Extreme programming (XP)

Thomas Reinwart

1. Automatisieren eines Build Prozesses für .net

Unter einem Build Prozess versteht man jenen Abschnitt bei der Entwicklung eines Produktes, bei dem sichergestellt wird, dass automatisch bzw. auf Knopfdruck aus dem aktuellen Quellcode ein qualitätsgetestetes Installationspaket erstellt wird. Das geht natürlich nicht einfach von selbst, dazu sind organisatorische Maßnahmen, Disziplin und einige Tools notwendig.

Ein Build Prozess ist ein entscheidender Abschnitt bei der Softwareentwicklung. Vor allem dann, wenn es mehrere Beteiligte eines Entwicklungsteams gibt und jeder einen Teil eines Gesamtprojekts beiträgt. Alle Teile müssen zusammenpassen und funktionieren.

In meiner C++ und VB Zeit (vor .net) hatte ich bei einer Firma in meiner Rolle als Entwickler, Qualität und Setup Verantwortlicher sehr viel Zeit damit verbracht, ein funktionstüchtiges Produkt in ein ebenso funktionstüchtiges Installationssetup zu packen. Das war zu jener Zeit, als die dll-hell (COM) ihren Höhepunkt hatte, verschiedenste Windows Generationen (Windows 95, NT, 2000, alles in verschiedenen Sprachen und Servicepack Versionen) gleichzeitig bei Kunden installiert waren. Bei der Entwicklung im Team waren damals (automatisierte) Tests technisch nicht bekannt bzw. in einem vertretbaren Budget umsetzbar. Es gab keine Möglichkeiten SourceSafe Eincheck Regeln zu definieren, oder eine Beschreibung der Änderung zu hinterlegen (Bsp. Aufruf einer API die nur auf Windows xy ab SP z verfügbar ist bzw. welches anderen Setups als Grundvoraussetzung benötigt) auf die im manuellen Build und in der Setup Routine am Ende Rücksicht genommen werden konnte.

Dazu eine Planung, bei der das Produkt erst ganz am Ende funktionsfähig war, und die nicht mehr viel Zeit für Tests zuließ. Das ganze wiederholte sich für jedes Produkt, immer wieder. Viel Zeit wurde damit ungenützt verschwendet. Mit der .net Generation kamen technisch und von der Denkweise her Neuerungen, mit der solche Arbeitsweisen nicht mehr notwendig waren. Die Schlagwörter dazu sind Extreme Programming und technisch Reflection in .net, mit dem Unit Tests möglich wurden. Die Tools gab es für Java schon längst.

1.1 Vorgehensweise

Die erste Grundlage ist, eine Überzeugungsleistung im Development Team zu leisten und die Vorteile zu veranschaulichen, die alten jahrelange Gewohnheiten („es geht ja eh" – irgendwie) zu ändern. Es betrifft nicht nur die Entwickler selber, auch die Organisation - oft bis zur Geschäftsleitung, hier die Veränderung eines Prozess zu veranschaulichen.

