bbv
MAKING VISIONS WORK.

BOOKLET

AGILE ENTWICKLUNG VON EMBEDDED SOFTWARE

PROFITIEREN SIE VON UNSERER ERFAHRUNG!

Kontakt Schweiz
bbv Software Services AG
Blumenrain 10
6002 Luzern
Telefon: +41 41 429 01 11
E-Mail: info@bbv.ch

Kontakt Deutschland
bbv Software Services GmbH
Agnes-Pockels-Bogen 1
80992 München
Telefon: +49 89 452 438 30
E-Mail: info@bbv.eu

INHALT

1 EINFÜHRUNG

Agile Softwareentwicklung ist nicht neu: Das Agile Manifest wurde vor über zehn Jahren veröffentlicht, und gewisse Methoden wie XP sind noch älter. Viel wurde zu dem Thema gesagt und geschrieben, und dennoch bleibt die Umsetzung von agilen Prinzipien in der Entwicklung von Embedded Software eine Herausforderung. Dieses Booklet fasst unsere Erfahrung bei bbv Software Services zusammen und liefert dem Embedded-Entwickler Ideen und Tipps für den «agilen Alltag».

Wir konzentrieren uns vor allem auf Punkte, die wir als besondere Herausforderung in der Entwicklung von Embedded Software wahrnehmen. Themen der allgemeinen Softwareentwicklung streifen wir nur am Rande. Diese werden insbesondere von den Autoren des Agilen Manifests schon ausführlich abgedeckt. Die agile Entwicklung von Hardware ist ebenfalls ein spannendes Gebiet, das wir hier aber nicht behandeln. Wir beschränken uns in unseren Ausführungen auf das, was wir am besten können – die agile Entwicklung von Embedded Software.

2 EMBEDDED IST ANDERS

Embedded Software ist nicht gleich wie andere Software. Diesen Satz wird wohl jeder Embedded-Entwickler unterschreiben. Nach unserer Erfahrung hat aber auch jedes Embedded-Software-Projekt seine eigene Charakteristik: Eine Liftsteuerung mit 100 000 Installationen packen wir anders an als eine Konfigurationssoftware auf einem Pocket-PC. Die Software für ein medizinisches Analysegerät entwickeln wir anders als die Steuerung für einen Industrieroboter. Es gibt also nicht eine Art Embedded Software, sondern einen weiten Bereich.

Wenn man von der anderen Seite her fragt, was denn eine agile Vorgehensweise auszeichnet, gibt das Agile Manifest eine Antwort, die im englischen Original wie folgt lautet (Beck et al., 2001):

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Welche Herausforderungen ergeben sich nun bei der Umsetzung dieser Grundsätze in einem Embedded-Projekt?

3 PERSONEN UND INTERAKTIONEN
(INDIVIDUALS AND INTERACTIONS)

Hier gibt es die wenigsten Unterschiede zur herkömmlichen Softwareentwicklung. Sowohl die Personen als auch ihre Interaktionen bleiben sich weithin gleich – wenn auch gewisse Anzeichen darauf hindeuten, dass der Embedded-Entwickler einige Eigenarten hat.

3.1 TEAMWORK

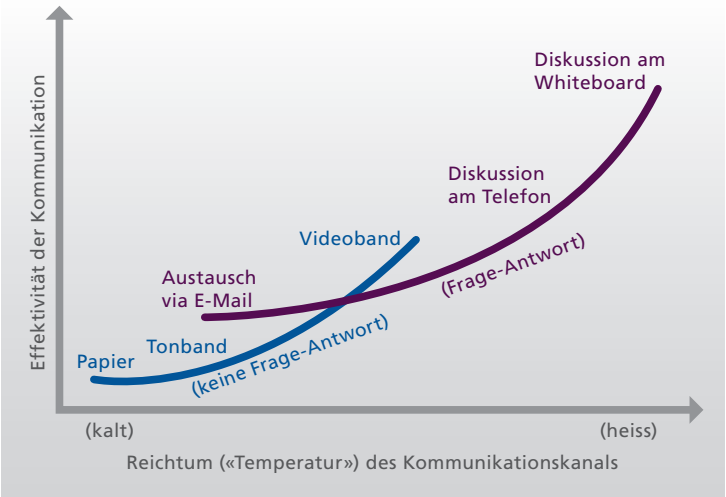
Auch als Demokratie gewohnter Schweizer ist es nicht einfach, ein agiles Team zu leiten, denn das Team führt sich weitgehend selbst. Wir brauchen das Vertrauen, dass jeder im Team das Optimum aus einer Iteration herausholen will, um das gemeinsame Ziel zu erreichen. Man muss sich daran gewöhnen, Regeln nicht vorzugeben. Das Team auferlegt sich aufgrund von Fehlern, die vorgekommen sind, in der Retrospektive die Regeln selber. Das Team wächst daran, und wir haben festgestellt, dass die Zufriedenheit und Motivation durch diese Selbstbestimmung sehr gross sind. Selbst wenn diese Selbstbestimmung nur bescheiden ist, da z. B. eine Grossfirma viele Bestimmungen hat, die man nicht von heute auf morgen ändern kann.

