Agilität hautnah

Die Woche eines agilen Entwicklers – Teil 2: Donnerstag bis Freitag

Alex Bepple, Henning Wolf

Agile Entwicklung bedeutet Entwickeln zwischen Testen, inkrementellem Design, Architekturdiskussionen, Aufwandsschätzungen, Fertigstellungsprognosen und Abstimmungen im Team. Der Artikel beschreibt aus der Sicht der Entwickler, welche Herausforderungen auf sie zukommen: Wie agil entwickelt wird; wie sich das Design ergibt; wer, wann und wo und wie über Architekturen diskutiert; wie die Qualitätssicherung erfolgt und sich alles immer noch mehr zum Guten wenden soll. Was haben Entwickler davon? Wirklich weniger Stress und Hektik? Steht nicht ständig der nächste Termin an?

Im ersten Teil in Heft 6/2008 ging es in dem fiktiven Tagebuch eines agilen Entwicklers um das Projektsetting, die Arbeitsumgebung, agile Architekturen und Architekturdiskussionen sowie die Planung und die Aufwandsabschätzung. In diesem abschließenden Teil betrachten wir Testerfahrungen, Code-Ownership und Retrospektiven.

Donnerstag: Testerfahrungen

Marcus und ich machen uns nach dem Standup an eine neue Aufgabe. Erst schauen wir uns an, wie die gleiche Aufgabe mit einem sehr ähnlichen Werkzeug unserer Anwendung gelöst wurde. Wir finden schwer verständlichen Legacy-Code vor. Der gewählte Lösungsansatz erscheint uns außerdem zu kompliziert. Schnell ist uns klar, wie es unserer Meinung nach besser ginge. Genau so schnell ist der erste Testfall skizziert. Doch nun kommen wir ins Stocken: Die getestete Klasse ist ein großer Monolith. Sie erzeugt sich viele ihrer Abhängigkeiten selbst. Ohne Veränderungen an der Klasse können wir keinen Unit-Test schreiben. Wir beschließen, den Test erst einmal ruhen zu lassen und die Klasse mittels rein automatischer Refactorings für Tests zugänglich zu machen. So führen wir zwei Setter ein und lagern einen Aufruf in eine eigene Methode aus, die wir in einer Unterklasse überschreiben können. Die Tests laufen durch, sodass Marcus und ich hier unseren ersten Commit vornehmen.

Nun programmieren wir schrittweise unseren Test. Wir formulieren unsere Testanforderung: Wir wollen, dass unsere Klasse einen ihrer Kollaborateure in einer ganz bestimmten Weise aufruft. Dann kommentieren wir diese Anforderung aus und beginnen, die Fixture (das Test-Set-Up) aufzubauen. Hierfür nutzen wir die eben erst erschaffenen Methoden. Wir schleusen einen Stub-Kollaborateur ein, der uns nicht weiter kümmert, als dass er bei bestimmten Aufrufen für den Test passende Werte zurückgibt. Über den anderen Setter schleusen wir ein Mock-Objekt ein, auf das unsere Testanforderung zielt. Außerdem bilden wir speziell für unseren Test eine Unterklasse, in der wir einige Methoden leer überschreiben. So gelingt es uns, Zugriffe auf einen weiteren Kollaborateur abzuklemmen, ohne ein umfangreiches Objektnetz aufzubauen. Schließlich kommentieren wir unsere Testanforderung ein. Der Test läuft rot. Wir brauchen nur wenig Code, um ihn zufrieden zu stellen. Es ist Zeit für den zweiten Commit.

Und für das Mittagessen. Zuvor aber starten wir unsere umfangreichste Test-Suite. Zehn Minuten braucht sie, um durchzulaufen. Zu lang, um vor jedem Commit ausgeführt zu werden, schnell genug für Pausen. Unser Sicherheitsnetz besteht aus drei Test-Suiten. Unsere Unit-Tests laufen in 19 Sekunden durch. Wir führen sie immer wieder zwischendurch aus. Die nächstgrößere Suite umfasst auch die GUI-Integrationstests. Die zwei Minuten, die sie braucht, nehmen wir uns vor jedem Commit. Darüber hinaus gibt es nur noch die Test-before-Break-Suite. Sie enthält zusätzlich eine Vielzahl von Integrations- und Fit-Tests.

Wenn sich im Rahmen einer Entwicklungsaufgabe die Gelegenheit bietet, überarbeiten wir unsere Tests. Denn wir schreiben sie heute anders als früher: mehr Testfälle, die dafür kleiner sind, und mehr Testklassen mit kleineren Fixtures, die sich manchmal ähneln. Die Tests, die dabei heraus kommen, sind vor allem eins: besser verständlich – und darum besser zu warten und zu ergänzen. Außerdem setzen wir heute viel mehr auf Mock-Objekte. So werden die Tests auch schneller. In unregelmäßigen Abständen überprüfen wir unsere Testabdeckung und die Testlaufzeiten. Der Trend ist seit Monaten der gleiche: Die Tests sichern in weniger Zeit einen größeren Teil des Codes ab.

