SCM patterns

Samenvatting van Berczuk & Appleton 2003

Inhoud

Figuren

Dit is een samenvatting van Stephen P. Berczuk & Brad Appleton, *Software configuration management patterns: effective teamwork, practical integration*. Boston, Mass.: Addison-Wesley, 2003. De patternnummers tussen haakjes verwijzen naar de hoofdstukken van dit boek. De Engelse SCM-vaktermen zijn veelal onvertaald gelaten (dus geen 'patronen'), al levert dat hier en daar wat verwrongen Nederlands op.

Het boek kijkt vooral de kant op van objectgeoriënteerde systeemontwikkeling op een UNIX- of Windows-platform, maar de meeste beweringen zijn evengoed geldig voor het traditionele COBOL-werk op een mainframe.

1 Begrippen

Workspace: een plek waar een ontwikkelaar alle artefacten bewaart die hij of zij nodig heeft om een taak te vervullen.

Codeline: de opeenvolging van toestanden waarin een coherente verzameling sourcebestanden en andere artefacten verkeert. Telkens wanneer je een artefact in een versiebeheersysteem wijzigt, creëer je een *revisie* van dat artefact. Een codeline bevat iedere versie van ieder artefact volgens één evolutionair pad.

Configuratie van de codeline: een snapshot van de verzameling revisies van elke component op een zeker moment. Iedere configuratie met een eigen naam of nummer is een *versie* van die codeline. Speciale versies kunnen als zodanig worden gemerkt met een *label*.

Er kunnen meerdere codelines zijn, ieder met een eigen doel, en met een eigen *policy*. Naarmate codelines voortschrijden, kun je bemerken dat sommige werkzaamheden in strijd zijn met de oorspronkelijke bedoeling van de codeline. In dat geval mag je de betrokken bestanden aftakken — *branch*en — naar een nieuwe codeline. Vaak zullen de wijzigingen die je daar aanbrengt weer moeten worden terugge*merge*d naar de oorspronkelijke codeline.

Dergelijke ontwikkelingen zijn grafisch weer te geven in zogeheten codelinediagrammen. Daarin wordt een codeline weergegeven als een horizontale lijn, op het startpunt, links, beginnend met een vetgelijnd blokje waarin de naam van de codeline is vermeld. Van links naar rechts kunnen de opeenvolgende versies van de codeline of — als je daarop wilt focussen — de opeenvolgende revisies van een enkel bestand worden geplaatst, in de vorm van cirkeltjes met eventuele identificatie (meestal versienummer). Een reeks van versies kan worden voorafgegaan door een grijs omkaderd blokje met de omschrijving van de taak of het project omwille waarvan die versies zijn ontstaan. Een branch wordt weergegeven met een omhoog of omlaag gerichte pijl, en bij een merge is de pijl gestippeld. Een documentsymbool (papiertje met omgevouwen hoek) gehecht aan het begin van een codeline geeft een policy aan. Een label gehecht aan een versiesymbool is de markering van een revisie met een speciale status, zoals een uiteindelijke release. De meeste van de hier genoemde symbolen kom je tegen in figuur 3.

2 Enige beginselen

De belangrijkste afweging bij softwareconfiguratiemanagement (SCM) is die tussen snelheid en kwaliteit. Bij falend SCM kunnen beide er zelfs onder lijden: langdurige 'freezes' brengen ontwikkelaars in de verleiding om bestanden 'illegaal' te betrekken of te verhandelen, d.w.z. buiten het zicht van het versiebeheersysteem, en op dat moment ben je ook de grip op de kwaliteit kwijt. SCM moet in zulke zaken voorzien als:

- ontwikkelen van de volgende versie van een stuk software terwijl je problemen met de huidige versie aan het verhelpen bent;
- op gecontroleerde wijze delen van code met andere teamleden (parallelontwikkeling, synchronisatie met de huidige codeline);

– vaststellen welke codeversies in een bepaalde component gegaan zijn [noot van de samenvatter: bv. welke COPY-versies met een bepaald COBOL-programma zijn meegecompileerd];
– analyse van waar een zekere change is geïmplementeerd in de wordingsgeschiedenis van een component.