Die Zusammensetzung des Teams spielt ebenfalls eine wichtige Rolle. Es hat sich in unseren Projekten bewährt, dass jeder Entwickler im Team mehrere Funktionen abdeckt, damit das Know-how optimal im Team verteilt wird, so wie das Scrum oder XP vorschlagen. Dadurch können längere Ferienabwesenheiten und Abgänge optimal abgefedert werden. Eine Ausnahme gibt es: Wir haben bemerkt, dass die Integration und die Tests auf der Hardware die Entwickler sehr viel Zeit kosten und nicht gut planbar sind. Durch defekte oder geänderte Hardware verloren die Entwickler viel Zeit, bis sie ihre Implementation testen konnten. Es hat sich bei uns bewährt, einen Tester/Integrator ins Team zu nehmen, der für die Hardware verantwortlich ist. Dadurch können sich die Entwickler auf ihre eigentliche Aufgabe konzentrieren, und ihre Software wird schon früh von einer anderen Person getestet.

3.2 EFFEKTIVE KOMMUNIKATION

Unsere Erfahrung und diverse Studien deuten darauf hin, wie wichtig die effektive Kommunikation für den Erfolg von Softwareprojekten ist (Allen, 1984). Das Team muss Wissen aufbauen und teilen. Je näher die involvierten Personen zusammenarbeiten, desto besser funktioniert der Wissensaustausch und desto weniger Zeit wird zur Wissensbeschaffung verschwendet. Bei Embedded-Projekten sind neben der Softwareentwicklung viele weitere Fachbereiche involviert: von der Hardwareentwicklung über Marketing, Prozessdesign, Integration bis hin zur Produktion. Von der Kommunikation her gesehen wäre es das Beste, wenn alle gemeinsam in einem Büro sitzen würden. In der Praxis steht diesem Ideal vieles im Wege: Experten arbeiten in diversen Projekten mit, die Hardwareentwickler benötigen eine spezielle Infrastruktur, die vorhandenen Büros sind zu klein oder bereits belegt. Bei all diesen guten Gründen müssen wir aber aufpassen, dass wir es uns nicht zu einfach machen und den Status quo einfach akzeptieren. Vielfach ist es möglich, zumindest die Softwareentwickler in einem Raum arbeiten zu lassen und effiziente Kommunikationskanäle zu den anderen involvierten Personen aufzubauen.

Alistair Cockburn beschreibt die Effektivität verschiedener Arten der Kommunikation (Cockburn, 2001). Er unterscheidet dabei zwei Situationen: Bei der ersten stehen Frage und Antwort zur Verfügung, bei der zweiten nicht.



Innerhalb des Softwareteams gibt es viele bewährte Kommunikationstechniken, wie Diskussionen am Whiteboard, kurze Meetings vor dem Taskboard, Pair-Programming oder in abgeschwächter Form Pair-Check-in des Quellcodes. In einem Embedded-Projekt gehören natürlich Diskussionen über Nebenläufigkeiten, Interruptlast und Speicherbedarf dazu – diese Themen sind leicht anders. Die Form der Kommunikation ist aber gleich wie in der herkömmlichen Softwareentwicklung. Hier bieten agile Methoden wie Scrum, XP, Crystal oder andere viele Hilfsmittel.

3.3 SOLISTEN

Viele Methoden zielen auf das effiziente Arbeiten im Team. Es gibt aber auch sehr kleine Embedded-Projekte, in denen im Extremfall ein einzelner Softwareentwickler arbeitet. In solchen Situationen verlieren Praktiken wie Pair-Programming und Daily Scrum ihren Sinn. Eine bestehende Methode 1:1 umzusetzen, ist weder praktikabel noch sinnvoll. Es gibt jedoch Elemente von Scrum und Co., die auch in Einpersonenprojekten erfolgreich eingesetzt werden können. Konkret empfehlen wir den Einsatz folgender Praktiken:

- Arbeiten in Iterationen
- Planung mittels Product-Backlog zusammen mit dem Kunden
- Demonstration am Iterationsende
- Verbesserungen nur bei entsprechendem ROI
- Test Driven Development

Es lohnt sich zudem, die gegebene Situation zu hinterfragen: Kann das Team vergrössert werden, indem Personen anderer Fachbereiche hinzukommen? Könnten zwei kleine Projekte sinnvoll zusammengelegt werden? Als Solist zu arbeiten, kann sehr einsam sein – und beim Ausfall eines Solisten können sich für die Firma grosse Probleme ergeben.

