität und erhöht sowohl die Geschwindigkeit als auch die Reproduzierbarkeit. Eine konsequente und durchgängige Automatisierung aller Build-Schritte und der Testausführung bringt anfangs möglicherweise einen geringen Mehraufwand mit sich, langfristig jedoch überwiegen die Vorteile. Die Prozesse der Entwickler werden vereinfacht und damit die Komplexität ihrer alltäglichen Aufgaben reduziert. Das Team erhält einen Geschwindigkeitsvorteil in der tagtäglichen Arbeit und während der „heißen" Release-Phase. Skripte, die zur Automatisierung genutzt werden, sind gleichzeitig eine präzise Dokumentation der Vorgehensweise und der benötigten Schritte. Die fördert den Know-how Transfer und reduziert Wissensinseln. Die Verwendung eines aktuellen deklarativen Build-Systems oder eines entsprechenden Generators, wie z. B. CMake/Ninja für C++, sollte selbstverständlich sein. Auf diese Weise kann für jedes Artefakt ein Target definiert und mit den zugehörigen Quellen assoziiert werden. Nun müssen nur noch die Abhängigkeiten zwischen den Targets festgelegt werden. Targets der SOUPs werden üblicherweise importiert. Den Rest erledigt der Generator. So können Quellen auf ein Minimum reduziert und Redundanzen vermieden werden.

Docker oder LDX

Virtualisierungslösungen, wie z. B. Docker oder LDX, erlauben eine deklarative Definition der Umgebung für Entwicklung, Continuous Integration (CI) und ggf. für den Betrieb. Alle Entwickler können sich so auf eine garantiert einheitliche Betriebssystem-Umgebung und Werkzeug-Konfiguration verlassen. Das bekannte „Works on my machine"-Problem sollte damit der Vergangenheit angehören. Auch die in der CI aufgedeckten Fehler lassen sich so zuverlässig durch den Entwickler reproduzieren.

Continuous Integration

In größeren Projekten mit vielen Produktvarianten, mehreren unterstützten Betriebssystemen und Compilern sowie ausführlichen Tests wird nicht jeder Entwickler immer alle notwendigen Konfigurationen prüfen können. Dafür ist die Continuous Integration ein etablierter Prozess und eine große Entlastung für den Entwickler. CI erlaubt die konsequente und automatisierte Prüfung aller Branches für alle Konfigurationen, bevor diese in den Produktiv-Zweig übernommen werden. Außerdem lässt sich die automatische Testausführung auf der Ziel-Hardware realisieren. Tests können gegen spezielle Simulationen laufen. So können beispielsweise Hardwarefehler simuliert oder durch simulierte Zeit die Ausführungsdauer langwieriger Tests verkürzt werden. Jeder Software-Stand lässt sich so umfassend prüfen. Fehler und Seiteneffekte in Medizinprodukten werden frühzeitig aufgedeckt.

Lohnt sich also eine Modernisierung der Softwareentwicklung?

Diese Frage kann mit einem klaren „Ja" beantwortet werden. Modernisierung leistet einen wesentlichen Beitrag zum Erreichen der anfangs genannten Ziele. Es werden dadurch die Robustheit der Software, die Geschwindigkeit, Kosteneffizienz und der Know-how Transfer gefördert und die Komplexität wird reduziert. Ebenso werden Nachvollziehbarkeit und Reproduzierbarkeit unterstützt sowie die Einhaltung von Safety- und Security-Zielen für Medizinprodukte, insbesondere in der Maintenance-Phase, erleichtert. Modernisierung und Automatisierung helfen dabei, Entwickler zu entlasten und die Softwareentwicklung zielgerichtet zu gestalten. Der Fokus wird so auf die eigentliche Problemlösung gelenkt, wodurch Fehler vermieden werden können. Modernisierung muss zudem nicht teuer sein. Die aufgeführten Bausteine sind im Open Source-Umfeld schon lange im Einsatz und in diversen Projekten reichlich erprobt. Viele der aufgeführten Werkzeuge stehen als freie Open Source-Software zur Verfügung.

Doch wo fängt man mit der Modernisierung an?

Die konsequente Einführung der folgenden drei essenziellen Methoden können beim Start helfen: Entwicklungsbegleitende Tests, Feature-Branch-basierte Entwicklung sowie gegenseitige Unterstützung in Code Reviews sind der Schlüssel für einen geradlinigen Weg zum nächsten Software-Release. ◄