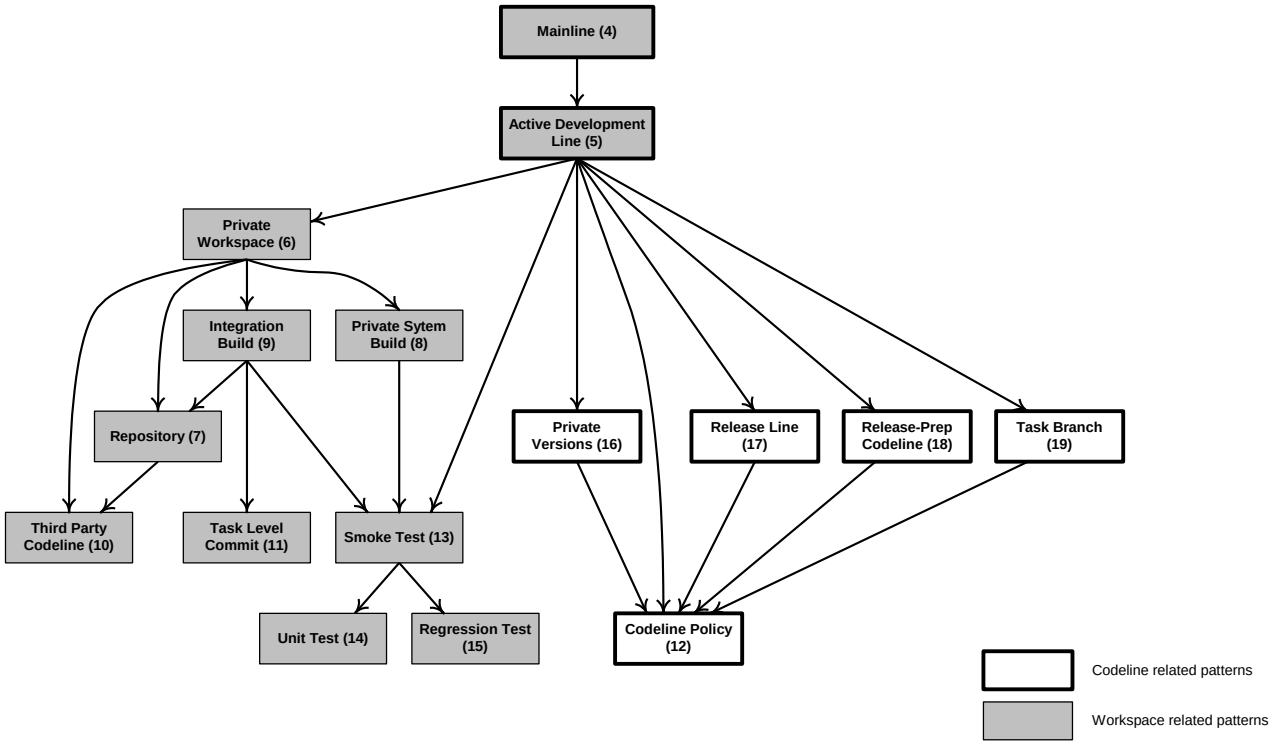
Enige algemene SCM-principes:

– Pas *versiebeheer* toe.
– Doe *periodieke builds*, en integreer veelvuldig. Hoe langer je ermee wacht, des te erger zijn de problemen die je dan tegenkomt.
– Sta *autonoom* werk toe. Ieder team zal wel eens behoefte hebben om aan verschillende 'tijdpunten' (d.w.z. oudere versies) in de codeline te werken.
– Gebruik *tools*. Als het allemaal met de hand gaat, worden er vergissingen gemaakt, en proberen mensen de bureaucratie te ontduiken.

Denk bij 'software-omgeving' aan de volgende elkaar onderling beïnvloedende structuren:

– waar de ontwikkelaar codeert: de *workspace*;
– in welk verband het coderen plaatsvindt: de *organisatie*;
– waar de code inpast: de *productarchitectuur* (hieraan gerelateerd is de releasestructuur, die specificeert hoeveel releases er samen worden ontwikkeld);
– waar de code (en code history) wordt bewaard: de *configuatiemanagementomgeving*, met inbegrip van tools, processen en policies.

3 SCM patterns (3)



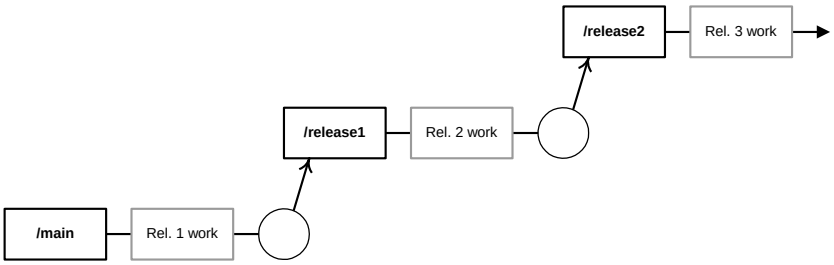
figuur 1 - SCM pattern language

Een pijl van pattern *A* naar pattern *B* in figuur 1 betekent dat pattern *A* in de context van pattern *B* ligt, en ook dat pattern *B* pattern *A* completeert.

Het boek belicht de patterns via de volgende rubrieken: titel, beeld, context, probleemstatement, gedetailleerde probleembeschrijving, samenvatting van de oplossing, gedetailleerde oplossingsbeschrijving, en openstaande kwesties.

