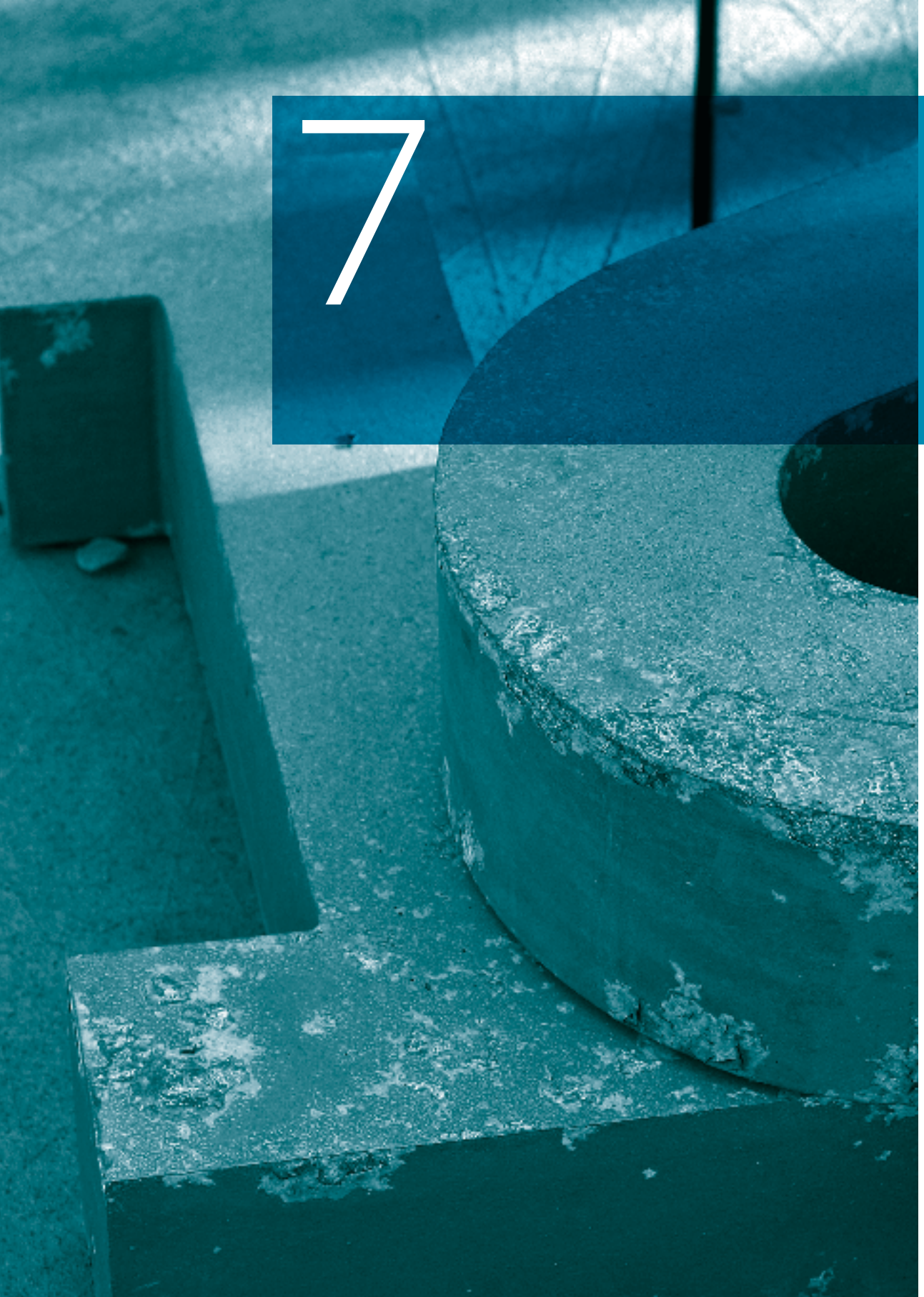

7

DOWNLOADPAKKETTEN MAKEN, RELEASES EN DAGELIJKSE ONTWIKKELING

Dit hoofdstuk gaat over hoe open source-softwareprojecten downloadpakketten maken en de software uitbrengen en over de manier waarop algemene ontwikkelingspatronen rond deze doelstellingen moeten worden georganiseerd.