In ähnlich kleinen Inkrementen wie vor dem Mittagessen entwickeln Marcus und ich am Nachmittag weiter. Bei einem manuellen Akzeptanztest fällt uns ein Problem auf, ohne dass die Ursache sofort klar ist. Schnell ist ein weiterer Testfall geschrieben; wir können das vorhandene Set-Up ohne Änderungen nutzen. Umgehend finden wir auch den Fehler.

Bei der nächsten Karte fällt ein anderes Problem auf. Da ist die Ursache sofort klar. Wir fügen schnell eine Zeile Code hinzu, starten die Anwendung und verifizieren unsere Korrektur. Natürlich sind wir versucht, den Code gleich einzuchecken. Jedoch rollen wir ihn zurück und schreiben einen Testfall, der den Fehler aufdeckt. Dabei fällt uns auf, dass der Aufrufer der kaputten Methode eine Berechnung durchführt, die nicht in seinen Zuständigkeitsbereich fällt. Wir formulieren unseren

Testfall im Sinne von „Tell, don't ask", und passen den Aufrufer gleich an. Nun können wir committen.

Spät am Nachmittag machen wir uns an eine neue Story. Zu Beginn schauen wir uns im Code um: Wie sind die Abhängigkeiten? An welchen Stellen müssen wir ändern? Im Gespräch verfestigt sich unser Eindruck, dass an einer Klasse ein umfangreiches Refactoring angebracht ist. Der vorhandene Code ist unzureichend nach Verantwortlichkeiten gegliedert: Er ist kaum verständlich und schlecht erweiterbar.

Zu Beginn der Release-Entwicklung haben wir alle umzusetzenden Anforderungen in Entwicklungsfeatures aufgespalten. Teilweise haben wir uns dabei auch kurz den betroffenen Code angeschaut. In diesem Fall offenbar nicht – oder nicht gründlich genug. Das Refactoring, das wir vorhaben, würde sich im Rahmen dieser Anforderung allein nicht lohnen. Das Team pflegt den Code aber unter der Annahme, dass die Anwendung weiter entwickelt wird. Tatsächlich hat der Kunde schon Änderungs- und Erweiterungswünsche für die nächsten zwei Jahre eingeplant.

Wir fügen der Anforderung daher zwei weitere Entwicklungsaufgaben hinzu: Refactoring, Teile eins und zwei. Früher waren wir nachgiebiger: Haben auf Refactorings verzichtet, um die Entwicklungsgeschwindigkeit zu erhöhen. Das bezahlen wir heute oft zweimal mit einer niedrigeren Geschwindigkeit: einmal, weil der Code schwer zu ändern ist, und noch einmal, weil wir Zeit investieren, um seine Qualität zu erhöhen, indem wir die Testabdeckung verbessern und refaktorisieren.

Freitag: Code-Ownership und Mini-Retrospektive

Am Mittwoch vor ein paar Wochen erläuterte Herr A, dass die Arbeit der Techniker vor Ort immer wieder stocken würde, weil sie die Kennzeichen der falsch geparkten Fahrzeuge auf Papier aufschreiben müssten. Unsere Software unterstützt sie dabei bisher nicht. Der Kunde hat zwar schon eine konkrete Idee für die Umsetzung. Wir haben allerdings noch ein paar Fragen. In Zusammenarbeit mit den Fachanwendern klärt Hans heute Vormittag die Abläufe genauer. Außerdem sind gesetzliche Regelungen noch nicht ausreichend berücksichtigt.

Immer wenn unser Kunde weitere Arbeitsprozesse elektronisch unterstützen will, klären wir die Anforderungen in Zusammenarbeit mit ihm ab. Den Großteil der Arbeit übernimmt bei uns Hans; er nimmt die Rolle des Domain-Designers wahr. Er bringt neue Anforderungen zur Angebotsreife. Dafür spricht er mit Herrn A und repräsentativen Anwendern und hält die Anforderungen in Zusammenarbeit mit ihnen schriftlich fest. Manchmal müssen sie dabei einen Schritt zurück gehen und die Anforderungen, mit denen Herr A an uns herangetreten ist, kritisch hinterfragen: Welche Bedürfnisse haben die Anwender in der vorgestellten Anwendungssituation? Durch welche Abläufe und Interaktionsmöglichkeiten können wir sie unterstützen? Indem wir kritisch nachfragen, erhöhen wir den Nutzen des Produkts für unseren Kunden und vermeiden weitestgehend Fehlleistungen und Verschwendung.