4 FUNKTIONIERENDE PROGRAMME
(WORKING SOFTWARE)

4.1 LIMITIERUNGEN DURCH DIE HARDWARE

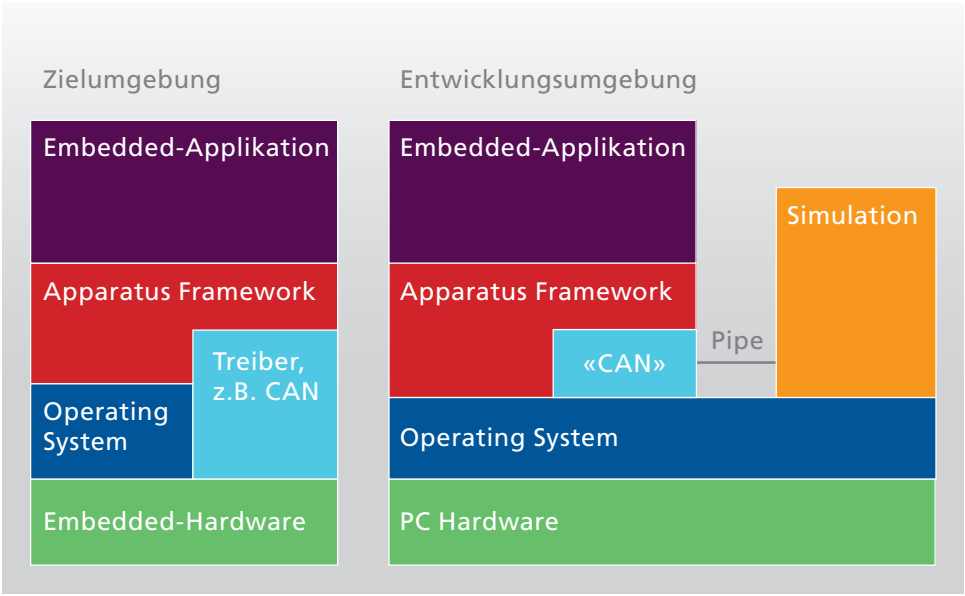
Fast in allen Embedded-Projekten benötigen wir spezifische Hardware. Es werden Schnittstellen wie CAN, digitale IO etc. eingesetzt, die auf Standard-PCs nicht vorhanden sind. Wir setzen Hardwarekomponenten ein, die kostengünstiger, sparsamer oder länger verfügbar sind als Standardkomponenten. Wir sind limitiert in Rechenleistung und Speicherplatz und müssen oft ohne Dateisystem, GUI und andere Funktionalität auskommen, die im Desktop garantiert sind.

Im Projektablauf sind wir oft auch auf andere Weise durch die Hardware limitiert:

- Finale Hardware ist noch nicht verfügbar
- Hardware ist fehlerhaft
- Einzelne Features der Hardware fehlen noch
- Mehrere Softwareentwickler müssen sich eine Hardware teilen

Die Herausforderung ist hier, ein funktionierendes Programm bereitzustellen, wenn die zugrunde liegende Hardware fehlt oder Mängel hat. Die offensichtliche Lösung dazu ist es, auf eine Hardware auszuweichen, die funktioniert. Das könnte ein Evaluation-Board sein. In unserer Erfahrung ist es aber langfristig sinnvoller, die Software auf dem Entwicklungsrechner lauffähig zu machen. Dies hat diverse Vorteile: Wir haben von Beginn an eine funktionstüchtige Plattform zur Verfügung, jeder Entwickler hat seine eigene Testumgebung, die Entwicklungszyklen sind kürzer und die Entwicklungswerkzeuge oft komfortabler. Manche Autoren bezeichnen es sogar als Vorteil, wenn die Hardware erst spät verfügbar ist – das zwinge die Entwickler dazu, die Probleme abstrakter anzusehen und die Hardware sauber zu abstrahieren (Grenning, 2004).

Das Apparatus Framework von bbv bietet die Grundlage, um Embedded Software auf dem Entwicklungssystem zum Laufen zu bringen. Aufrufe des Betriebssystems sind abstrahiert, ebenso der Zugriff auf Schnittstellen. Mit einer Zeile Code kann zum Beispiel die CAN-Kommunikation auf eine Pipe oder einen Socket umgeleitet werden. Ein Simulationsprogramm auf der anderen Seite der Pipe oder des Socket nimmt die Pakete entgegen und stimuliert die Embedded-Applikation wie nötig.