Een belangrijk verschil tussen open source-projecten en propriëtaire projecten is het ontbreken van een gecentraliseerde controle over het ontwikkelingsteam. Dit verschil is met name van belang bij het voorbereiden van een nieuwe release. Een bedrijf kan het gehele team van ontwikkelaars vragen zich te concentreren op de komende release en nieuwe toekomstige ontwikkelingen en het verhelpen van niet-kritische bugs op een laag pitje te zetten totdat de release is uitgebracht. Groepen vrijwilligers zijn niet zo homogeen. Mensen werken om uiteenlopende redenen aan een project en mensen die niet geïnteresseerd zijn in een bepaalde release willen liever doorgaan met hun normale ontwikkelingswerk terwijl de release wordt voorbereid en uitgebracht. Omdat de ontwikkeling niet wordt onderbroken, duren releaseprocessen van open source-projecten meestal langer, maar ontwrichten het proces minder dan commerciële releaseprocessen. Het lijkt een beetje op een reparatie aan een snelweg. Er zijn twee manieren om een weg te repareren: je kunt de weg volledig afsluiten, zodat de werklui zich volledig en met de volle capaciteit kunnen inzetten totdat het probleem is opgelost, of je kunt aan een paar rijbanen tegelijk werken terwijl de andere voor verkeer open blijven. De eerste manier is het meest efficiënt *voor de werklui*, maar niet voor de weggebruikers. De weg wordt volledig afgesloten totdat de klus is geklaard. De tweede manier kost veel meer tijd en moeite van de werklui (ze moeten met minder mensen en minder apparatuur werken, op een beperkte ruimte en met mensen die met vlaggen het verkeer moeten waarschuwen enz.), maar de weg blijft bruikbaar, hoewel niet volledig.

Open source-projecten werken meestal op de tweede manier. In feite bevindt het project, wanneer er sprake is van volledig ontwikkelde software met verschillende releaselijnen die tegelijkertijd worden onderhouden, zich permanent in een staat van kleinschalige wegwerkzaamheden. Er zijn altijd wel een paar rijbanen afgesloten en de groep ontwikkelaars tolereert aanhoudende maar wel minimale ongemakken, zodat de releases volgens een regulier programma kunnen worden uitgebracht.

Het model dat dit mogelijk maakt, zorgt ervoor dat er meer dan één release tegelijk kan worden uitgebracht. Het parallel laten lopen van taken die niet van elkaar afhankelijk zijn, is natuurlijk in geen geval uniek voor de ontwikkeling van open source-software, maar wel een principe dat open source-projecten op een geheel eigen wijze implementeren. Ze kunnen het zich niet veroorloven de werklui én het normale verkeer te veel te hinderen, maar ze kunnen het zich ook niet veroorloven mensen speciaal toe te wijzen aan het zwaaien met vlaggetjes om het verkeer te waarschuwen. Daarom moeten ze terugvallen op processen met een laag en constant niveau van administratieve rompslomp, in plaats van één met pieken en dalen. Vrijwilligers zijn over het algemeen bereid te werken met een kleine maar consistente hoeveelheid ongemak, waarvan de voorspelbaarheid hun de mogelijkheid geeft om in hun eigen tempo te werken zonder dat ze zich zorgen hoeven maken dat hun agenda botst met wat er binnen het project gebeurt. Als het project echter afhankelijk zou zijn van een overkoepelend programma waarbij sommige activiteiten andere activiteiten uitsluiten, zou het gevolg zijn dat veel ontwikkelaars een groot deel van de tijd niets kunnen doen. Dat is niet alleen weinig efficiënt maar ook nog eens saai en daarom riskant, omdat de kans groot is dat een verveelde ontwikkelaar straks een ex-ontwikkelaar is.

Releasewerk is vaak de meest in het oog springende niet-ontwikkelingstaak die parallel aan het ontwikkelen plaatsvindt. De in de volgende gedeelten beschreven methodes zijn dan ook grotendeels gericht op het mogelijk maken van releases. Merk echter op dat ze ook van toepassing zijn op andere taken die naast elkaar kunnen lopen, zoals vertalingen en internationalisering, breedschalige API-veranderingen die stapsgewijs door het hele codebestand worden doorgevoerd enz.

7.1 RELEASENUMMERS

Voordat we het gaan hebben over hoe een release moet worden gemaakt, kijken we eerst naar hoe we releases een naam moeten geven. Daarvoor moeten we weten wat releases in de praktijk voor gebruikers betekenen. Een release betekent dat:
oude bugs zijn opgelost. Dit is waarschijnlijk het enige waar gebruikers zeker van kunnen zijn bij iedere release;

- er nieuwe bugs zijn bijgekomen. Ook dit is meestal zeker, behalve soms in het geval van beveiligingsreleases en andere eenmalige releases (zie het gedeelte ‘Beveiligingsreleases’ verderop in dit hoofdstuk);
- er nieuwe functies zijn toegevoegd;
- er nieuwe configuratieopties kunnen zijn toegevoegd of de betekenis van oude opties iets kunnen zijn veranderd. De installatieprocedure kan ook iets zijn veranderd ten opzichte van de laatste release, hoewel iedereen altijd hoopt dat dit niet het geval is;
- er veranderingen kunnen zijn doorgevoerd waardoor nieuwe versies niet meer compatibel zijn met oudere versies, zodat gegevensformaten die worden gebruikt door oudere versies van de software niet langer gebruikt kunnen worden zonder één of andere vorm van (mogelijk handmatige) conversie.