Bevor die Anforderungsbeschreibung zur Grundlage eines Festpreisangebots wird, holt Hans das Feedback des Teams ein. Schließlich liegt hohe Qualität allen Team-Mitgliedern am Herzen.

Während sich Hans mit den Fachanwendern trifft, beseitigen Torben und ich Fehler in der Produktivsoftware. Der hauseigene IT-Dienstleister unseres Kunden übernimmt den direkten Anwender-Support und einfache Schadensbehebungen. Unsere Fehler von früher müssen wir allerdings selbst ausbügeln. Wir schauen im gemeinsam mit dem Kunden genutzten Mantis-

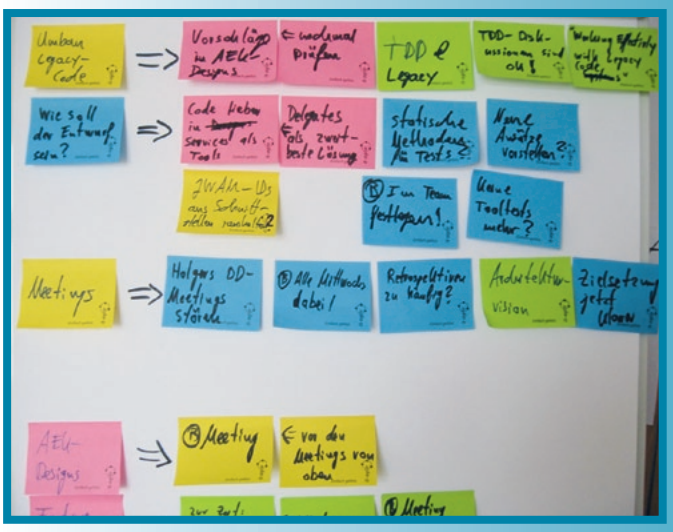

Abb. 1: Retrospektiven-Notizen

System (unserem Bugtracker) nach, welche Fehler noch nicht beseitigt sind. Es gibt keine Blocker, dafür aber zwei wichtige Fehler. Da wir nicht einschätzen können, welcher wichtiger ist, gehen wir zu Herrn A, um die Fehler priorisieren zu lassen. Derzeit verwenden wir Woche für Woche einen Personentag für die Betriebsführung. Ausnahmen machen wir nur, wenn Fehler auftauchen, die die Arbeit von Anwendern lahm legen. So haben wir eine gute Planungssicherheit und verbessern kontinuierlich die Qualität des laufenden Systems.

Für die Fehlerbehebung müssen wir Code ändern, den wir nicht geschrieben haben. Das ist bei uns kein Problem. Denn wir haben keine Komponenten- oder Package-Fürsten. Jeder darf jede Zeile Code ändern. Ärger haben wir deswegen nicht. Vielmehr ein breit verteiltes Wissen über den Code und keine Hindernisse im Arbeitsfluss – wir müssen uns für Änderungen nicht erst aufwändig mit anderen abstimmen, und wenn einer im Urlaub ist, ist die Entwicklung an keiner Ecke blockiert.

Allerdings gibt es diese Vorteile nicht gratis. Regelmäßig arbeiten wir daran, dass es keinen Ärger wegen unerwünschter Änderungen gibt. Dafür bilden und überarbeiten wir Absprachen zu Code-Konventionen und Entwurfsmustern. Ein Beispiel: Vor einigen Monaten haben wir uns entschlossen, neue Wege beim Testen zu gehen. Früher waren unsere Testfälle groß, da wir oft verschiedene Szenarien innerhalb desselben Testfalls überprüften. Im Resultat waren die Testklassen schwer nachvollziehbar und kaum wartbar. Außerdem gab es Abhängigkeiten zwischen den Tests. Nun schreiben wir kleinere, fokussierte Testfälle. Dabei orientieren wir uns an Vorschlägen der verhaltensorientierten Entwicklung. Bevor wir diese Absprache getroffen hatten, gab es immer wieder Diskussionen im Team, wie ein Testfall zu formulieren und wo er einzuordnen wäre. Dann haben wir uns ausgesprochen, die verschiedenen Vorstellungen ausgetauscht und abgewogen – und schließlich eine gemeinsame Entscheidung getroffen. Nach dieser Festlegung waren die Diskussionen um Testformulierung weitgehend verstummt. Seit wenigen Wochen führen wir wieder Diskussionen: Welche Vorteile hat es, wenn ein Testfall alle Auswirkungen einer Methode überprüft? Welche Vorzüge hat es, eine Auswirkung pro Testfall zu kontrollieren? Um wieder zu einer gemeinsamen Vorgehensweise zu finden, werden wir demnächst gemeinsam Test-Code reviewen, diskutieren und uns schließlich auf einen gemeinsamen Weg verständigen.