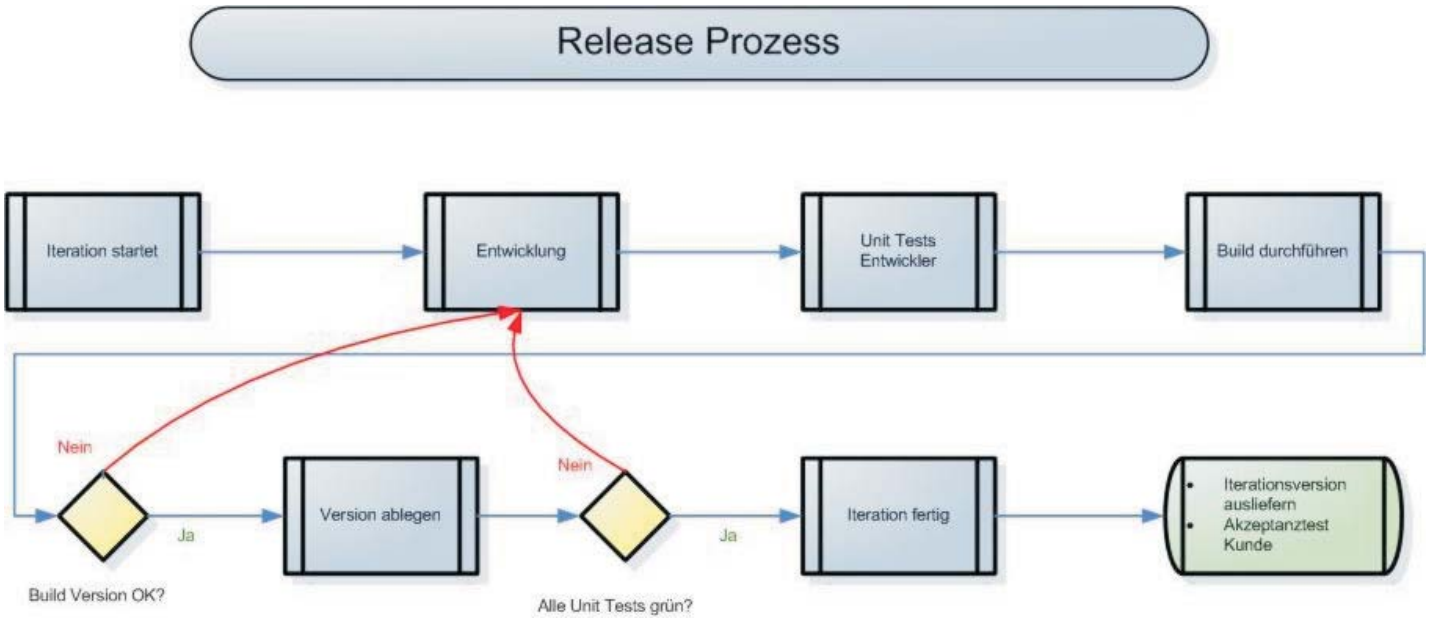
Die technischen Voraussetzungen sind ein Source Safe System, ein Build Server (kann auch der Source Safe Server sein) und weitere Software Tools.

Die organisatorischen Voraussetzungen sind Aufgeschlossenheit für Veränderungen, Disziplin im Umgang mit dem Source Safe System, schreiben von Unit Tests, Einhaltung der Qualitätsrichtlinien des Codes, Code Doku.

Ein Build Prozess wird gleich zu Beginn an eingerichtet. Es gibt laufend einen messbaren Fortschritt, immer einen aktuellen Stand für Präsentationen und für Prototyping.

1.2 Prozess

Beispielhafter Prozess, wie er in gehandhabt wird.



1.3 Funktionsweise

Zur Automatisierung des Builds von .net solutions (Visual Studio sln 2003 bis 2008) gibt es Open Source Varianten sowie von Microsoft den Team Foundation Server.

Was ist der grundsätzliche Sinn:

Der Prozess des Builds ist wiederkehrend, veranschlagt viel Zeit und ist fehleranfällig. Durch die Automatisierung wird der aktuelle Sourcecode aus dem System geholt, ein Build durchgeführt. Der Sourcecode kann entweder von einem bestimmten Label geholt werden, oder immer der zuletzt eingecheckte Stand. Das setzt in der Organisation voraus, dass hier nur funktionierender und nur compilierbarer Code vom Team ins Source Safe eingecheckt wird. Bei einem Major Release muss zusätzlich ein Label (mit Versionsnummer und Beschreibung) im Source Safe gesetzt werden.

Das Anstoßen kann über den Windows Sceduler (**Nightly Build**), durch manuelles anstoßen oder durch einen Mechanismus, der bei jedem Check-In Vorgang einen Build durchführt. (eigenen Build Server wegen Performance von Vorteil) Die Buildnummer kann in den Assemblies automatisch vergeben werden (Anstatt einer Buildnummer kann man auch einen Timestamp verwenden), so gibt es nach einer Auslieferung immer den Zusammenhang der Unit Test und Build Reports.

Damit wird erreicht, dass ich vom Projekt auf Knopfdruck eine Version erstellen kann, mit dem Ergebnis compilierfähig oder nicht. Das reicht aber nicht. (**Packaging the build result**)

Um nun eine messbare Aussage über Qualität und Fortschritt zu erhalten, sind Unit Tests sinnvoll. (**Unit test build result**) Dieser Begriff ist einer der Punkte aus XP. Heutzutage werden Sofwareprojekte mit diesem Ansatz umgesetzt.