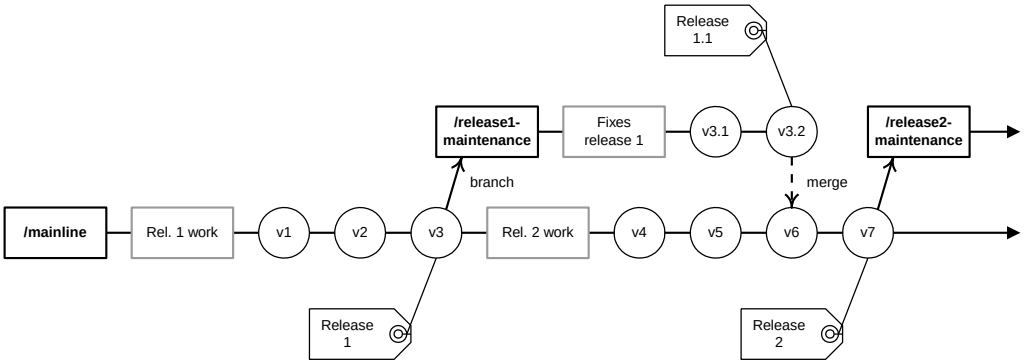
4 Mainline (4)

Probleem: Hoe beperk je je huidige actieve codelines tot een beheersbaar aantal en hoe minimaliseer je de overhead van merging?



figuur 2 – Promotion model: staircase branching (cascade)

Oplossing: Vereenvoudig je branching model (vertakkingsmodel). Ontwikkel vanaf een mainline. Als je werkelijk brancht (een aftakking maakt), denk dan nog eens goed na over je algemene strategie. Kies bij twijfel voor de eenvoud.



figuur 3 – Mainline development & release lines

Bijzondere gevallen:

- Customer releases, waarop bug-fixes kunnen worden aangebracht.
- Parallelle projecten. Maak een *Task branch (19)* aan als de codeline te instabiel dreigt te worden.
- Integratie. Creëer bij customer releases een integration branch als alternatief voor een code freeze, dan kan het werk aan de mainline doorgaan. Bug-fixes in de integration line moeten in de mainline worden geïntegreerd, wat dan niet al te moeilijk meer is.

Bij eventuele branches binnen een project zo snel mogelijk integreren, voordat het te moeilijk wordt. Houd de mainline bruikbaar en correct.

In veel traditionele SCM-processen gaat zo'n 90% van het SCM-'werk' op aan promotie-activiteiten, bij gebrek aan een mainline.

Openstaande kwesties: Hoe houd je de mainline bruikbaar als er zoveel mensen op werken? → *Active Development Line (5)*.

5 Active Development Line (5)

Probleem: Hoe houd je een zich snel ontwikkelende codeline stabiel? Het werk van de ene programmeur aan de ene module kan de (braaf uitgeteste) module van een ander weer 'gebrekkig' maken voordat deze wordt ingecheckt. Een freeze op testen + inchecken maakt de codeline min of meer dood, en branching verhoogt de complexiteit en merge-inspanningen.

Oplossing: Bepaal je doel. Stel policies op die je main development line stabiel genoeg maken om er mee te kunnen werken. Zorg voor checkpoints die gegarandeerd 'goed' zijn (labeling van stabiele builds), en andere die goed genoeg zijn voor wie echt op de spits wil ontwikkelen.

Partijen die behoorlijke stabiliteit nodig hebben, zouden alleen *Named Stable Bases (20)* moeten gebruiken. Als tegen het eind van het project meer stabiliteit nodig is, kan een *Release Line (17)* worden neergezet. Voorzie, om totale chaos op de mainline te voorkomen, iedere ontwikkelaar van een *Private Workspace (6)* waar hij een *Private System Build (8)*, *Unit Test (14)* en *Smoke Test (13)* kan doen.

Openstaande kwesties: Formuleer een *Codeline Policy (12)*. Individuele ontwikkelaars hebben toch afzondering nodig om de *Active Development Line (5)* in leven te houden: → *Private Workspace (6)*. Als de noodzaak van stabiliteit dichterbij komt, zal een deel van het werk naar een *Release-Prep Codeline (18)* moeten worden overgebracht. Sommige langdurige taken hebben misschien meer stabiliteit nodig dan een active development line kan bieden → *Task Branch (19)*; dit is ook een manier om de primaire codeline te isoleren van heel riskante changes.

6 Private Workspace (6)

Probleem: Hoe blijf je in de pas met een voortdurende veranderende codeline en maak je ook nog vorderingen zonder gek te worden van een omgeving die steeds onder je wegdrijft?