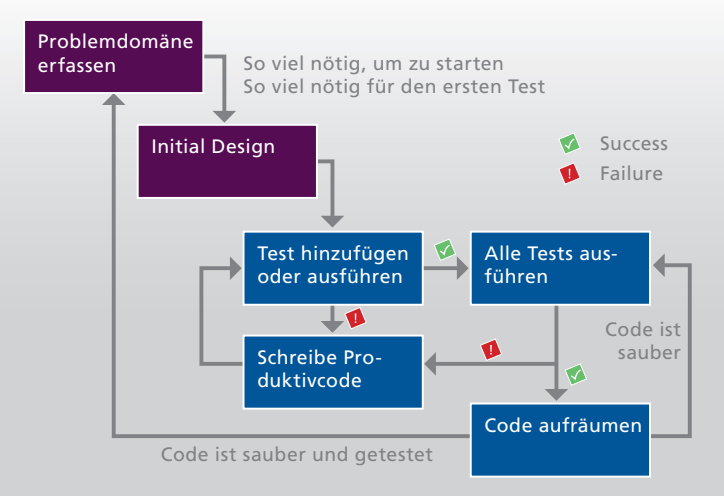


4.2 VERTRAUEN DURCH ITERATIVE TESTS

Um früh im Projektverlauf Risiken und Verzögerungen zu vermeiden, ist es wichtig, das Testen und die Integration in den Entwicklungszyklus einzubauen. Test Driven Development (TDD) ist eine Technik, die den automatisierten Unit-Test, den Acceptance-Test und den Code, der die Tests erfüllt, ins Zentrum stellt. Der Test wird dabei vor dem Code entwickelt, der durch die Tests durchlaufen wird. TDD als Praktik kommt nicht aus dem Embedded-Umfeld, doch die Prinzipien und Techniken erhöhen mit wenigen Anpassungen auch in der Embedded-Entwicklung die Qualität der Software.

TDD als Entwicklungspraxis beeinflusst das Design der Software erheblich. Die Embedded Software muss so aufgebaut werden, dass die Komponenten zu jeder Zeit automatisiert getestet werden können. Doch die Entwicklungsumgebung für ein Embedded-System unterscheidet sich in der Regel wesentlich von der Zielplattform. Die Zielplattform ist oft limitiert in der Stückzahl, teuer in der Fertigung oder erst spät im Projektverlauf als Prototyp erhältlich. Falls die Zielplattform vorhanden ist, kostet der Download der Software auf die Plattform Zeit und das Debuggen ist oft nur unter erschwerten Bedingungen möglich, was die Produktivität eines Teams erheblich einschränkt.

Der TDD-Zyklus soll diesem Problem entgegenwirken. Eine leicht modifizierte schematische Darstellung des TDD-Zyklus von Kent Beck beschreibt ihn folgendermassen (Beck, Test Driven Development: By Example, 2002):



Der Zyklus darf nur wenige Minuten dauern. Alle paar Minuten erhält der Entwickler ein direktes Feedback, ob der Produktivcode wirklich die Anforderungen erfüllt. Alle Tests sind automatisiert, und nach jeder Änderung werden die Tests erneut ausgeführt.

Die Übertragung des TDD-Entwicklungszyklus auf Embedded Software kann mit wenigen Modifikationen vorgenommen werden. Folgende Frage ist zentral: Wann werden die Tests ausgeführt? Der TDD-Zyklus schreibt vor, dass die Tests so oft wie möglich ausgeführt werden sollen. Bei Embedded Software muss aber auch die Verfügbarkeit der Hardware, der Prototypen-Hardware mit limitierter Funktionalität oder der Endplattform berücksichtigt werden. Ein Embedded-Entwickler, der nach TDD arbeitet, schreibt Code, der sowohl auf dem Entwicklungssystem als auch auf der Zielplattform ausgeführt werden kann. Sehr oft wird gegen TDD im Embedded-

Umfeld argumentiert, dass man reale Hardware zum Testen des Codes verwenden muss. Hardwareabhängigkeiten sind nicht vermeidbar, der Code kann aber mittels etablierten Designpatterns relativ einfach von der Hardware entkoppelt werden. Es gilt das Dogma:

Nicht jede Komponente oder jeder Codeteil ist auf allen Plattformen ausführbar. Aber ein grosser Teil kann mit einfachen Mitteln von der Zielplattform entkoppelt werden, sodass ein genügend grosses Vertrauen in die Korrektheit und Stabilität der Software aufgebaut werden kann.

