Agile en Oracle gaan goed samen

Technische competentie onontbeerlijk

Agile-softwareontwikkeling is tegenwoordig overal. Waar het een decennium terug echt uitzonderlijk was om een Agile project tegen te komen, is het tegenwoordig bijna mainstream te noemen. De meeste ontwikkelaars komen dus langzamerhand wel in aanraking met deze manier van werken. Helaas is het niet altijd even duidelijk wat hierbij komt kijken, en is er op meerdere fronten niet altijd goede begeleiding beschikbaar. Van requirementsanalyse, via planning tot en met de technische werkwijze. En op al die vlakken is weldegelijk een inspanning, en aanpassing, nodig om succesvol 'Agile' te werken.

Voor ontwikkelaars die dicht op de database werken, zijn er ook heel specifieke uitdagingen. Veel van de technische ideeën uit de Agile-wereld zijn namelijk beïnvloed vanuit de object-georiënteerde wereld van Smalltalk en Java. Toch zijn veel van die ideeën goed toepasbaar binnen Oracle, en kunnen ze ook daar veel voordelen bieden. Zeker in projecten waarin dichte samenwerking tussen Oracle en Java noodzakelijk is.

Agile

Laten we eerst nog even kort stilstaan bij wat 'Agile' nu eigenlijk betekent. Of, beter nog, laten we beginnen met wat het niet betekent. Het betekent bijvoorbeeld niet 'niet documenteren', of 'geen analyse doen', of 'niet ontwerpen'. Over het algemeen wordt er in de loop van een project juist méér aandacht daaraan besteed, niet minder.

Geschiedenis

De term 'Agile' is redelijk nieuw. Tien jaar geleden kwam een aantal mensen bij elkaar die, elk op hun eigen manier, hadden geconcludeerd dat een andere aanpak bij softwareontwikkeling gewoon beter werkt. Je moet niet vergeten dat uit bijvoorbeeld het raport van de Standish Group in 1994 bleek dat slechts 15% van IT-projecten slaagde, en dat de gemiddelde kosten en tijd op 170% van de oorspronkelijke raming uitkwamen. Redenen genoeg om te zoeken naar alternatieven.

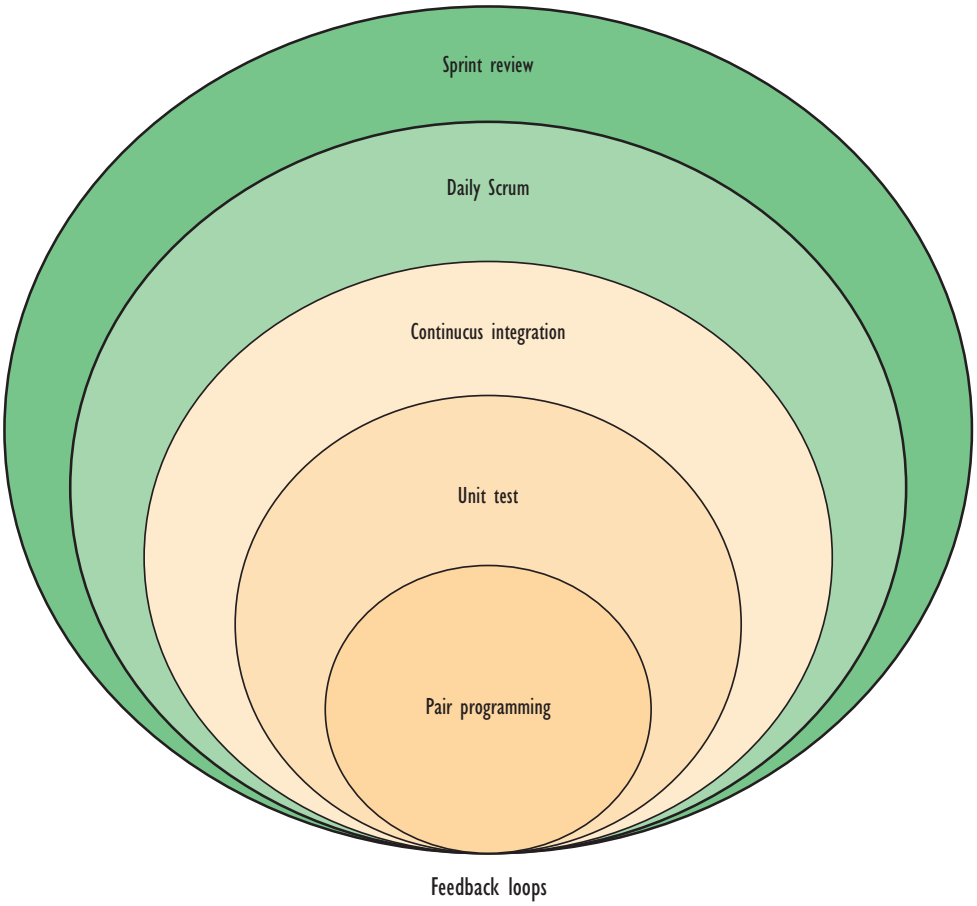
Die alternatieven zelf waren overigens niet zo heel erg nieuw. Of het nu Scrum, Crystal, Extreme Programming, DSDM of Evo heet, het was er al voor de naam Agile werd bedacht. Wat specifiek aan Agile is, is de focus op samenwerking tussen mensen, zowel binnen een organisatie als met de klant, het flexibel kunnen accepteren van veranderingen, en het hechten van groot belang aan het eindresultaat: werkende software, boven andere deliverables.