Zoals u kunt zien zijn het niet alleen goede dingen die veranderen. Daarom benaderen ervaren gebruikers nieuwe releases altijd met enige terughoudendheid, met name wanneer de software al uitontwikkeld is en in grote lijnen deed wat zij wilden (of dachten dat zij wilden). Zelfs de komst van nieuwe functies kan een twijfelachtig voordeel zijn, omdat de software onverwachte dingen kan gaan doen.

De bedoeling van releasenummers is daarom tweeledig. Uiteraard laten de nummers ondubbelzinnig de volgorde van de releases zien. Dat wil zeggen dat men aan twee releasenummers direct kan zien welke de meest recente is. Ze kunnen echter ook in een zo compact mogelijke vorm de mate en aard van de veranderingen in een release aangeven.

En dat allemaal met één nummer? Nou ja, min of meer. De discussie over de methodiek van de releasenummering is zo'n beetje de moeder van alle triviale discussies (zie het gedeelte ‘Hoe makkelijker het onderwerp, des te langer de discussie’ in Hoofdstuk 6, *Communicatie*) en het valt dan ook niet te verwachten dat er in de nabije toekomst één complete, mondiale norm zal komen. Er bestaan echter enkele goede methodes, alsmede één principe waar iedereen het over eens is: *wees consistent*. Kies een nummeringssysteem, documenteer het en wijk er niet van af. Uw gebruikers zullen u er dankbaar voor zijn.

Componenten van releasenummers

Dit gedeelte beschrijft in detail de formele conventies van releasenummering. Hiervoor is zeer weinig voorkennis nodig. Het is vooral als referentiemateriaal bedoeld. Als u reeds op de hoogte bent van deze afspraken, kunt u dit gedeelte overslaan.

Releasenummers zijn groepen getallen gescheiden door punten:

Scanley 2.3
Singer 5.11.4
... enz. De punten zijn *geen* decimale scheidingstekens, ze zijn alleen bedoeld als separator. ‘5.3.9’ wordt opgevolgd door ‘5.3.10’. Een paar projecten hebben het soms iets anders aangepakt. Het beroemdste voorbeeld daarvan is de Linux-kernel met de nummering ‘0.95’, ‘0.96’ ... ‘0.99’ die uiteindelijke leidde naar Linux 1.0. De algemene afspraak dat de punten geen decimale scheidingstekens zijn, is inmiddels echter stevig verankerd en moet als norm worden beschouwd. Er is geen limiet aan het aantal componenten (groepjes cijfers die geen punt bevatten), maar de meeste projecten gebruiken er niet meer dan drie of vier. De redenen daarvoor worden later duidelijk.

Naast de numerieke componenten voegen projecten soms ook een beschrijvend label toe, zoals ‘alfa’ of ‘bèta’ (zie alfa en bèta), bijvoorbeeld:

Scanley 2.3.0 (Alpha)
Singer 5.11.4 (Beta)

Een alfa- of bèta-achtervoegsel betekent dat deze release *voorafgaat* aan een toekomstige release met hetzelfde nummer, maar dan zonder het achtervoegsel.

‘2.3.0 (alfa)’ leidt dus uiteindelijk naar ‘2.3.0’. Om verschillende van deze kandidaat-releases achter elkaar te kunnen vrijgeven, kunnen deze achtervoegsels zelf zijn voorzien van meta-achtervoegsels. Hier is bijvoorbeeld een reeks releases in de volgorde waarmee ze beschikbaar zouden worden gesteld aan het publiek:

Scanley 2.3.0 (Alpha 1)
Scanley 2.3.0 (Alpha 2)
Scanley 2.3.0 (Beta 1)
Scanley 2.3.0 (Beta 2)
Scanley 2.3.0 (Beta 3)
Scanley 2.3.0

Merk op dat wanneer er een ‘alfa’-achtervoegsel is, Scanley ‘2.3’ wordt geschreven als ‘2.3.0’. De twee nummers zijn gelijkwaardig. Componenten met alleen nullen kunnen altijd worden weggelaten om de naam korter te maken. Wanneer er echter een achtervoegsel is, is bondigheid sowieso niet van het grootste belang en kan men net zo goed kiezen voor volledigheid.

Andere achtervoegsels die semiregulier zijn, zijn onder andere ‘Stable’, ‘Unstable’, ‘Development’ en ‘RC’ (voor ‘Release Candidate’). De meest gebruikte zijn nog steeds ‘alfa’ en ‘bèta’, met ‘RC’ vlak daarachter als goede derde. NB: ‘RC’ heeft altijd een meta-achtervoegsel. Dat wil zeggen dat u nooit release ‘Scanley 2.3.0 (RC)’ uitgeeft, maar wel ‘Scanley 2.3.0 (RC 1)’, gevolgd door RC 2 enz.