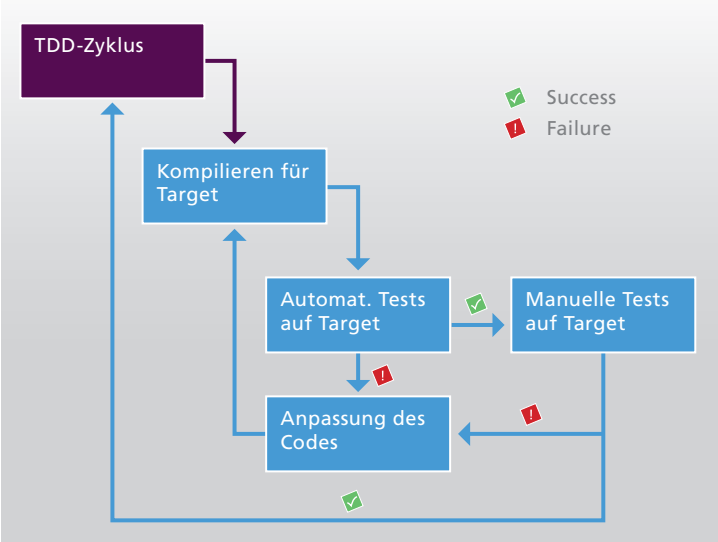
In der frühen Projektphase, wenn die Zielplattform noch nicht erhältlich ist, können die Tests auf dem Entwicklungssystem ausgeführt werden, in dem die Abhängigkeiten auf die Hardware durch Schnittstellen und Stellvertreterobjekte (Mocks) simuliert werden. Dazu muss die zu realisierende Software entsprechend geschichtet werden.

Sobald der Prototyp verfügbar ist, werden die Tests zwar grundsätzlich immer noch auf dem Entwicklungssystem ausgeführt, jedoch periodisch auf dem Prototyp automatisiert wiederholt. Dies stellt sicher, dass der generierte Code auch auf der Zielplattform dasselbe Verhalten aufweist und korrekt für die Zielplattform kompi-

liert werden kann. Die Tests sollten nach jeder Änderung oder nach dem Check-in in die Versionsverwaltung ausgeführt werden.

Wenn die Zielplattform erhältlich ist, werden Tests für jenen Code entwickelt, der direkt auf die Hardware zugreift. Automatisierte Tests sind in diesem Bereich komplexer zu entwickeln, da vielfach manuelle Verifikation oder externe Instrumentation nötig sind. Wichtig ist, dass solche Tests einfach auszuführen sind, denn sonst werden sie vom Entwickler nicht genutzt. Schlussendlich erhöhen End-zu-End-Tests das Vertrauen in die Software, da durch die «gemockte» Hardware das System gezielt in Zustände gebracht werden kann, um auch seltenere Szenarien zu testen.

Dies führt uns zu folgendem TDD-Zyklus für Embedded-Systeme:



Mit dem angepassten TDD-Zyklus für Embedded-Plattformen kann die Stabilität gesichert und die langfristige Wartbarkeit der Plattform gewährleistet werden. Automatische Tests für das Target sichern bei regelmässigen Releases, dass bereits abgenommene Features der Plattform auch nach Anpassungen das gewünschte Verhalten aufweisen.

4.3 METRIKEN IN SCRUM

Die bekannteste und wichtigste Metrik für das Team ist sicher der Sprint-Burndown-Chart, ein wesentlicher Bestandteil von Scrum. Wird Scrum neu eingeführt, reicht aber dieser Chart meistens nicht aus. Das Team und das Management brauchen weitere Metriken, um zum Beispiel zu überprüfen, ob TDD umgesetzt wird oder ob die Tests unter der Sprint-Velocity leiden. Um einen Anhaltspunkt über die Testabdeckung zu haben, hat sich bei uns Code-Coverage bewährt. Dabei spielt es nach unserer Meinung eine untergeordnete Rolle, ob Line-, Statement- oder Branch-Coverage verwendet wird. Denn die Qualität der Tests soll durch Reviews überprüft werden und nicht durch Metriken. Das Vertrauen zum Team wird so gestärkt und auch der Entwickler sieht, wo er Lücken in seinen Tests hat.