Oplossing: Scherm je werk af om grip te krijgen op verandering. Werk in een eigen workspace (liefst evenveel workspaces als dat je taken hebt), waar je zelf volledig kunt bepalen met welke codeversies je werkt. Een workspace is een plaats waar een configuratie-item vele tijdelijke en inconsistente toestanden kan doorlopen aleer het wordt ingecheckt. Niettemin hoort het volgende *niet* in een *Private Workspace* thuis:

- persoonlijke versies van systeembrede scripts die een policy afdwingen;
- componenten die onder versiebeheer vallen, maar van iemand anders zijn betrokken;
- tools (compilers e.d.) die voor alle versies van het product gelijk moeten blijven.

Bij het coderen voor mainline development ga je aldus te werk:

- Zorg dat je qua versies waaraan je wilt gaan werken up to date bent.
- Breng je wijzigingen aan.
- Doe een *Private System Build (8)* om afgeleide objecten te actualiseren.
- Test je wijzigingen uit in een *Unit Test (14)*.
- Werk de workspace bij met de laatste versies van alle componenten die je niet hebt aangepast.
- Rebuild en voer een *Smoke Test (13)* uit om er zeker van te zijn dat je niets kapot hebt gemaakt.

Openstaande kwesties: Heb je eenmaal stabiliteit voor jezelf, dan moet je nog steeds voorkomen dat je fouten in het systeem introduceert wanneer je je aanpassingen incheckt → *Private System Build (8)*. Je dient je workspace te vullen vanuit een *Repository (7)* die alle source en gerelateerde componenten bevat; externe componenten komen uit een *Third Party Codeline (10)*. Eenmaal klaar met je lokale werkzaamheden, moet je het geproduceerde materiaal opnemen in de rest van het systeem via een *Integration Build (9)*.

7 Repository (7)

Voor het vullen van een *Private Workspace (6)* of het uitvoeren van een betrouwbare *Integration Build (9)* heb je de juiste componenenten nodig.

Probleem: Hoe krijg je de juiste versies van de juiste componenten in een nieuwe workspace?

Oplossing: 'One stop shopping.' Zorg voor één uitgiftepunt, ofwel een magazijn, voor alle code en gerelateerde artefacten. Maak het inrichten van een workspace zo simpel en transparant mogelijk. En herhaalbaar, en werkend voor alle versies van het project. Gebruik daartoe bij voorkeur een adequaat versiebeheersysteem.

Openstaande kwesties: Regel code van derden met *Third Party Codeline (10)*.

8 Private System Build (8)

Probleem: Hoe verifieer je dat jouw aanpassingen de build of het systeem niet onklaar maken, voordat je ze incheckt?

Oplossing: 'Think globally by building locally.' Doe vóór het inchecken een eigen build analoog de nachtelijke build. Daar hoort bij:

- moet zoveel mogelijk lijken op de *Integration Build (9)* en de product builds (zelfde compiler, zelfde versies van externe componenten, zelfde directorystructuur);
- bevat alle afhankelijkheden;
- bevat alle componenten die door de aanpassing geraakt worden.

Zorg ervoor dat je workspace schoon is, d.w.z. geen irrelevante / verouderde componenten bevat. Er zijn nog veel keuzes te maken inzake wat je allemaal in je build opneemt.

Openstaande kwesties: Weet je eenmaal dat je een build van het systeem kunt maken, dan is nog altijd niet zeker of je de functionaliteit intact hebt gelaten → *Smoke Test (13)*. Als het systeem erg groot is, is het misschien niet efficiënt meer om alles te builden wat joiuw component gebruikt. Overgebleven afhankelijkheden kunnen worden gevalideerd in de *Integration Build (9)*. Wat te doen als je ten gevolge van jouw aanpassingen een build error krijgt in andere code? Hangt van de teamdynamiek af.

9 Integration Build (9)

Alle ontwikkelaars werken in hun eigen *Private Workspace (6)* zodat ze zelf kunnen bepalen in hoeverre zij andere changes zien. Dat helpt de individuele ontwikkelaars vooruit, maar uiteindelijk moet al het afzonderlijke werk worden geïntegreerd tot één heel systeem.

Probleem: Hoe kom je erachter of de code base altijd betrouwbaar buildt?

Oplossing: Doe een centrale build. Vergewis je ervan dat alle wijzigingen (en hun afhankelijkheden) worden meegenomen in een centraal buildproces. Dit buildproces behoort:

- reproduceerbaar te zijn;
- zoveel mogelijk op de build van het eindproduct te lijken;
- zomin mogelijk menselijke interventie te vergen;
- gelogd te worden om build errors te kunnen terugvinden.

Als de omstandigheden dat toelaten, kan er bij elke check-in een build worden gedaan. Identificeer de build met een label.