Deze drie labels, ‘alfa’, ‘bèta’ en ‘RC’, zijn inmiddels vrij algemeen bekend en ik adviseer geen andere te gebruiken, zelfs wanneer andere achtervoegsel op het eerste gezicht betere opties lijken omdat het gewone woorden zijn en geen jargon. Mensen die releasesoftware installeren zijn echter reeds bekend met deze grote drie en er is geen reden dingen nodeloos anders te doen dan anderen.

Hoewel de punten in releasenummers geen decimale punten zijn, geven ze wel het belang aan van een cijfer door de plaats waar het staat. Alle releases ‘0.X.Y’ gaan vooraf aan ‘1.0’ (wat uiteraard gelijk staat aan ‘1.0.0’). ‘3.14.158’ gaat direct vooraf aan ‘3.14.159’ en gaat ook vooraf, maar niet direct, aan ‘3.14.160’, evenals ‘3.15. watdanook’ enz.

Een consistent beleid ten aanzien van releasenummers geeft de gebruiker de kans aan de hand van twee releasenummers van de software de belangrijkste verschillende tussen de twee releases te herkennen. In een doorsneesysteem met drie componenten is de eerste component het *hoofdnummer*, het tweede het *subnummer* en het derde het *micronummer*. Release ‘2.10.17’ is bijvoorbeeld de zeventiende microrelease uit de tiende subreleasereeks binnen de tweede hoofdreleaseserie. De woorden ‘reeks’ en ‘serie’ worden hier informeel gebruikt, maar ze betekenen wel wat men verwacht dat ze betekenen. Een hoofdserie bestaat simpelweg uit alle releases die hetzelfde hoofdnummer hebben en een subserie (of subreeks) bestaat uit alle releases die hetzelfde sub- *en* hoofdnummer hebben. Dat wil zeggen dat ‘2.4.0’ en ‘3.4.1’ niet in dezelfde subreeks zitten, hoewel het subnummer voor beide wel een ‘4’ is. Aan de andere kant zitten ‘2.4.0’ en ‘2.4.2’ wel in dezelfde subreeks, hoewel ze elkaar niet direct opvolgden indien ‘2.4.1’ tussen beide in is uitgebracht.

De betekenis van deze cijfers is precies wat u ervan zou verwachten: een stijging in het hoofdnummer betekent dat er een belangrijke verandering is, een stijging in het subnummer geeft aan dat het om een kleine verandering gaat en een verhoging van het micronummer staat voor onbeduidende veranderingen. Sommige projecten voegen hier nog een vierde component aan toe, dat meestal het *patchnummer* wordt genoemd, voor bijzonder minutieuze controle over de verschillen tussen de releases (verwarrend genoeg gebruiken andere projecten het woord ‘patch’ als synoniem voor ‘micro’ in een systeem met drie componenten). Er zijn ook projecten die de laatste component gebruiken als *build number*, dat iedere keer wordt verhoogd wanneer de software wordt opgebouwd *(built)* en dat geen enkele andere verandering aangeeft dan de betreffende build. Dit helpt het project ieder bugrapport te koppelen aan een specifieke build en is waarschijnlijk het meest bruikbaar wanneer er binaire pakketten worden gebruikt als standaarddistributiemethode.

Hoewel er zeer veel verschillende afspraken zijn over hoeveel componenten er moeten worden gebruikt en wat de componenten betekenen, zijn de verschillen daartussen minimaal. U hebt dus een klein beetje speelruimte, maar niet veel. De volgende twee gedeelten behandelen enkele van de meest gebruikte systemen.

De simpele methode

De meeste projecten hebben regels over welke soort veranderingen zijn toegestaan in een release wanneer alleen het micronummer wordt verhoogd, andere regels voor het subnummer en weer andere voor het hoofdnummer. Er is nog geen vaste norm voor deze regels, maar ik behandel hier een methode die met succes door verschillende projecten is gebruikt. U kunt deze methode gewoon overnemen voor uw eigen project. Zelfs als u dat niet doet, is het nog steeds een goed voorbeeld van het soort informatie dat releasenummers zouden moeten uitdrukken. Deze methode is overgenomen van het nummersysteem dat wordt gebruikt door het APR-project; zie http://apr.apache.org/versioning.html.

1. Veranderingen in alleen het micronummer (dat wil zeggen veranderingen binnen dezelfde subreeks) moeten zowel vooruit als achteruit compatibel zijn. Dat wil zeggen dat veranderingen alleen bugfixes zijn of zeer kleine verbeteringen aan bestaande functies. Nieuwe functies zouden niet in een microrelease mogen worden geïntroduceerd.

2. Veranderingen in het subnummer (dat wil dus zeggen binnen dezelfde hoofdlijn) moeten wel achteruit compatibel zijn, maar hoeven niet vooruit compatibel te zijn. Het is normaal nieuwe functies te introduceren binnen een subrelease, maar normaal gesproken niet te veel tegelijk.