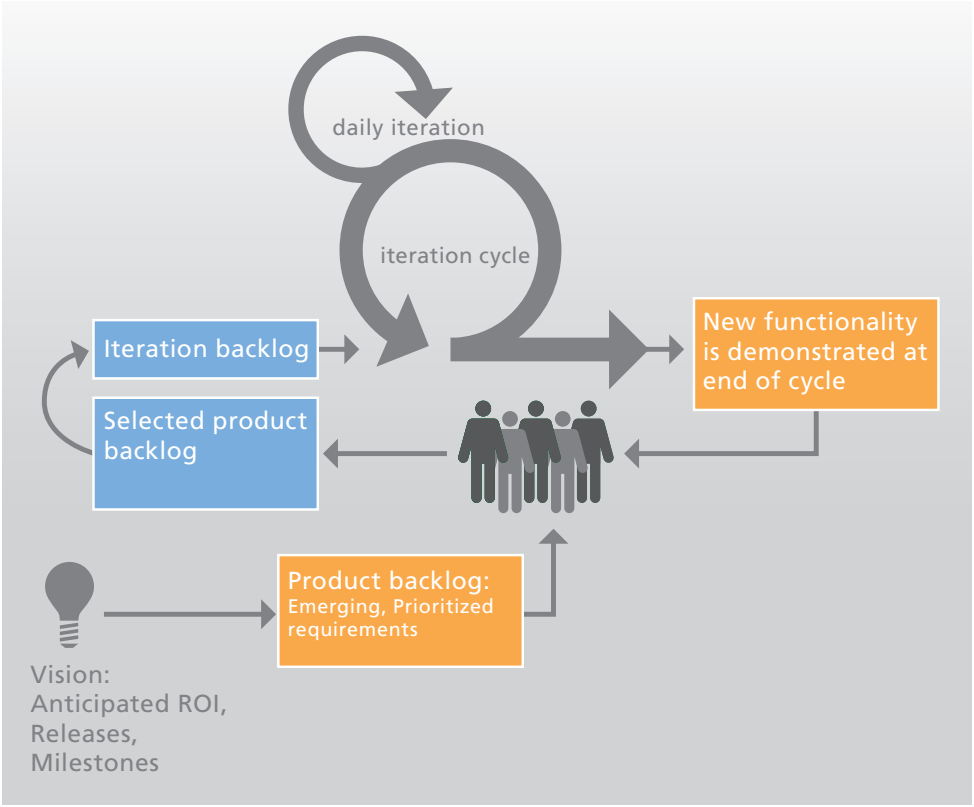
Eine weitere Metrik, die einem den Übergang zu Scrum erleichtert, ist das Zählen der geschriebenen Codezeilen nach einem Sprint. Mit dieser Metrik wird verhindert, dass man in das Wasserfall-Modell zurückfällt und in einem Sprint Analyse/Design macht und im nächsten Sprint implementiert.

Je mehr ein Team eingespielt ist und die agilen Methoden gelebt werden, desto weniger wichtig werden die Metriken. Unabhängig davon, welche und wie viele Metriken angewendet werden: Sie dürfen das Team nicht belasten, sondern müssen automatisch generiert werden.

5 STETIGE ZUSAMMENARBEIT MIT DEM KUNDEN
(CUSTOMER COLLABORATION)

Es gibt verschiedene Studien, die belegen, dass Fehler in den Anforderungen viel höhere Kosten mit sich bringen als z. B. Fehler in der Implementierung (Boehm, 1981). Eine stetige Zusammenarbeit mit dem Kunden hilft, Fehler und Missverständnisse in den Anforderungen möglichst früh aufzudecken.

Wir haben sehr gute Erfahrungen mit Sprint-Reviews nach Scrum gemacht, bei denen die implementierten Features dem Kunden vorgeführt und von diesem formal akzeptiert werden. So erhalten die Entwickler wertvolles Feedback, der Kunde ist laufend über den Stand des Projektes informiert, und akzeptierte Features können als «done» abgehakt werden. Auch hier ist es hilfreich, möglichst schnell eine lauffähige Umgebung bereitzustellen. Viele wichtige Aspekte können bereits auf einer simulierten Umgebung angeschaut und akzeptiert werden. Der Kunde hat in der Regel kein Problem damit, dass Teile der Umgebung erst simuliert sind – Feedback zu Missverständnissen oder Änderungen in den Anforderungen erfolgt im gleichen Mass. Zu einem späteren Zeitpunkt kann eine formale Abnahme auf der realen Hardware erfolgen.



Manche agile Methoden gehen noch weiter: In Scrum bestimmt der Kunde die Priorität der Features und somit die Reihenfolge ihrer Implementierung (Kniberg, 2007). XP fordert sogar einen Kunden, der ständig vor Ort ist (Beck, Extreme Programming Explained, 1999). Dadurch sollen Fragen zu den Anforderungen ohne Verzögerung geklärt werden können. Wie weit man hier gehen kann und will, bleibt offen. Wichtig ist zu erkennen, dass von einer guten und stetigen Zusammenarbeit sowohl Entwicklung wie Kunde profitieren.

6 MUT UND OFFENHEIT FÜR ÄNDERUNGEN
(RESPONDING TO CHANGE)

Man kann über die Entwickler von Embedded Software lästern, wie man will, niemand wird ihnen vorwerfen, ein wilder Haufen draufgängerischer Haudegen zu sein, die mal schnell die Kernlogik ihrer Applikation umschreiben. Die Entwickler sind im Gegenteil enorm vorsichtig: Jede Änderung wird genau geprüft, abgewogen und auch oft als «zu riskant» taxiert. Und das mit gutem Grund: Mit einer Modifikation können sich die funktionalen oder zeitlichen Abläufe so ändern, dass ein Fehlverhalten entsteht.