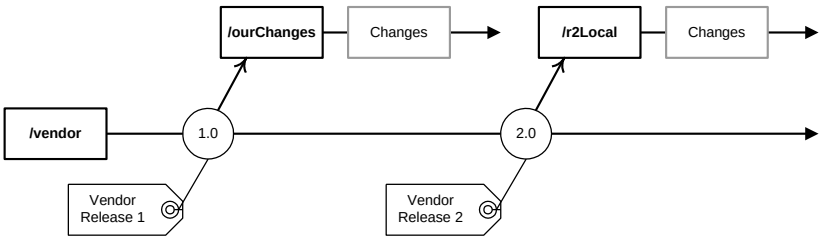
Openstaande kwesties: Een geslaagde build betekent nog niet dat het systeem goed werkt. Laat de integration build volgen door een *Smoke Test (13)* om de bruikbaarheid van de build te

garanderen. Indien de build ook gepubliceerd wordt als een named stable bases, doe dan ook een *Regression Test (15)*.

10 Third Party Codeline (10)

Probleem: Wat is de meest effectieve aanpak om de versies van de door derden geleverde producten af te stemmen met de versies van eigen producten?

Oplossing: Gebruik de tools die je al hebt. Leg een codeline aan voor code van derden. Bouw vanuit deze codeline workspaces en installatiekits op. Neem de versies van de leverancierssoftware eveneens in het versiebeheersysteem op.



figuur 4 – Third-party codeline

Let op: patches van eigen makelij op software van derden zijn eigen producten, niet de producten van de leverancier van de gepatchte software.

Openstaande kwesties: Voor producten van derden die zeer stabiel zijn en die je nooit zult aanpassen, hoef je misschien geen branch aan te leggen. De kosten van zo'n branch zijn klein, en je krijgt de flexibiliteit om later toch veranderingen aan te brengen.

11 Task Level Commit (11)

Een *Integration Build (9)* is gemakkelijker te debuggen als je weet wat erin ging. Hieronder een afweging van stabiliteit, snelheid en atomiciteit.

Probleem: Hoeveel werk moet je doen, oftewel hoe lang moet je wachten, alvorens in te checken in het versiebeheersysteem?

Oplossing: Doe één commit per fijnkorrelige, consistente taak. Dat stelt je in staat de belangrijkste wijzigingen eerst aan te brengen. De eenheid van werk kan een nieuwe feature zijn, een probleemrapport, of een refactoringtaak. Wijzigingingen kunnen worden gestapeld als dat de tijdspanne waarin je aan je lokale kopie werkt niet serieus verlengd. Betrek in je afwegingen de complexiteit, het aantal aan te passen componenten, en de mate waarin de aanpassing het systeem beïnvloedt (het risico).

Neig bij twijfel naar meer check-ins. Dat is gemakkelijker bij roll-backs, en geeft in de administratie ook beter de 'hartslag' van het ontwikkelwerk weer. Streef naar minstens één check-in per dag, voor zover zinvol.

Laat om hinder van de precheck-in validatie te voorkomen, deze zo gestroomlijnd zijn dat alleen wordt getest wat echt nodig is.

Openstaande kwesties: Sommige changes zijn verreikend, ingrijpend, ontwrichtend, en de verwezenlijking ervan heeft een lange doorlooptijd → *Task Branch (19)*. Unit tests, smoke tests, regression tests en de edit policy geven een richtsnoer hoe fijnkorrelige check-ins aan te moedigen. Goede integratie van ontwikkelomgeving met versiebeheersysteem laat het commitproces beter inpassen in de flow van het ontwikkelwerk.

12 Codeline Policy (12)

Probleem: Hoe weten de ontwikkelaars in welke codeline ze hun code moeten inchecken, wanneer ze dat moeten doen, en welke tests ze voor de check-in moeten uitvoeren?

Oplossing: Omschrijf de verkeersregels. Formuleer voor iedere branch of codeline een policy die bepaalt hoe en wanneer ontwikkelaars veranderingen moeten aanbrengen. De policy moet bondig en controleerbaar zijn. Mogelijke inhoud:

- een coherent doel;
- hoe en wanneer elementen ingecheckt, uitgecheckt, gebrancht en gemerged moeten worden;
- toegangsbeperkingen voor verschillende individuen, rollen, groepen;
- import/export-relaties: de namen van de codelines vanwaar de changes binnenkomen en die van de codelines waaraan het changes moet doorgeven;
- de duur van het werk of de voorwaarden voor opheffing van de codeline;
- de verwachte activiteitlast en de integratiefrequentie.

Omvang: één à drie paragrafen, maximaal één pagina.

Branch wanneer je een onverenigbare policy hebt. Voorbeelden van policies:

- Development codeline — tussentijds code-aanpassingen mogen worden ingecheckt; geraakte componenten moeten buildbaar zijn.
- Release codeline — software moet builden en regression test passeren vóór de check-in; check-ins blijven beperkt tot bug-fixes; er mogen geen nieuwe features of functionaliteit worden ingecheckt; de branch wordt bevroren totdat de hele QA-cyclus is doorlopen.
- Mainline — alle componenten moeten compileer- en linkbaar zijn en door de regression test heenkomen; wanneer klaar, mogen geteste nieuwe features worden ingecheckt.