3. Veranderingen in het hoofdnummer geven de grenzen van compatibiliteit aan. Een nieuwe hoofdrelease hoeft zowel vooruit als achteruit niet compatibel te zijn. Van een hoofdrelease wordt verwacht dat deze nieuwe functies heeft. Hij kan zelfs een geheel nieuwe set functies hebben.

Wat *achteruit compatibel* en *vooruit compatibel* precies betekenen hangt af van wat uw software doet, maar binnen de context zijn de termen vaak niet geschikt voor al te brede interpretatie. Als uw project bijvoorbeeld een client/servertoepassing is, dan betekent 'achteruit compatibel' dat het upgraden naar 2.6.0 er niet toe leidt dat clients met 2.5.4 functionaliteit verliezen of zich anders gaan gedragen dan voorheen (natuurlijk met uitzondering van opgeloste bugs). Aan de andere kant kan een upgrade naar één van deze clients naar 2.6.0, samen met de server, *nieuwe* functionaliteit beschikbaar stellen aan die client, functionaliteit die clients met 2.5.4 niet kunnen gebruiken. Als dat gebeurt is de upgrade *niet* 'vooruit compatibel': het is duidelijk dat u de client niet kunt downgraden naar 2.5.4 en toch alle functionaliteit behouden die het ook met 2.6.0 had, omdat sommige functionaliteiten nieuw waren in 2.6.0.

Dat is de reden waarom microreleases in principe alleen bedoeld zijn voor bugfixes. Ze moeten compatibel blijven in beide richtingen. Als u upgradet van 2.5.3 naar 2.5.4, vervolgens van gedachten verandert en weer teruggaat naar 2.5.3, dan mag geen functionaliteit verloren gaan. Natuurlijk komen de bugs die in 2.5.4 waren gerepareerd terug na de downgrade, maar u raakt geen functies kwijt, behalve in de gevallen waarin de teruggezette bugs het gebruik van bestaande functies verhinderen.

Client/serverprotocollen zijn slechts één van de vele mogelijke compatibiliteitsgebieden. Een andere is gegevensformaten. Schrijft de software de gegevens weg op een permanent opslagmedium? Als dat het geval is, dan moeten de formaten die het kan lezen en schrijven overeenkomen met de compatibiliteitsrichtlijnen zoals verwacht kan worden uit het releasenummersysteem. Versie 2.6.0 moet in staat kunnen zijn om bestanden te lezen die geschreven zijn met 2.5.4, maar kan het format stilletjes upgraden naar iets dat 2.5.4 niet kan lezen, omdat de mogelijkheid tot downgraden niet vereist is binnen de grenzen van een subnummer. Als uw project codebibliotheken distribueert voor gebruik in andere programma's, dan vormen de API's ook een compatibiliteitsgebied. U moet er voor zorgen dat de compatibiliteitsregels voor de bron- en de binaire code zodanig zijn uitgewerkt dat de geïnformeerde gebruiker zich nooit hoeft af te vragen of het wel of niet veilig is om te upgraden. Hij kan dit direct aan de nummers zien.

In dit systeem krijgt u nooit de kans op een nieuwe start totdat u het hoofdnummer verhoogt. Dit kan vaak heel onhandig zijn. Er kunnen functies zijn die u zou willen toevoegen of protocollen die u opnieuw zou willen vormgeven, maar u kunt dit niet doen omdat u de compatibiliteit moet waarborgen. Er is hiervoor geen eenduidige oplossing, behalve dat u zou moeten proberen de dingen vanaf het eerste begin op een uitbreidbare manier te ontwerpen. (Over dit onderwerp zou een heel boek geschreven kunnen worden, hetgeen absoluut buiten het bestek van dit boek valt.) Het publiceren van beleid over releasecompatibiliteit, en zich daar ook aan houden, is een onmisbaar onderdeel van het distribueren van software. Eén onaangename verrassing kan voldoende zijn om veel gebruikers weg te jagen. De hierboven beschreven methode is voor een deel goed omdat hij nogal wijdverbreid is, maar ook omdat hij makkelijk uit te leggen en te onthouden is, zelfs voor degenen die er niet vertrouwd mee zijn.

Algemeen wordt aanvaard dat deze regels niet gelden voor releases voorafgaande aan de 1.0-release, hoewel uw releasebeleid dat expliciet zou moeten vermelden, al is het alleen maar voor de duidelijkheid. Een project dat zich nog in de eerste ontwikkelingsfase bevindt, kan de releases 0.1, 0.2, 0.3 enz. op rij uitbrengen, tot het moment dat het klaar is voor release 1.0, en de verschillen tussen deze releases kunnen willekeurig groot zijn. Micronummers voor releases voorafgaande aan de 1.0-release zijn optioneel. Afhankelijk van de aard van uw project en de verschillen tussen de releases kan het wel of niet handig voor u zijn de nummers 0.1.0, 0.1.1 enz. te gebruiken. De afspraken over de releases voorafgaande aan release 1.0 zijn niet erg strikt, voornamelijk omdat mensen begrijpen dat strikte regels voor compatibiliteit in de beginfase de ontwikkeling te veel zou hinderen en omdat early adopters *('eerste gebruikers')* sowieso nogal vergevingsgezind zijn.