So berechtigt diese Einstellung sein mag, so verheerend ist sie, wenn wir agil auf Änderungen reagieren wollen. Wenn jede Modifikation «zu riskant» ist, verlieren wir jeglichen Handlungsspielraum. Unser Ziel soll es sein, Änderungen rasch, aber kontrolliert einfliessen zu lassen, immer mit der Gewissheit, dass wir nichts kaputt gemacht haben. Dazu benötigen die Softwareentwickler jede Hilfe, die sie bekommen können.

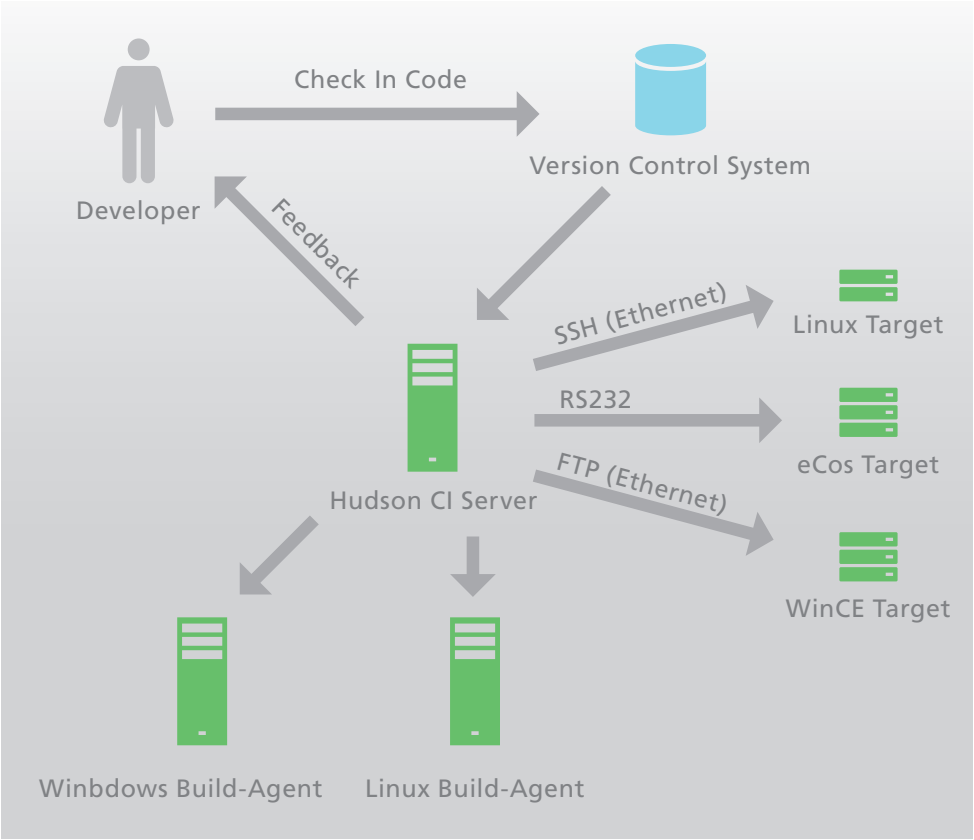
6.1 TOOLS IM AGILEN WERKZEUGKOFFER

Das bei Weitem wichtigste Hilfsmittel ist die Versionskontrolle. Sie beantwortet so essenzielle Fragen wie: «Ist die Änderung XY im Release 1.1 schon vorhanden?» oder: «Wann wurde Bug YZ eingeführt?». Auch kann damit im Notfall auf eine frühere, funktionierende Version zurückgegriffen werden. Wer je mit einem Tool zur Versionskontrolle gearbeitet hat, wird sich kaum vorstellen können, ohne es auszukommen. Wir haben sehr gute Erfahrungen mit Subversion gemacht, aber die Anforderungen variieren, und viele andere Tools sind ebenso brauchbar.

Wie bereits im Abschnitt «Vertrauen durch iterative Tests» beschrieben, sind automatisierte Tests ein weiteres wichtiges Puzzlestück in der agilen Entwicklung. Wenn die Tests einwandfrei durchlaufen, haben wir die Gewissheit, im abgedeckten Funktionsumfang keine Fehler eingeführt zu haben. Es gibt eine ganze Reihe von Unit-Test-Frameworks für C und C++, welche auch für Embedded Software geeignet sind. Konkrete Erfahrungen haben wir mit Boost-Test, Google-Test, CppUnit und dem Testing-Framework im Apparatus Framework. Die Tests lassen wir in der Entwicklungsumgebung sowie auf dem Target laufen – wenn das vorhandene Memory dies erlaubt.