Openstaande kwesties: Om een codeline-policy af te dwingen, zullen de mogelijkheden van automatisering en van de groepscultuur moeten worden afgewogen.

13 Smoke Test (13)

Een *Integration Build (9)* en een *Private System Build (8)* zijn geschikt om buildtime-zaken te verifiëren. Maar zelfs als de code buildt, is het nog steeds nodig om de runtime-kwaliteit te toetsen. Essentieel als je een *Active Development Line (5)* wilt onderhouden.

Probleem: Hoe weet je of het systeem nog werkt nadat je iets hebt aangepast?

Oplossing: Verifieer de basisfunctionaliteit. Onderwerp iedere build aan een smoke test[1] die verifieert of de applicatie niet overduidelijk kapot is.

Een smoke test moet goed genoeg zijn om 'showstopper' defecten te vangen: je test alleen de basale functies en eenvoudige integratie-aangelegenheden, maar dat alles wel met een hoge dekkingsgraad van het systeem. De smoke test is geen vervanging van het diepergaande integratietesten, en ontslaat de ontwikkelaar evenmin van de verplichting zijn werk zelf te testen alvorens het aan de repository toe te vertrouwen. Een smoke test als onderdeel van een dagelijkse build — *Daily Build and Smoke Test (20)* — is de sleutel om tot *Named Stable Bases (20)* te komen, die op hun beurt de basis vormen voor de vulling van workspaces.

[1] Noot van de samenvatter: *smoke test* (in het Nederlands te vertalen met *rooktest*) is een metafoor, ontleend aan (1) het loodgietersbedrijf, waar rook door nieuwe of gerepareerde leidingen wordt gejaagd om eventuele lekkage vast te stellen, en (2) de electronica, waar rook uit een circuit nadat de spanning erop is gezet wijst op een serieus mankement.

Een smoke test moet 'snel' uit te voeren zijn, en met zomin mogelijk menselijke tussenkomst. Lastig punt bij het inrichten van zo'n test is het voorzien in realistische data.

Zorg dat je een consistente build aan het testen bent, denk aan een *Private System Build (8)*.

Als de kwaliteitseisen zodanig zijn dat je volledig moet testen, overweeg dan het gebruik van task branches, het branchen van release lines, of houdt een andere codeline policy aan.

Openstaande kwesties: Een smoke test laat gaten vallen, die moeten worden gedicht door een grondige QA-procedure. De *Regression Test (15)* dient eventuele degeneratie van het systeemgedrag zichtbaar te maken, en de ontwikkelaar moet via een *Unit Test (14)* verifiëren dat de module die hij wil inchecken behoorlijk werkt.

We moeten zien in hoeverre we de snelheid van de check-in willen uitruilen tegen de grondigheid van de test. Een langere check-intijd verleidt de ontwikkelaars tot grofkorreliger commits, wat tegen de geest van versiebeheer ingaat.

14 Unit Test (14)

Een *Smoke Test (13)* is niet genoeg om een wijziging aan een module in detail te testen.

Probleem: Hoe test je of een module na een aanpassing nog naar behoren werkt?

Oplossing: Test het contract. Ontwikkel unit tests en voer ze uit, om te zien of een component zich nog aan zijn contract houdt. Een goede unit test heeft de volgende eigenschappen:

- *Automatisch en zelfevaluerend*. De gebruiker zou alleen de gedetailleerde testresultaten moeten hoeven inzien als er iets fout is gegaan.
- *Fijnkorrelig*. Elke significante interfacemethode van een klasse dient bekende input te testen. Test geen triviale zaken, maar alleen de dingen die kapot kunnen gaan.
- *Op zichzelf staand*. Een unit test heeft geen interacties met andere tests.
- *Het contract toetsend*. Externe wijzigingen mogen de resultaten niet beïnvloeden. Als een externe interface verandert, moet je de test daarop aanpassen.
- *Eenvoudig uit te voeren*. De spreekwoordelijke druk op de knop, verder geen gedoe.

Voer unit tests uit:

- tijdens het coderen;
- vlak voor het inchecken en na het updaten van je code naar de huidige versie;
- als je een probleem moet oplossen (uit smoke test, regression test, probleem- of incidentmelding van een gebruiker).

Unittesten is onmisbaar bij structuurwijzigingen die het gedrag niet mogen beïnvloeden, zoals refactoring.

Is het moeilijk om een goede unit test te verzinnen voor een klasse of verzameling van klassen, overtuig je er dan van dat het ontwerp niet te gecompliceerd of te abstract is.

Openstaande kwesties: Unit tests schrijven kan vervelend werk zijn. Als je publieke interface nauw is, maar je andere functies wilt testen, dan moet je bekijken of je de interface wijder open wilt zetten dan wel een andere list verzint.