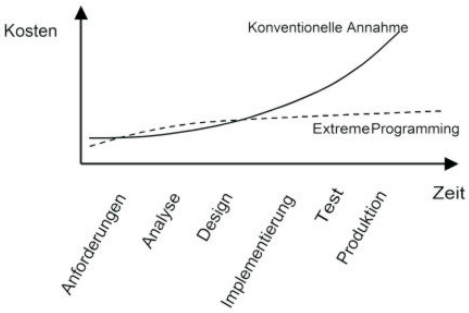
Die Ergebnisse des Build werden per Email versendet, Ergebnisse in Form von Assemblies am Server abgelegt. Im Build Verzeichnis mit Build Nummer und/oder Timestamp im IIS, ftp, oder Filesystem als Setup, ClickOnce oder Xcopy. Zusätzlich wird die aktuell erstellte Version am Testsystem automatisch installiert. Die Ergebnisse eines NUnit Tests sind XML Dateien, deren Inhalt man ohne Probleme in eine DB schreiben kann. Im Ergebnis steht welcher Test es war, der Status und die Laufzeit. Bei Ablage

der Daten in einer Datenbank lassen sich diverse Auswertungen über Anzahl der Tests und deren Laufzeit analysieren. Es wird immer eine Debug und Release Version gebuildet. Das hat sich in der Praxis bewährt, manchmal ist eine der Build Varianten nicht buildfähig.

2 XP Prinzipien

Seit einigen Jahren wird **Extreme programming (XP)** in der Softwareentwicklung eingesetzt. Anstatt des klassischen Wasserfall-Models durchläuft der Entwicklungsprozess immer wieder kürzere Zyklen von Design, Implementierung und Testen. Im der aktuellen Iteration werden die derzeit bekannten Anforderungen umgesetzt. Zu Beginn eines Projektes sind meist noch nicht alle Anforderungen des Kunden bekannt. Auf eine vollständige Erhebung zu Beginn wird verzichtet.

Durch verschiedene Maßnahmen soll die Qualität und Flexibilität der Software gesteigert werden.



Einige der Prinzipien

Pair-Programming: (Zwei Entwickler teilen sich eine Tastatur und Monitor – einer codiert, einer denkt mit) Vorteil: Wissensaustausch technisch und projektbezogen

Build: Integration der einzelnen Komponenten zu einem lauffähigen Gesamtsystem in kurzen Zeitabständen

Test first development: Es werden erst die Unit-Tests geschrieben, bevor die eigentliche Funktionalität programmiert wird. Die Tests werden vom Entwickler ausgeführt, bevor der Code eingecheckt wird.

Enge Einbeziehung des Kunden, d.h. der Kunde gibt das Iterationsziel vor und hat sofort die Möglichkeit **Akzeptanztests** durchzuführen.

Laufende **Refaktorisierung**, ständige Architekturverbesserung

Akzeptable Arbeitszeit: 40-Stunden-Woche, denn Überstunden mindern die Freude an der Arbeit und somit auch die Qualität des Produkts.

Der Kunde testet die aktuelle Build Version durch den Akzeptanztest. Ein Akzeptanztest ist oft Grundlage für eine Rechnungsstellung. Beim Akzeptanztest betrachtet der Kunde die Applikation und ob das Verhalten den Anforderungen und Wünschen entspricht, nicht aber den Sourcecode. Am Ende gibt es ein Prüfprotokoll, ob die Anforderungen mit dem Pflichtenheft übereinstimmen.

Die Qualitätsicherung von Software lässt sich mittels definierten **Unit Tests** überprüfen. Die Tests überprüfen das Verhalten von Teilen einer Applikation. Ein Test selber zielt immer nur auf einen kleinen Bereich ab, auf eine Unit. Eine Unit ist eine public Methode einer Klasse, die von der Anwendung genutzt wird. Der Unit Test ruft die Methode auf, mit gültigen und ungültigen Parametern. Das gewünschte Verhalten der Methode wird im Test überprüft.