Uitgangspunten

Bij een Agile-project is het uitgangspunt dus om dicht samen te werken met de klant, en er snel achter te komen dat wat je bouwt niet is wat de klant wil. Het is dan ook cruciaal om zo open mogelijk te communiceren met die klant, alsook met je collega's (en dan het liefst face-to-face, dat is het meest effectief). Tegelijkertijd is het belangrijk dat, zodra je erachter komt dat je inderdaad wat anders moet bouwen, je de code zo gemaakt hebt dat veranderingen zo makkelijk mogelijk door te voeren zijn.

Als je kijkt naar de manier van werken die wordt aangemoedigd in Agile-projecten, dan is één van de belangrijkste uitgangspunten dat van snelle feedback. Zoals ook te zien in het diagram, zijn de verschillende werkwijzen van Agile bedoeld om zo snel mogelijk terugkoppeling te krijgen. Als er één ding is dat we geleerd hebben bij softwareontwikkeling, is het wel dat alles dat fout kán gaan, fout zál gaan. Als we die wijsheid van Murphy toevoegen aan de bevinding van McConnell, dat hoe later je een probleem vindt, hoe duurder het is om het op te lossen, dan is het duidelijk dat hoe eerder je merkt dat het misloopt, hoe beter het is: Fail Early, Fail Often!

Na elke Sprint (Iteratie) kijk je of het goed gaat, en waar je bij kunt sturen. Elke dag houd je elkaar op de hoogte van wat de status is, en waar de problemen zitten tijdens de stand-up-meeting (of 'Daily Scrum' in Scrum). Elke keer dat je code incheckt in de source code repository wordt de code meteen gecompileerd, gedeployed op een testomgeving, en getest



Figuur 1: Het doel van Agile werken is zo snel mogelijk terugkoppeling krijgen.

door de Continuous Integration Server. Bij elke verandering die je maakt, schrijf je direct een 'microtest' of unit-test om te controleren dat de aanpassing inderdaad dat doet wat je van plan was. En als je aan pair-programming doet, corrigeer en review je elkaar van minuut tot minuut. Een kortere feedback-loop is nauwelijks voor te stellen!

Iteratief, incrementeel en multi-disciplinair

De iteratieve aanpak helpt dus om snel bij te sturen. Incrementeel werken betekent dat de dingen die in elke sprint gemaakt worden niet losse technische onderdelen zijn, die later pas aan elkaar worden gekoppeld. Elk stukje werk moet een klein stukje functionaliteit aan de eindgebruiker leveren. Zodanig dat het in gebruik zou kunnen worden genomen.

Het komt vaak voor dat in een project meerdere softwarelagen dienstdoen, zoals 'Oracle', 'Java service' en een 'web front-end'. Incrementeel werken betekent dan dat we in elke Sprint, en voor elk stukje functionaliteit, voor al die betrokken lagen het benodigde werk afmaken en integreren. Dit voorkomt heel veel integratieproblemen in een later stadium, maar vergt wel van het team dat het doorlopend hecht samenwerkt. Dit is ook de reden dat een Scrum-team wordt geacht multi-disciplinair te zijn: alle expertise die nodig is om de functionaliteit op te leveren moet in één team zitten, zodat we geen lastige afhankelijkheden krijgen die ons kunnen vertragen.

