LANGLEBIGE PROJEKTE

Software fürs Leben

Vorkehrungen treffen, damit Quellcode nach langer Zeit wiederbelebt werden kann.

Als der Auftraggeber mit ein paar 3,5-Zoll-Disketten ins Sitzungszimmer trat, staunten wir nicht schlecht. „Unser Novell-Server steigt immer öfter aus, wodurch die Produktion lahmgelegt wird." Ob wir die Software auf Windows portieren und an SAP anbinden können. „Und die Firma, welche die Software geschrieben hat?" – „Die gibt es nicht mehr."

Es betraf Steuerungs- und Visualisierungssoftware aus dem Jahr 1996 für ein paar Sondermaschinen, geschrieben in Borland C, MS-DOS. Die Maschinen wurden vor sechs Jahren nach Tschechien verlegt. Niemand wusste genau, wie sie funktionierten, Dokumentation gab es keine.

Wie groß würden Sie den Aufwand schätzen, um einen solchen Auftrag zum Erfolg zu bringen?

Mittlere bis große Softwareprojekte sind teuer und sollten daher möglichst lange leben. Auch für Software gilt: Was gut gepflegt wird, lebt länger. Was aber, wenn das Projekt erfolgreich abgeschlossen wird und der Quellcode in den Archivschrank verschwindet? Was, wenn nach zehn Jahren ein neues Modul an diese archivierte Software angebunden werden soll? Sind die Entwickler dann noch da? Ist das Domänenwissen von damals noch vorhanden? Kann man das Projekt mit der neuen Entwicklungsumgebung noch kompilieren? Gibt es NuGet [1], NPM [2] oder Yarn [3] in zehn Jahren noch?

Alles Wichtige an einem Ort

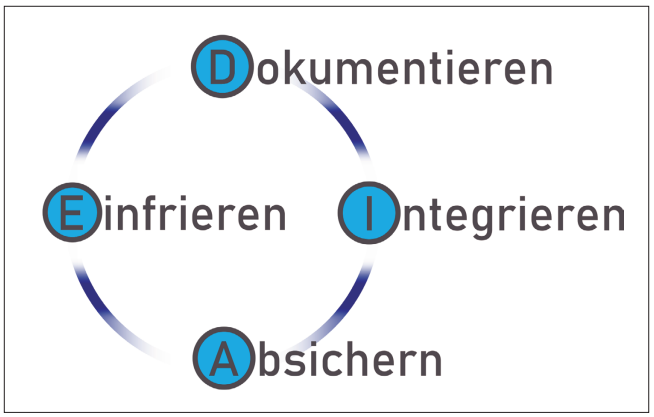
Das Domänenwissen in Confluence [4], die Anforderungen in Jira [5], externer Code auf einem NuGet-Server [1], Quellcode in Git [6] und der Buildprozess in der Azure Cloud [7]: So kann ein mittleres bis großes .NET-Projekt heute aussehen. Während der Entwicklungsphase funktioniert das wunderbar. Wenn das Projekt aber abgeschlossen ist, muss man alles für die Wiederbelebung Wichtige an einem Ort ablegen. Häufig dauert es nicht lange, bis eines der Systeme durch ein Neueres abgelöst wird. Auch der Kunde wird an einem funktionierenden Back-up interessiert sein, da er keine Garantie hat, dass es seinen Dienstleister in ein paar Jahren noch gibt.

„Alles Wichtige" ist alles, was es braucht, um das Projekt auch in zehn Jahren noch zu verstehen, zu kompilieren, zu deployen und laufen zu lassen. Dies bedingt Maßnahmen in folgenden Bereichen (vergleiche Bild 1):

- Dokumentation
- Beseitigen externer Abhängigkeiten
- Einfrieren der Entwicklungs- und Laufzeitumgebung
- Best Practices für nachhaltige Software