Vergeet niet dat al deze regels alleen van toepassing zijn op het systeem met drie componenten. Uw project kan met evenzoveel gemak een ander systeem met drie componenten bedenken, of beslissen dat een dergelijke fijne indeling helemaal niet nodig is en een systeem met twee componenten gebruiken. Het belangrijkste is dat u hier in een vroeg stadium over beslist, nauwkeurig publiceert wat de componenten inhouden en u zich er aan houdt.

De even-/onevenmethode

Sommige projecten gebruiken de pariteit van de subcomponent om de stabiliteit van de software aan te geven: even betekent stabiel, oneven betekent instabiel. Dit is alleen van toepassing op het subnummer, niet op de hoofd- en de micronummers. Een verhoging van het micronummer houdt nog steeds een bugfix in (geen nieuwe functies) en een verhoging in het hoofdnummer betekent een grote verandering, nieuwe feature enz.

Het voordeel van dit even-/onevensysteem, dat onder andere werd gebruikt bij het Linux kernel-project, is dat het een mogelijkheid biedt nieuwe functionaliteit uit te brengen om te worden getest, zonder dat andere gebruikers mogelijk met instabiele code moeten werken. Mensen kunnen aan de cijfers al zien dat ze '2.4.21' met een gerust hart kunnen installeren op hun operationele webserver, maar dat ze '2.5.1' beter kunnen bewaren voor experimenten op hun werkstation thuis. Het ontwikkelingsteam behandelt de bugrapporten van instabiele (dus met oneven subnummers) series en als alles na een aantal microreleases wat stabieler is geworden binnen de serie verhogen ze het subnummer (waarmee ze het dus even maken). Ze zetten het micronummer terug op '0' en releasen een vermoedelijk stabiel pakket.

Dit systeem houdt vast aan, of conflicteert in ieder geval niet met, de eerder genoemde richtlijnen voor compatibiliteit. Het voegt alleen wat extra informatie toe aan het subnummer. Dit betekent alleen dat het subnummer ongeveer twee keer zo vaak moet worden verhoogd als anders het geval zou zijn geweest, maar dat kan niet zo veel kwaad. Het systeem met even/oneven subnummers is waarschijnlijk het meest geschikt voor projecten met lange releasecycli en projecten, die door hun aard een groot aantal conservatieve gebruikers hebben die meer belang hechten aan stabiliteit dan aan nieuwe functies. Het is echter niet de enige manier om nieuwe functionaliteiten in de praktijk te testen. Het gedeelte 'Een release stabiliseren'

verderop in dit hoofdstuk beschrijft een andere, misschien vaker gebruikte methode voor het releasen van mogelijk instabiele code bij het publiek. De release wordt van een markering voorzien, waaraan mensen direct kunnen zien of er risico's dan wel voordelen aan zitten.

7.2 RELEASE-BRANCHES

Vanuit het oogpunt van een ontwikkelaar bevindt een open source-softwareproject zich permanent in de releasefase. Ontwikkelaars gebruiken eigenlijk altijd de meest recente code, omdat ze bugs willen opsporen en omdat ze het project van dichtbij genoeg volgen om van mogelijke instabiele gebieden binnen de functies af te blijven. Ze updaten hun versie van de software vaak iedere dag, soms meerdere keren per dag, en wanneer ze een verandering controleren, kunnen ze met redelijke zekerheid stellen dat alle andere ontwikkelaars deze ook binnen 24 uur zullen hebben.

Hoe kan het project dan een formele release uitbrengen? Moet het gewoon een momentopname maken van de software, er een pakketje van maken en het aan de wereld overhandigen als, bijvoorbeeld, versie '3.5.0'? Ons gezond verstand zegt ons dat het zo niet moet. Allereerst is er mogelijk geen enkel moment waarop de hele ontwikkelings-tree schoon is en klaar voor release. Er zijn mogelijk tal van nieuw begonnen functies in diverse stadia van voltooiing. Iemand heeft misschien een grote verandering ingediend om een bug te repareren, maar de verandering is wellicht controversieel en staat ter discussie op het moment dat de momentopname wordt gemaakt. Als dat het geval is, werkt het niet om de momentopname gewoon uit te stellen totdat de discussie is afgelopen. Immers, intussen kan een andere discussie zijn begonnen die daar helemaal los van staat, zodat u ook moet wachten tot *die discussie* weer is afgesloten. Er bestaat geen garantie dat dit proces ooit zal stoppen.