15 Regression Test (15)

Een *Smoke Test (13)* is snel, maar niet uitputtend. Als je release candidates wilt gaan opbouwen, dien je er zeker van te zijn dat de code base robuust is.

Probleem: Hoe ben je er zeker van dat je bestaande code niet bederft wanneer je verbeteringen aanbrengt?

Oplossing: 'Test for changes.' Voer regression tests uit zodra je zeker wilt zijn van de stabiliteit van de codeline, zoals vóór het vrijgeven van een build of voor een bijzonder riskante change. Stel de regression tests samen uit testgevallen waarop het systeem in het verleden stukliep:

- problemen die je tegenkomt in het prerelease QA-proces;
- problemen gemeld door de klant of door de gebruikers;
- tests op systeemniveau die zijn gebaseerd op requirements.

Bij ieder probleem kunnen verscheidene testgevallen betrokken zijn. Verwijder alleen testgevallen uit de aldus opgebouwde testset als daar goede redenenen voorzijn. Je kunt unit tests meenemen met het regressietesten, maar het is beter als er systeeminput bij betrokken wordt.

Een regression test kan een gebrek op systeemniveau zichtbaar maken, maar hoeft niet precies aan te geven waar de fout zit (daarvoor kunnen aanvullende debugging en unit tests nodig zijn).

Omdat regression tests lang kunnen duren, wil je ze misschien niet vóór elke check-in of zelfs na elke build draaien. Niettemin is het aan te bevelen de regression test als onderdeel van de nachtelijke build te draaien, en ontwikkelaars zouden het voor iedere substantiele en verstrekkende aanpassing moeten doen.

16 Private Versions (16)

Soms wil je snel een gecompliceerde change evalueren die het systeem onderuit kan halen terwijl je een *Active Development Line (5)* aanhoudt.

Probleem: Hoe kun je experimenteren met een complexe change en profiteren van het versiebeheersysteem zonder de change publiek te maken? En zonder de versiehistory in de codeline te belasten met teruggedraaide wijzigingen?

Oplossing: Een privé-history. Voorzie ontwikkelaars van een mechanisme om checkpoints aan te brengen met een korreligheid waar ze zich prettig bij voelen. Dat kan worden verschaft door middel van een local revision control area. Alleen stabiele code sets worden ingecheckt in de project repository.

Er zijn vele manieren om dit te implementeren:

- Een hele *Private Workspace (6)* gewijd aan een taak. Vooral geschikt voor experimenten met een globale change (bv. van een interface).
- Een 'privé-repository', of een ontwikkelaarspecifieke branch van de main repository die niet is geïntegreerd met de active line. Het onderlinge verkeer moet stroken met de policies.

Sommige tools voorzien in promotion levels ofwel stages.

Het is belangrijk dat ontwikkelaars die aan private versioning doen eraan denken hun werk met redelijke intervallen te migreren naar het gemeenschappelijke versiebeheersysteem. Dat lukt het best als alles kan worden geïmplementeerd binnen het raamwerk van één tool.

17 Release Line (17)

Probleem: Hoe pleeg je onderhoud op vrijgegeven versies zonder je lopende ontwikkelwerk te verstoren, en zonder dat het ontwikkelwerk de vrijgegeven code base instabiel maakt?

Oplossing: Branch alvorens te releasen (vrij te geven). Splits maintenance/release en active development uit in aparte codelines. Houd elke vrijgegeven versie op een release line, en laat deze line zelfstandig doorlopen ten behoeve van bug-fixes. Branch iedere release van de mainline, maar label de code op de mainline dan eerst. Zie ook figuur 3.

Zorg ervoor dat changes in de maintenance/release line regelmatig naar de active development line worden doorgezet. Geef bug-fixes in de mainline waar mogelijk door naar de release line. De release line loopt pas dood als de betreffende release niet meer wordt ondersteund.

18 Release-Prep Codeline (18)

Je legt de laatste hand aan een release en moet alvast met ontwikkelen voor de volgende release beginnen. Je wilt een *Active Development Line (5)* aanhouden.

Probleem: Hoe stabiliseer je een codeline voor een op handen zijnde release en sta je tegelijkertijd nieuw werk op een active codeline toe?

Oplossing: 'Branch instead of freeze.' Creëer een release engineering branch wanneer de code releasekwaliteit begint te krijgen. Voltooi de release op deze branch, en verlaat de mainline om plaats te maken voor active development. De branch begint als Release-Prep Codeline, en wordt na een succesvolle release *Release Line (17)*.