Dokumentation

Wenn ein Wiki nicht nachhaltig genug ist, um wichtige Informationen zu speichern, wohin dann mit dem projektrelevanten Domänenwissen? Eine gute Lösung ist ein technisches Handbuch, das zusammen mit dem Quellcode gepflegt und gespeichert wird. Ein solches Dokument eignet sich ebenfalls, um alle Softwarefunktionen zu beschreiben, weil diese in der Regel als Softwareanforderungen im Ticketsystem gefangen sind. Das Handbuch kann man während des Buildprozesses automatisch versionieren und als PDF exportieren. Das PDF wiederum kann als Online-Hilfe im Produkt integriert werden. Dies verlangt jedoch vom Entwickler (oder zum Beispiel dem Product Owner), dass dieser nach der Implementation einer neuen Funktion das Handbuch aktualisiert. Ein Eintrag in der „Definition of Done" der User Stories sorgt dafür, dass nichts „absichtlich" vergessen wird.



Vier Punkte, die beachtet werden sollten (Bild 1)

Softwarekommentare sind ein weiteres Hilfsmittel, um das Projekt zu dokumentieren. Sobald etwas nicht offensichtlich ist oder an einer Stelle spezielles Domänenwissen gefordert wird, sollte der Entwickler dies ausführlich in einem Kommentar beschreiben. Damit ist nicht nur seinen Nachfolgern geholfen, sondern auch ihm selbst. Denn was heute kristallklar ist, kann schon nach einem Jahr für jeden ein Rätsel sein.

Zur projektrelevanten Dokumentation gehören auch Handbücher von Drittanbietern (API-Dokumentation, Peripherie). Auch diese sollten am gleichen Ort wie das Handbuch im Projekt gespeichert werden.

Zur Sicherheit exportieren Sie das Wiki und das Ticketsystem als PDF und archivieren es gleich mit.

Beseitigen externer Abhängigkeiten

Extern ausgelagerter Code ist ein Risiko für die Wiederbelebung der archivierten Software. Spätestens in der Abschlussphase sollte der gesamte externe Code lokal im Projekt integriert sein. Wer garantiert Ihnen, dass es die verwendeten Pa-

kete in zehn Jahren noch auf NuGet oder NPM gibt? Gibt es diese Paketmanager dann überhaupt noch?

Ein paar Tipps für die Nachhaltigkeit:

- Verzichten Sie auf externe Pakete, wo immer es geht. Ein AutoMapper ist mit zwei einfachen Unit-Tests zum Beispiel überflüssig.
- Integrieren Sie, wenn möglich, die Funktionalität als Quellcode im Projekt. Nicht mehr weiterentwickelte Bibliotheken können so für neuere Framework-Versionen kompiliert werden.
- Verstecken Sie externe Funktionalität hinter einer Fassade; das macht sie einfacher austauschbar.
- Bleiben Sie an der Oberfläche; benutzen Sie nur Basisfunktionalität. So verhindern Sie, dass Sie sich von einem bestimmten Hersteller abhängig machen, weil nur sein Produkt diese spezielle Funktionalität anbietet.
- Schalten Sie alle automatischen Updates aus (Dependencies, Entwicklungsumgebung et cetera). So verhindern Sie, dass Sie später unerwartet in Teufels Küche kommen.
- Vergessen Sie nicht die lokal installierten Bibliotheken oder spezielle DLLs, die sich im GAC (Global Assembly Cache) verstecken [8]. Am besten werden die originalen Installer ebenfalls im Projekt abgelegt.

Einfrieren der Entwicklungs- und Laufzeitumgebung

Den Quellcode von heute können Sie mit großer Wahrscheinlichkeit bereits in fünf Jahren nicht mehr kompilieren, ohne das Projekt zuerst mühsam zu konvertieren. Dies ist insbesonders dann ineffizient, wenn nur ein kleiner Fehler schnell behoben werden muss. Erstellen Sie deshalb beim Projektabschluss eine virtuelle Maschine mit der gesamten Entwicklungsumgebung. Übrigens ist die Virtualisierung der Laufzeitumgebung von Servern heute schon gang und gäbe.