Zu jedem Projekt wird in der Solution ein gleichnamiges Projekt mit der Endung (.test für die Übersicht) vom Typ dll erstellt. Unit Tests werden normalerweise nicht ausgeliefert. Im Visual Studio Team System kann das Gerüst des Unit Test inzwischen auf Knopfdruck generiert werden. Das Ausführen übernimmt Visual Studio 2008 selber, oder PlugIns fürs Studio oder externe Programme wie Nunit.

Welchen Sinn hat das Ganze: wenn ich eine Methode implementiere, gibt es dazu ja auch Vorstellungen im Kopf, eine Vorgabe oder ein Pflichtenheft, auf jeden Fall ein klares Verhalten und nicht Verhalten (Fehlerfälle, Exceptionhandling). Mit einem Test des Verhaltens kann immer eindeutig festgestellt werden, ob es nach Änderung, Erweiterung im Sourcecode eine Änderung der Methode bewirkt. Also z. B. wenn ein Kollege etwas ändert, und es doch Auswirkungen hat, an die keiner gedacht hat. (**Seiteneffekte ausschließen**) Es muss nicht einmal eine Änderung im Sourcecode sein, es kann darum gehen, sein Produkt nach Einspielen eines Security Hotfixes , eines SP des Betriebssystems oder auf einem neuen Betriebssystem zu testen.

Die meisten Entwickler werden nun denken, dass ist ja sehr zeitintensiv, da muss ich ja so viele Tests schreiben – „Jein". Es ist ein Selbstschutz für den Entwickler, damit er bestätigen kann, die Implementierung laut Definition erfolgreich umgesetzt zu haben. Der erstellte Unit Test wird nun so lange wiederkehrend ausgeführt, solange auch der Code dafür existiert, bis zum Ende der Lifetime des Produkts. Der Einmalige Aufwand der Unittest Erstellung zum Zeitpunkt wo die Anforderungen noch klar sind, steht in keinem Verhältnis für den Aufwand der über Jahre hinweg notwendig werden kann, einen Qualitätslevel hinterherzuprogramieren.

Für jede genutzte Methode muss es Unit Tests geben. Ein Test besteht immer nur aus wenigen Zeilen. Den Mock Daten (fixen Daten bzw. Objekte zum Testen), dem Aufruf der Methode die getestet werden soll, und dem Vergleich erwartetes und tatsächliches Resultat). Jeder Entwickler schreibt immer seine Unit Test zu seinem Code. Beim Schreiben des Tests bemerkt man nun, wie die eigene Klasse zu benutzen ist. Oft stellt sich nun heraus, dass dies gar nicht so angenehm ist, etwas fehlt, oder gleiche Parameter mehrmals übergeben werden, ein Parameter fehlt usw. Deswegen gibt es den XP Ansatz, den Unit Test vor der bzw. gleichzeitig mit der Implementierung der Klasse umzusetzen. Dadurch ergibt sich nämlich ein entschiedener Vorteil: Man denkt vorher nach, was man haben möchte, schreibt kein unnötigen Code (noch mehr Tests). Das ist ein Umdenkprozess beim Entwickler, aber so arbeitet man wesentlich effektiver. D.h. ich schreibe zuerst die Unit Tests, das spiegelt für mich die API dar, die ich mir vorstelle und auch wie ich das Verhalten der Methoden haben möchte. Pro Methode gibt es jeweils mehrere Tests. Da das Projekt inklusive der Tests kompilierbar ist, habe ich nun erreicht, dass mein Soll-Ist-Vergleich im Build messbar wird. Zu Beginn gibt es viele Tests, die fehlerhaft sind (rot), da sie noch kein Ergebnis erhalten, aber immer mehr Tests werden grün und damit funktionsfähig.

Code Wartbarkeit

Die Zeit der Fehlersuche und Behebung hat unmittelbar mit der Codequalität, dem Einhalten der Coding Richtlinien, zu tun. Jedes Teammitglied hat sich daran zu halten, jeder muss den Code des anderen lesen und ausbessern können. Um die Wartbarkeit zu verbessern, wird so genanntes **Refactoring** angewandt. Das Betriff Lesbarkeit, Testbarkeit durch unit tests, Modularität und Redundanz. Refactoring wird nur bei funktionierendem Code durchgeführt, bei dem die Funktion erhalten bleibt. Damit sichergestellt wird, dass das Verhalten gleich bleibt, werden Unit Tests verwendet. Dafür gibt integrierte Möglichkeiten im Visual Studio, aber auch Tools, etwa den Resharper der Firma Jetbrains.