Practische implementatie

Bij het werken binnen de korte iteraties willen we ervoor zorgen dat we in kleine stapjes verder komen. Die kleine stapjes zijn goed, maar brengen meestal boven water dat er wat dingen zijn die we moeten doen om iets in productie (of zelfs op test) te krijgen, die veel tijd kosten. Een lange testcyclus. Een buildproces met veel handmatige stappen. Late integratie tussen componenten. Veel bugs bij acceptatietesten. Om beter werk te leveren en vooral om goede samenwerking mogelijk te maken tussen de verschillende technische gebieden is daarom veel discipline nodig.
Voor de beschreven problemen zijn natuurlijk oplossingen. Binnen Agile richten die zich voornamelijk op het verkorten

van de feedbackcycli. In de wereld van Java-development spreekt het voor zich dat bij elke verandering, de bijbehorende micro-/unit-testen worden geschreven, de code in een source code-managementsysteem wordt gezet (subversion, CVS, git), de hele codebase automatisch wordt gebouwd door de build-server, waarbij dan ook meteen alle automatische tests worden uitgevoerd. Mocht er iets misgaan, dan weet je dat binnen tien minuten. Dus ook als er meerdere mensen aan gerelateerde code werken, gaat er nooit te veel tijd verloren met integreren.

Hier een streamer? Hier een streamer? Hier een streamer? Hier een streamer?

Maar hoe kunnen we het Oracle-specifieke werk hierin opnemen? Daar zijn meerdere manieren voor. De auteur van dit artikel heeft in de loop van de tijd een aantal van Agile's voorwaarden ondersteund met zelfgeschreven oplossingen. En voor een aantal andere issues zijn sindsdien standaard bibliotheken en -methodes beschikbaar gekomen, die we dan ook zullen belichten.

Versiebeheer

Een basisbehoefte van elk softwareproject is goed versiebeheer. Alle code, zowel Java, als HTML, DDL en databaseprocedures, moeten in een centraal versiebeheersysteem staan. Om ervoor te zorgen dat de code hanteerbaar blijft, is het ook van belang dat de structuur goed ondersteunt dat er met meerdere mensen tegelijk aan wordt gewerkt. Voor SQL betekent dat bijvoorbeeld dat je het netjes opsplitst in aparte bestanden voor verschillende procedures, tabellen, etc. En dat die bestanden worden opgedeeld over directories voor verschillende packages.

Versiebeheer voor alleen de code is niet genoeg. Ook voor de databasestructuur is versiebeheer noodzakelijk. Als de nieuwe functionaliteit, die in de huidige iteratie speelt, veranderingen veroorzaakt in de databasestructuur, dan moeten die veranderingen afzonderlijk te releasen zijn. En ook een eventuele datamigratie moet dan op eenvoudige wijze uit het systeem te halen zijn. In een van de projecten waarbij we dit hebben toegepast, is dit geregeld door veranderingen in de database door middel van SVN uit de ingecheckte code te extraheren en door een zelfgemaakt script te verzamelen in een database-releasescript.

Continuous Integration

Continuous Integration is de discipline die elke verandering aan de code in het hele project compileert, test, inpakt en deployt naar een testomgeving. Voor systemen die sterk afhankelijk zijn van de database komen daar natuurlijk een heel scala aan haken en ogen bij kijken.
Zoals al genoemd kan bij de build van het project alle databasecode worden meegenomen. Op het testen daarvan komen we hieronder terug. Het verpakken van de databasecode is te doen op basis van versiebeheer, het goed opsplitsen van de code en wat handige buildscripts. Een andere optie is het gebruik van een tool als LiquiBase (http://www.liquibase.org/). Hiermee is het aanpassen van het schema, inclusief mogelijke datamigraties en rollbacks, per database-change te configureren, en kan bij elke release automatisch een migratiescript worden gegenereerd. Inclusief alle nodige veranderingen tussen de versie die in productie staat, en de versie die voor release klaarstaat.

Het belangrijkste is natuurlijk dat dit soort releasescripts automatisch worden gegenereerd bij elke build, om die automatische Continuous Integration mogelijk te maken. Als er tientallen keren per dag een nieuwe build komt, is het tenslotte onmogelijk om dat te doen als de databaseveranderingen niet vanzelf worden meegenomen.

Testen

Als we dan zover zijn dat het hele systeem netjes gebouwd en gedeployed wordt bij elke verandering, dan kunnen we ontdekken of die laatste verandering er niet voor heeft gezorgd dat er iets is stuk gegaan! Dat betekent dat bij elke build er een volledige test wordt gedraaid. Dat kan natuurlijk alleen als die test er is, en netjes (en snel!) zonder menselijke tussenkomst uitvoerbaar is.