Seltener stellen wir auch mal unseren Architekturstil in Frage. Einen Anlass dafür lieferte z. B. vor kurzem ein Entwickler unseres Teams. Er hat eine Idee, wie wir unsere GUI-Architektur modernisieren können. Sein Vorschlag weicht erheblich von dem bisher praktizierten Stil ab, und nicht alle sind überzeugt, dass der Vorschlag besser ist. Fraglich ist auch, ob wir die gesamte Anwendung dem neuen Vorschlag entsprechend umbauen können, und wenn nicht, ob wir eine Mehrzahl von Architekturstilen nebeneinander für tragbar halten. Die gemeinsame Diskussion und Entscheidung stehen noch aus. Bis dahin brechen immer wieder Wortwechsel zu dem Thema aus.

Am Nachmittag beschließen wir die Arbeitswoche mit einer Retrospektive (s. Abb. 1). Alle zwei bis drei Wochen führen wir eine solche durch. Dann tragen wir zusammen, was uns bei der Arbeit behindert, beschäftigt und für Diskussionsstoff sorgt. Eine gute Hand voll Probleme, Fragen und Vorschläge sind es meist. Alle können wir in den anderthalb Stunden, die wir uns nehmen, gar nicht besprechen. Das ist auch nicht nötig, denn die nächste Retrospektive kommt bestimmt. Also priorisieren wir die Themen. Regelmäßig gelingt es uns, die wichtigsten beiden zu analysieren und Antworten auf die Herausforderungen zu finden. In den letzten beiden Iterationen haben mehrfach auftretende Probleme bei der Integration eines Drittsystems uns Zeit gekostet und auf die Stimmung gedrückt. Außerdem sind unsere Standups in letzter Zeit mehrmals länger als nötig ausgefallen. Um unsere Diskussion zu kanalisieren, bitten wir einen agil-erfahrenen Kollegen aus einem Schwesterprojekt, die Retrospektive zu moderieren. Mit seiner Hilfe finden wir zum Beispiel heraus, dass ein Grund für die Reibungsverluste bei den Integrationsversuchen viele manuelle Akzeptanztests an den Schnittstellen mit dem Drittsystem sind. Wir führen sie durch, weil wir die Qualität der Drittkomponente oft als unzureichend empfinden. Eine Ursache dafür wiederum könnte sein, dass der Hersteller jenes Systems bisher keine Akzeptanztests durchführt. Hier setzt dann einer der Lösungsvorschläge an: Wir werden ihm nahe legen, Akzeptanztests mit unserer Anwendung durchzuführen. Am Ende der anderthalb Stunden haben wir unseren Prozess ein bisschen verbessert: Wir haben die Ursachen unserer Probleme verstanden und Maßnahmen eingeplant, um sie zu lösen. Guter Dinge starten wir nun ins Wochenende.

Fazit

Ich bin Entwickler. Meine Woche besteht nach wie vor zum größten Teil aus Entwickeln. Daneben aber beschreibe ich auch Anforderungen, verändere den Systementwurf, teste die Software und spreche ständig mit dem Kunden. Das alles natürlich im Team.

Meine Stimme zählt für alle Entscheidungen, die wir im Team fällen. Nicht immer gefällt mir alles davon, aber die Vereinheitlichung hat für uns viele Vorteile. So hätte ich es früher nie für möglich gehalten, dass man sich als Einzelner in einer so großen Menge an Code zurechtfinden kann. Teamentscheidungen sind auch viel angenehmer als Entscheidungen von Projektleitern, die oft nicht mehr viel aktuelle Programmiererfahrung haben. Vermutlich entscheiden wir in vielen Bereichen sogar konservativer, denn wir müssen unsere eigenen Entscheidungen ja auch ausbaden. Wenn man dann mal daneben liegt und sieht, wie die Entwicklungsgeschwindigkeit sinkt, dann muss man eben den Mut haben, seine Meinung wieder zu ändern.

Was mir auch noch wichtig ist: Entwickeln im Team bringt viel mehr Spaß. Ich möchte agile Softwareentwicklung nicht missen.

Alex Bepple ist Softwareentwickler bei der akquinet it-agile GmbH in Hamburg. Er ist ScrumMaster und überzeugter testgetriebener Entwickler. Aktuell arbeitet er in einem Scrum-/XP-Team. E-Mail: alex.bepple@it-agile.de.

Dipl.-Inform. **Henning Wolf** ist Geschäftsführer der akquinet it-agile GmbH. Er war jahrelang extremer Programmierer. Heute berät er Kunden bei der Einführung agiler Methoden. Er ist Koautor des 2008 bei dpunkt erschienenen Buchs „Agile Softwareentwicklung". E-Mail: henning.wolf@it-agile.de.