Praxis

Beim der Neuerstellung (**Projektstart**) eines Visual Studio Projects werden die einzelnen Projekte (dll, Forms, Web, …) festgelegt. Parallel dazu werden die Projekte für Unit Tests erstellt. (.test). In den Unit Tests manuell bzw. automatisch erstellt. Durch das Konzept der Software Architektur ist mir bekannt, welche Klassen welche Funktionen wiederspiegeln.

Unter Berücksichtigung des XP Ansatzes, die Unit Test zuerst zu schreiben, werden nun die Unit Test erstellt. Diese sind jetzt beim Ausführen rot, aber das ist beabsichtigt. Damit lässt sich nun von Projektstart an eine Qualitäts- und Fortschrittsmessung protokollieren, die Entwicklung ist von Anfang an transparent. Aussagen von Entwicklern wie *„ich bin zu 75% fertig"*, die bisher aus dem Bauch heraus getroffen wurden, werden damit überprüft. Ich kann quasi zusehen, wie die geplante Iteration fertig wird. Es kann kein Bereich der Implementierung vergessen werden, Abhängigkeiten der Klassen werden sichtbar, eine Verteilung im Team ist wesentlich einfacher möglich. Am Ende das Tages sieht jeder ein Ergebnis, nämlich das ein oder mehrere Tests grün sind, also messbar auch für den Entwickler, psychologisch wichtig, etwas geleistet zu haben.

Voraussetzung ist immer, dass es ausreichend Unit Tests gibt, die tatsächlich eine sinnvolle Prüfung der Methode vornehmen. Für die Verwaltung der **Unit Test Ergebnisse** gibt mehrere Möglichkeiten: sammeln der Email HTML Reports, schreiben der Ergebnisse in eine Datenbank, Reporting Server verwenden.

Was mache ich aber bei einem Projekt, wo es noch keine Unit Tests gibt? (**Existierendes Projekt**)

XP sagt, ein und derselbe Fehler darf kein zweites Mal auftreten. Wie ist das möglich? Durch Unit Tests. Wenn ein Fehler auftritt, ist es immer der erste Schritt diesen zu reproduzieren. Genau das ist der Code für Unit Test. Der nächste Schritt ist, einen Bugfix dafür zu schreiben. Anschließend wird der Unit Test ausgeführt, er wird grün sein. D.h. für diesen Fehler gibt es nun einen Test, den ich im Build ausführen kann. (**Wiederauftreten - Regression**)

Change Requests – Änderungen im Sourcecode notwendig

Die Ausgangssituation ist, es gibt eine Anwendung und einen Unit Test dazu. Ein Entwickler muss eine Änderung im Code eines anderen Entwicklers vornehmen. Der Entwickler überprüft nun zuerst den bestehenden Test, dieser wird funktionieren. Er erkennt nun das Verhalten des Codes. Die Änderung kann vorgenom-

2.3 Vergleich der Tools

Funktion	Open Source	Microsoft Visual Studio / TFS
Unit Tests (x)	Nunit	Visual Studio Team System
Automatischer Build	Nant	Team Foundation Server
Source Analyse	Ndepend	Visual Studio Team System
Code Dokumentation Erstellung	Ndoc	Sandcastle
Build Report	HTML Reports basierend auf Nunit xml Ergebnissen	Team Foundation Server (umfangreiches Reporting)
Code Richtlinien Check		FxCop, Visual Studio Team System
Abdeckungsgrad von Unit Tests	NCover	Visual Studio Team System

(x) Nunit Tests sind von Syntax her minimal Unterschiedlich zum Microsoft Unit Test Syntax

men werden. Im Anschluss wird der vorige Unit Test durchgeführt, bis dieser wieder grün ist. Zur Sicherheit werden auch alle anderen Unit Tests durchgeführt, ob es hier nicht auswirkende Zusammenhänge gibt. (**test-driven software development**) Es wird nichts implementiert, was nicht auch getestet wird.