Aan de Java-kant van dat verhaal zijn er een heleboel mogelijke manieren om dat aan te pakken. Het schrijven van micro- (of unit-)tests is bijvoorbeeld een geaccepteerde basis voor het creëeren van een goed werkend systeem. Ook integratietesten worden vaak met dezelfde technieken geschreven. Maar hoe doen we dat testwerk voor de database?

Eén optie is om de bestaande Java-infrastructuur voor testen te gebruiken, en de databasetests vanuit Java te draaien. Zolang dat er niet voor zorgt dat het testwerk op Java-ontwikkelaars wordt afgeschoven werkt dat prima, maar het maakt het draaien van de test wel lastiger en langzamer. De andere optie is om in elk geval de microtests te maken en uit te voeren binnen de Oracle-omgeving. Via unit-test-frameworks als utPLSQL (utplsql.sf.net), pluto (http://code.google.com/p/pluto-

test-framework/), Oracle SQLDeveloper (http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html) en/of DBFit (http://www.fitnesse.info/dbfit, meer bedoeld voor integratietesten). Een leuke introductie tot deze materie is te vinden op http://www.oracleunittesting.com/

Testdata

Als gesteld is het essentieel dat de tests bij elke check-in automatisch uitvoerbaar zijn. Dat laatste vergt bij databases wat extra werk, aangezien de database normaal gesproken de status onthoudt. Dus om te voorkomen dat een test alleen de eerste keer goed werkt, moet ook de database state worden ingesteld voor elke testrun.

Er zijn verschillende manieren om hiermee om te gaan. Het is mogelijk om steeds te beginnen met een lege database, die te vullen met een basisset testdata, en daarmee te werken. Zowel het eerder genoemde LiquiBase, als tools zoals DBUnit (http://dbunit.sourceforge.net/) zijn hiervoor goed geschikt, maar zelfbouw is niet al te moeilijk.

Je kunt ook een snapshot van een productiedatabase gebruiken, bijvoorbeeld omdat dat een beter beeld van queryperformance geeft, en die voor elke testrun terugzetten in diens beginstaat. Dit kan met Oracle's flashback functionaliteit, of door de database binnen een virtualisatie systeem te draaien met gebruikmaking van de snapshotfunctionaliteit, zodat je niet steeds grote hoeveelheden data hoeft te kopiëren.

Optioneel kun je ervoor kiezen om de tests steeds netjes de veranderingen weer te laten opruimen, maar dat heeft een risico: een test die faalt, kan de database in een verkeerde staat achterlaten.

Integratie

Genoemde mogelijkheden zijn stuk voor stuk manieren om efficiënter te werken, onafhankelijk van welk proces je gebruikt. Welke de voorkeur ook heeft, de rode draad van het geheel blijft: het werken met kleine stappen, met steeds de feedback van de klant. Toch zie je bij veel teams die beginnen met Agile dat de samenwerking tussen de verschillende disciplines (Java, front-end, Oracle, Test) nog niet optimaal is. Er wordt nog te vaak in een 'mini-waterval'-modus gewerkt, waarbij de Oracle developer eerst zijn taken volbrengt, waarna de Java-programmeur aan de gang gaat, dan de front-end bovenop de Java-service komt, waarna de tests pas plaatsvinden. Vooruit, wel allemaal binnen die ene iteratie, maar het werk wordt nog steeds niet als team opgepakt. Zonde, want door de focus te houden op het helemaal afmaken van elk stuk functionaliteit en door dichtere samenwerking tussen de teamleden, zijn die laatste hand-offs eenvoudig weg te nemen.

Elke dag houd je elkaar op de hoogte van wat de status is, en waar de problemen zitten tijdens de stand-up-meeting.

Het is misschien verrassend dat een verhaal dat begint met meer sociale aspecten als het belang van samenwerking met de klant, en binnen een team, zoveel aandacht geeft aan dit soort technische disciplines. Dat is dan ook één van de valkuilen bij de introductie van Agile-processen zoals Scrum. De regels van het proces zijn op het oog erg eenvoudig, en zeker als je zwaardere processen als RUP en waterval gewend bent, lijken ze makkelijk. Maar als puntje bij paaltje komt merk je, als je echt elke twee weken een complete oplevering moet doen, dat daar een hoog niveau van technische competentie bij komt kijken om het proces te ondersteunen.

Wouter Lagerweij is als Agile Coach werkzaam bij Qualogy, www.qualogy.com