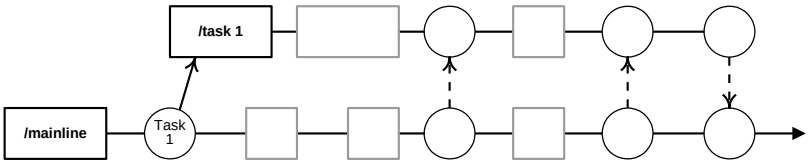
Begin de 'anti-freeze' line ook weer niet te vroeg, want dat betekent potentieel meer mergewerk.

Openstaande kwesties: Werken er maar weinig mensen aan de volgende release, begin dan een *Task Branch (19)* voor het nieuwe werk in plaats van een release-prep branch voor de huidige release.

19 Task Branch (19)

Probleem: Hoe kan jouw team meerdere langdurige, elkaar overlappende wijzigingen in een codeline aanbrengen zonder haar consistentie en integriteit te beschadigen?

Oplossing: Gebruik branches voor afzondering. Splits een aparte branch af voor iedere activiteit die aanmerkelijke veranderingen op een codeline teweegbrengt.



figuur 5 – Task branch

De wijzigingen op de active development line moeten frequent in de task branch worden geïntegreerd, om de uiteindelijke integratie van de task branch in de active codeline zo soepel mogelijk te laten verlopen.

Task branches zijn ook bijzonder nuttig wanneer verscheidene mensen werk delen dat uiteindelijk één geheel moet vormen. Je kunt het integratiewerk dan op een aparte task branch uitvoeren, om na succes de task branch terug te mergen in de active development line.

20 Andere patterns

De in dit hoofdstuk opgevoerde patterns zijn oorspronkelijk gepubliceerd in James O. Coplien, 'A generative development process pattern language', in *Pattern languages of program design*. Reading, Mass.: Addison-Wesley, 1995.

20.1 NAMED STABLE BASES (20)

Hoe vaak integreer je? Stabiliseer systeeminterfaces niet meer dan eens per week. Andere software kan frequenter worden gewijzigd en geïntegreerd.

20.2 DAILY BUILD AND SMOKE TEST (20)

Hoe voorkom je dat wijzigingen uit de hand lopen en de build bederven? Build ten minste één keer per dag de software en voer een smoke test uit om vast te stellen dat de software nog steeds bruikbaar is.

A Tools en termen

In deze samenvatting is alleen Visual Source Safe (VSS) van Microsoft uitgewerkt, omdat dit product destijds (2006) bij AEGON in gebruik was. Van andere producten zijn slechts de daar gebruikte termen genoemd, om de diversiteit van de terminologie in de SCM-wereld te illustreren.

SCM pattern term	VSS-term	Elders gebezigde termen
Repository	Master project	Depot, master repository, versioned object base, project, base database, project database, top-level project
Development workspace	Working directory	Working copies, client workspace, client spec, developer repository, workspace, development stream, work area, working files/folders, private workset, sandbox
Codeline	Share *(voorheen* pin, *linkt)*; branch *(kopieert)*	Branch, integration repository, stream, project branch, project, project object, project view, named branch, development path
Change task	*(niet aanwezig)*	Changelist, change-set, transaction, activity, task, work package, change package
Workspace update	Get project	Update, sync, pull and resolve, findmerge, rebase, update members, check-in, resync
Task-level commit	Check-in project	Commit, submit, commit and push, promote, findmerge, deliver, complete task, merge, promote action, check-in
Task branch	*(niet aanwezig)*	Branch, developer repository, workspace stream, private branch, activity, task, branching view, work package, change package
Label	Label project	Tag, real version, baseline, version label, project checkpoint
Checkpoint	*(niet aanwezig)*	Change-set, tag, view label
Third party codeline	*(niet aanwezig)*	Vendor branch
Codeline policy	*(niet aanwezig)*	Project policy, project template, control plan, promotion groups

Ofschoon VSS een aardige GUI had en naadloos aansloot op andere Microsoft-producten, was het een van de minst capabele onder de talrijke commerciële en open source tools. Het bood weinig ondersteuning aan branching en parallel ontwikkelen, en was niet geschikt voor projecten die regelmatig meerdere codelines nodig hadden.[2, 3]

[2] De samenvatter kon daar in 2006 nog het volgende aan toevoegen: ook andere bronnen doen nogal geringschattend over VSS. De belangrijkste klachten zijn: (1) VSS crasht nogal eens, met soms een onherstelbare repository tot gevolg; (2) er is in geen tien jaar serieus iets aan VSS verbeterd, het product is allang ingehaald door de concurrentie; (3) het is erg rigide. Dat laatste wordt ook wel als voordeel gezien, het is maar welke policy je aanhangt. Veelbetekenend is verder dat Microsoft in eigen huis nooit VSS heeft gebruikt voor versiebeheer.

[3] Anno 2024 is VSS lang en breed geschiedenis, zie https://en.wikipedia.org/wiki/Microsoft_Visual_SourceSafe. Daarom is deze appendix in de huidige bewerking van dit document omgewerkt naar de verleden tijd.