Unit Tests zielen auf Teile der Business Logik, Datenbank Layer ab. Nicht für Web oder Windows Forms GUI geeignet, dafür gibt es Produkte wie Mercury. D.h. die Qualitätssicherung durch Unit Tests wird entscheidend besser, jedoch muss ein User auch die Oberfläche durchklicken und das Ergebnis mit dem Designvorschlägen des Pflichtenhefts überprüfen. Vor allem bei Web Anwendungen kommen noch viele Faktoren hinzu, Bsp. gefühlte und tatsächliche Performance, Verhalten des Browsers etc.

2.1 Nant

Das N steht für .net, ursprünglich gab es diese Tools nur in Java. Für Java lauten die Namen der Tools ohne „N".

Nant ist ein Build Tool für .net, Open Source. Es unterstützt mehrere SourceSafe System, darunter Microsoft Source Safe und TFS.

2.2 Nunit

Ursprüngliches OpenSource Tool für .net Unit Tests. Inzwischen im Studio integriert.

2.4 Team Foundation Server (TFS)

Der TFS ist eine all in one Lösung. Neben einem SoureSafe System beinhaltet er auch gleich den Build. Es können Checkin-Regeln definiert werden, nach dem z.B. nur complierbarer Code mit Dokumentation eingecheckt werden kann. Aber das kann in einer super heißen Hotfix-Phase vom Entwickler umgegangen werden. Auf TFS Serverseite hat der Entwicklungsleiter immer die komplette Information über den CheckIn Vorgang, d.h. etwaige Unstimmigkeiten fallen sofort auf.

Der Vorteil gegenüber Nant ist, das der TFS entscheiden kann, gleich einen Build aufgrund der eingecheckten Files zu machen, sowie jeder Entwickler im Team mit einem Visual Studio Team Client den Server Build manuell vom Client aus starten kann.

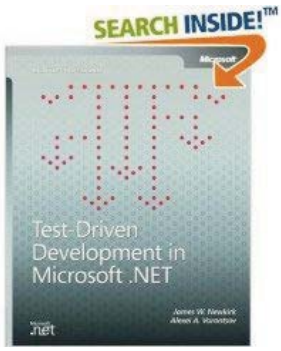
TFS benutzt den SQL Server als Datenablage. Neben dem Sourcecode werden hier noch ein Task/Ticket system und der Zusammenhang zwischen den CheckIn-Vorgängen abgelegt. Durch diese Datensammlung gibt es ein umfangreiches Reporting, das für die Entwickler im Studio als auch für andere Bereiche über Web- und Office-Integration zur Verfügung steht.

3 Tools und Links

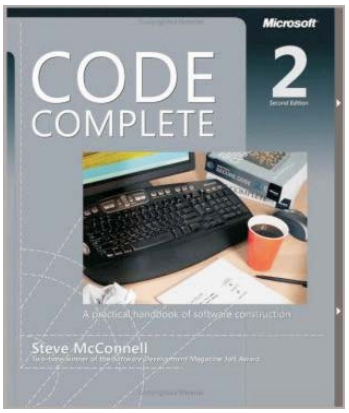
- http://www.nunit.org (Nunit)
- http://nant.sourceforge.net (Nant)
- http://nantcontrib.sourceforge.net/ (Nant Contrib)
- http://www.ndepend.com/ (NDepend)
- http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx (FxCop)
- http://www.jetbrains.com/resharper/ (Code Refactoring)

4 Literatur

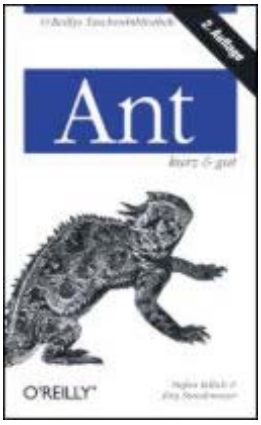
Pragmatic Unit Testing in C# with NUnit. (O'Reilly) ISBN 978-0-9776166-7-1



Test-Driven Development in Microsoft .NET (Microsoft Professional) ISBN 978-0735619487



Code Complete: A Practical Handbook of Software Construction (Microsoft Professional) ISBN 978-0735619678



Ant (Java Version) ISBN 978-3-89721-519-1