Automatisierte Tests

Tests sind wie eine Versicherung: Sie verhindern möglichen Schaden, haben aber auch ihren Preis. Die Kosten für das Schreiben und insbesondere für die Aufrechterhaltung automatisierter Tests werden oft unterschätzt. Wie viele Projekte haben Sie schon erlebt, in denen nicht alle Tests grün (passed) waren, weil der Zeitdruck zu hoch war oder am Ende das Budget dafür fehlte?

Automatisierte Tests können für den Nachfolger sehr hilfreich sein. Weil dieser am Anfang unmöglich den ganzen Code verstehen kann, ist die Wahrscheinlichkeit groß, dass seine Änderung an einem anderen Ort etwas kaputt macht. Ein guter Test würde diesen Fehler aufdecken. Meistens aber muss der Test nur nachgepflegt werden, damit er nach der Änderung wieder grün wird. Dadurch wird der Nachfolger gezwungen, sich mit anderen Codestellen zu beschäftigen, und er wird schneller mit dem Projekt vertraut.

Sonstiges

Rechnen Sie damit, dass die Entwickler von heute in ein paar Jahren nicht mehr da sind. Ingenieure werden auf manchen Portalen sogar darauf hingewiesen, dass „mehr als fünf Jahre Verweildauer vor einem Jobwechsel mit Vorsicht zu genießen sind“ [9]. Also muss der Arbeitgeber auf den Verlust seiner Entwickler vorbereitet sein.

Die oben genannte Dokumentation ist daher ein Muss für den in der Regel nicht überlappenden Wissenstransfer. Wichtig ist es auch, bei Lösungen an der Oberfläche zu bleiben und eher pragmatisch vorzugehen. Entwickler wählen meistens die kompliziertere Lösung, weil sie sich damit verwirklichen können (Software-Ego). Durch unnötige Komplexität verliert der Nachfolger viel Zeit damit, sich einzuarbeiten, und das Risiko, dass er einen Fehler macht, ist unnötig groß. Die Lösung soll nicht komplizierter sein als das Problem.

Bleiben Sie konservativ: Wenden Sie nur etablierte Technologien an, vermeiden Sie neue Technologien mit Hype-Status. Teilen Sie den Code in unabhängige, eigenständige Komponenten auf, so gut es geht. Diese komponentenbasierte Entwicklung stand vor 15 Jahren im Vordergrund, ist aber heute irgendwie untergegangen. Eigenständige Komponenten sind einfacher zu testen und, wenn sie gut geschrieben sind, ideal zur Wiederverwendung. Der Nachfolger kann sich auf eine Komponente konzentrieren, ohne den Rest der Software verstehen zu müssen.

Fazit

Die oben erwähnten Best Practices sorgen dafür, dass archivierte Software nach langer Zeit mit möglichst wenig Aufwand wiederbelebt werden kann. Viele der erwähnten Maßnahmen gelten aber auch für Projekte, die kontinuierlich gepflegt werden. Die meisten Softwareentwickler sind schon mit einer funktionierenden Lösung zufrieden. Ein erfahrener Entwickler aber hat einen weiteren Horizont und macht sich darüber hinaus Gedanken über Nachhaltigkeit und den Kostenaspekt einer Lösung. ■

[1] NuGet Package Manager, www.nuget.org
[2] NPM, JavaScript Package Manager, https://npmjs.com
[3] Yarn JavaScript Package Manager, https://yarnpkg.com
[4] Wiki-Software Confluence, www.dotnetpro.de/SL1906LangeProjekte1
[5] Atlassian Jira, www.dotnetpro.de/SL1906LangeProjekte2
[6] Versionsverwaltung Git, https://github.com/git
[7] Microsoft Azure, https://azure.microsoft.com
[8] Code Project, Copy DLL from GAC, www.dotnetpro.de/SL1906LangeProjekte3
[9] Ingenieur.de, Technik-Karriere-News, www.dotnetpro.de/SL1906LangeProjekte4

Erik Stroeken
arbeitet als Expert Software Engineer bei der Noser Engineering AG in der Schweiz. Der gebürtige Niederländer ist seit fast 35 Jahren mit Herz und Seele in der Softwareentwicklung im industriellen Umfeld tätig.
erik.stroeken@noser.com

dnpCode A1906LangeProjekte