Beim Testen kommt man oft in die Situation, dass man eine Unit von ihrer Umgebung isolieren möchte, sei es, um Fehlerfälle zu testen oder Folgefehler zu vermeiden. In diesem Zusammenhang haben sich Mock-Objekte bewährt: Sie ersetzen Clients der Unit Under Test (UUT), überprüfen die erfolgten Aufrufe und geben gewünschte Daten an die UUT zurück. In Sprachen wie Java oder C#, welche Reflection unterstützen, sind sogenannte Mocking-Frameworks bereits etabliert. In C++ ist etwas mehr Handarbeit gefordert, doch bieten auch hier Frameworks wie Google-Mock oder Hippo-Mocks Unterstützung.

Aufbauend auf den oben erwähnten Tools erstellt ein Continuous-Integration (CI)-Server nach jedem Commit einen neuen Build, installiert die Applikation, lässt die Tests laufen und benachrichtigt die beteiligten Entwickler im Falle eines Problems. Es gibt diverse CI-Produkte, z. B. CruiseControl, TeamCity oder Hudson. Bei unseren Recherchen haben wir kein System gefunden, das den Download auf Embedded Targets von Haus aus unterstützt. Somit ist immer etwas Handarbeit gefragt, wenn wir die Tests nicht nur auf der Entwicklungsumgebung laufen lassen wollen. Unser Apparatus Framework lassen wir von einem Hudson-CI-Server integrieren und auf diversen Targets testen.



Weitere Tools können die Entwickler bei der statischen Codeanalyse, beim Refactoring oder beim Messen der Code-Coverage unterstützen. Die Kombination dieser Tools gibt dem Entwickler eine gute Gewissheit, dass seine Änderungen keine Fehler enthalten. Natürlich liegt die Gewissheit nie bei 100% – gerade wenn das Laufzeitverhalten kritisch ist, müssen weitere Analysen und Tests vorgenommen werden.

7 FAZIT

Die Hürde bei der Umsetzung agiler Grundsätze ist in Embedded-Projekten leicht höher als bei der allgemeinen Softwareentwicklung. Gerade für die Bereiche Unit-Testing und Continuous Integration können keine Standardlösungen verwendet werden. Es bleibt immer eine Herausforderung, mit den Limitierungen der Hardware fertig zu werden. Dennoch haben wir insbesondere mit Scrum gute Erfahrungen gemacht: Die Motivation im Team steigt, Fortschritte und Qualität werden sichtbar, auf Änderungen kann schneller reagiert werden.

Wir hoffen, in diesem Booklet einige brauchbare Tipps und Hinweise für die agile Entwicklung von Embedded Software zu präsentieren. Da jedes Embedded-Projekt anders ist, wird die Umsetzung immer wieder anders aussehen. Das ist eine grosse Herausforderung, die aber die Arbeit im Embedded-Umfeld auch so spannend macht.

8 QUELLENVERZEICHNIS

Allen, T. (1984). Managing the Flow of Technology. MIT Press.

Beck et al., K. (2001). Manifesto for Agile Software Development. Retrieved from Agile Manifesto: http://agilemanifesto.org/

Beck, K. (1999). Extreme Programming Explained. Addison-Wesley.
Beck, K. (2002). Test Driven Development: By Example. Addison-Wesley Professional.

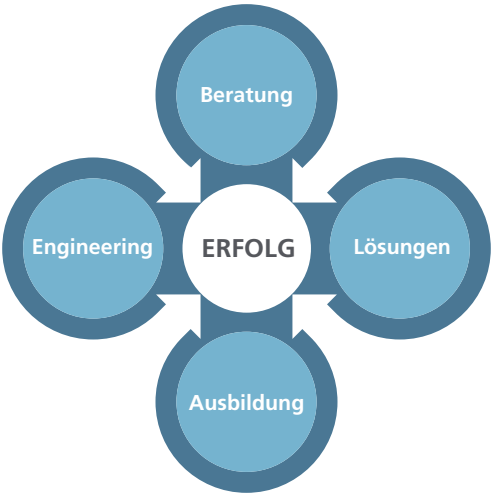
Boehm, B. (1981). Software Engineering Economics. Prentice-Hall.
Cockburn, A. (2001). Agile Software Development. Addison-Wesley Professional.

Grenning, J. W. (2004, March). Agile Embedded Software Development. Retrieved from Renaissance Software Consulting: http://www.renaissancesoftware.net

Kniberg, H. (2007). Scrum and XP from the Trenches. Lulu PR.

bbv
MAKING VISIONS WORK.

bbv Software Services AG ist ein Schweizer Software- und Beratungsunternehmen, das Kunden bei der Realisierung ihrer Visionen und Projekte unterstützt. Wir entwickeln individuelle Softwarelösungen und begleiten Kunden mit fundierter Beratung, erstklassigem Software Engineering und langjähriger Branchenerfahrung auf dem Weg zur erfolgreichen Lösung.



Unsere Booklets und vieles mehr finden Sie unter
www.bbv.ch/publikationen

MAKING VISIONS WORK.

www.bbv.ch · info@bbv.ch