In ieder geval zal het gebruiken van momentopnames van de gehele softwaretree voor releases de lopende ontwikkelingswerkzaamheden in de weg staan, zelfs als de tree kan worden omgezet naar een vorm die geschikt is voor release. Stel dat deze momentopname versie '3.5.0' wordt. De volgende momentopname zou in dat geval '3.5.1' zijn en moet vooral fixes bevatten voor bugs in release 3.5.0. Maar beide zijn momentopnamen van dezelfde tree. Wat moeten de ontwikkelaars dan doen in de tijd tussen de twee releases? Ze kunnen geen nieuwe functies toevoegen: de richtlijnen voor compatibiliteit verbieden dat. Maar niet iedereen loopt warm voor het repareren van de bugs in release 3.5.0. Sommige mensen zullen nieuwe functies hebben die ze proberen af te ronden. Zij raken waarschijnlijk geïrriteerd als ze worden gedwongen te kiezen tussen op hun achterste zitten en niks doen of werken aan dingen waar ze niet in geïnteresseerd zijn, alleen omdat het releaseproces van het project van de ontwikkelingstree verwacht dat deze onnatuurlijk rustig blijft.

De oplossing voor deze problemen is om altijd een *release-branch* te gebruiken. Een release-branch is een aftakking in het versiebeheersysteem (zie *branch*), waarbinnen de code die bedoeld is voor deze release kan worden afgezonderd van de hoofdontwikkelingslijn. Het concept van release-branches is absoluut niet specifiek voor open source-software. Veel commerciële ontwikkelingsorganisaties gebruiken ze ook. In commerciële omgevingen worden release-branches echter soms beschouwd als luxe, een soort formele 'best practice' waar men onder druk van een belangrijke deadline ook best zonder kan omdat het hele team de koppen bij elkaar moet steken om de hoofdlijn te stabiliseren.

Release-branches kunnen echter voor open source-projecten als noodzakelijk worden beschouwd. Ik heb projecten gezien waar niet met release-branches werd gewerkt, maar het gevolg was altijd dat sommige ontwikkelaars met de armen over elkaar zaten terwijl anderen, meestal een minderheid, aan het werk waren om de release de deur uit te krijgen. Dit heeft over het algemeen verschillende negatieve gevolgen. Ten eerste gaat de vaart uit het ontwikkelingsproces. Ten tweede is de kwaliteit van de release onnodig slechter, omdat er maar een paar mensen aan werkten en ze zich haastten om het af te krijgen zodat de rest ook weer aan het werk zou kunnen. Ten derde zorgt het voor een psychologische splitsing in het ontwikkelingsteam, omdat er een situatie ontstaat waarbij de verschillende soorten werkzaamheden elkaar onnodig in de weg lopen. De ontwikkelaars die niks te doen hebben, zouden waarschijnlijk maar al te graag *iets* van hun energie in de release-branch steken, zolang ze daar zelf een keuze in hebben overeenkomstig hun eigen agenda en interesses. Zonder branch kunnen ze alleen maar kiezen voor "Participeer ik vandaag in het project of niet?" in plaats van "Werk ik vandaag aan de release of aan die nieuwe feature die ik heb ontwikkeld voor de hoofdlijn?"

De werking van release-branches

De exacte techniek voor het creëren van een release-branch hangt natuurlijk af van uw versiebeheersysteem, maar het algemene concept is gelijk voor de meeste systemen. Een branch begint meestal vanuit een andere branch of vanuit de trunk ('stam'). De trunk is vanouds de plek waar de hoofdlijn van de ontwikkeling plaats vindt, niet gehinderd door de beperkingen van een release. De eerste release-branch, die leidt naar release '1.0', begint vanuit de trunk. In CVS zou de branch-opdracht er ongeveer zo uit zien:

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_0_X
```

en in Subversion zo:

```
$ svn copy http://.../repos/trunk http://.../repos/branches/1,0.x
```

(Deze voorbeelden gaan uit van een releasenummersysteem met drie componenten. Ik kan hier niet de exacte opdrachten voor alle versiebeheersystemen laten zien, maar ik geef voorbeelden voor CVS en Subversion in de hoop dat de corresponderende opdrachten in de andere systemen uit deze twee kunnen worden afgeleid.)

Merk op dat we een branch '1.0.x' (met een feitelijke 'x') in plaats van '1.0.0' hebben gecreëerd. De reden hiervoor is dat dezelfde sublijn, d.w.z. dezelfde branch, zal worden gebruikt voor alle microreleases binnen deze lijn. Het feitelijke proces van het stabiliseren van een branch voor de release wordt behandeld in het gedeelte 'Een release stabiliseren' verderop in dit hoofdstuk. We houden ons hier alleen bezig

met de interactie tussen het versiebeheersysteem en het releaseproces. Wanneer de release-branch gestabiliseerd en gereed is, is het tijd een momentopname van de branch te labelen:

```
$ cd RELEASE_1_0_X-working-copy
$ cvs tag RELEASE_1_0_0
```

of

```
$ svn copy http://.../repos/branches/1.0.x http://.../repos/
tags/1.0.0
```

Dit label staat nu voor de exacte status van de source tree van het project in release 1.0.0 (dit kan handig zijn wanneer iemand ooit een oude versie nodig heeft nadat de distributiepakketten en binaries zijn weggehaald). De volgende microrelease wordt op dezelfde manier binnen de 1.0.x-branch gemaakt en wanneer deze klaar is wordt een tag gemaakt voor 1.0.1. Vervolgens wordt deze weer opgepoetst voor 1.0.2 enz. Wanneer het tijd is om over een 1.1.x-release te gaan nadenken, maakt u een nieuwe branch vanaf de trunk:

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_1_X
```

of

```
$ svn copy http://.../repos/trunk http://.../repos/branches/1.1.x
```

Het onderhoud voor zowel 1.0.x en 1.1.x kan parallel worden gedaan en releases kunnen onafhankelijk van elkaar vanuit beide lijnen worden uitgebracht. Het is zelfs niet ongewoon dat bijna tegelijkertijd releases worden vrijgegeven uit twee verschillende lijnen. De oudere serie wordt aanbevolen voor meer conservatieve websitebeheerders, die de grote sprong naar (bijvoorbeeld) 1.1 niet willen maken zonder zorgvuldig te zijn voorbereid. Intussen kunnen de meer avontuurlijke types de meest recente release van de hoogste lijn gebruiken, zodat ze zeker weten dat ze de nieuwste functies hebben, ondanks het risico van instabiliteit.

Dit is natuurlijk niet de enige strategie voor het werken met release-branches. In sommige situaties hoeft dit niet eens de beste strategie te zijn, hoewel het wel prima heeft gewerkt voor projecten waarbij ik betrokken ben geweest. U kunt iedere strategie gebruiken die lijkt te werken, maar onthoud de belangrijkste punten: het doel van een release-branch is het releasewerk te isoleren van de veranderingen in de dagelijkse ontwikkeling en het project een fysieke entiteit te geven waaromheen u het releaseproces kunt organiseren. Dit proces wordt in het volgende gedeelte verder uitgewerkt.

7.3 EEN RELEASE STABILISEREN

Stabilisatie is het proces waarbij een release-branch zo wordt bewerkt dat deze klaar is om te worden uitgebracht. Dat wil zeggen dat moet worden besloten welke veranderingen in de release worden opgenomen (en welke niet) en dat de inhoud van de branch dienovereenkomstig moet worden vormgegeven.

Dat ene woord 'besloten' kan een heleboel mogelijke problemen inhouden. De haast op het laatste moment nog functies toe te voegen is een bekend fenomeen bij cooperatieve softwareprojecten. Zodra ontwikkelaars weten dat een nieuwe release nabij is, raffelen ze het afronden van de bestaande veranderingen af om de boot niet te missen. Dat is natuurlijk precies het tegenovergestelde van wat u bij het uitbrengen van een release zou willen. Het zou veel beter zijn wanneer mensen rustig aan functies kunnen werken en zich niet druk hoeven maken of hun veranderingen in deze release worden opgenomen of in de volgende. Hoe meer veranderingen iemand op het laatste nippertje nog in de release probeert te proppen, des te instabieler de code wordt en des te meer nieuwe bugs er (meestal) worden gecreëerd.

De meeste software-engineers zijn het in theorie eens over de globale criteria ten aanzien van welke veranderingen in een releaselijn moeten worden toegelaten tijdens het stabilisatieproces. Uiteraard horen fixes voor ernstige bugs te worden opgenomen, met name bugs die niet op een andere manier kunnen worden omzeild. Updates van de documentatie zijn prima, evenals fixes in de foutmeldingen (behalve als ze deel uitmaken van de interface en stabiel moeten blijven). Veel projecten staan ook bepaalde niet-centrale veranderingen met weinig risico's toe tijdens de stabilisatie en beschikken zelfs over formele richtlijnen om de risico's te meten. Maar het proces kan nog zo geformaliseerd zijn, er is altijd menselijk beoordelingsvermogen nodig. Er zullen altijd situaties zijn waarbij het project gewoon moet beslissen of een bepaalde verandering wel of niet in de release wordt opgenomen. Het risico bestaat dat iedere persoon zijn eigen favoriete veranderingen in de release wil terugzien. Dat betekent dat er veel mensen zijn die veranderingen willen toestaan en maar weinig die ze willen tegenhouden.

Het proces van releasestabilisatie draait dus voornamelijk om het creëren van een mechanisme om 'nee' te zeggen. De oplossing voor open source-projecten in het bijzonder is een manier te vinden om 'nee' te zeggen zonder dat er te veel mensen op hun ziel worden getrapt of ontwikkelaars teleurgesteld zijn, maar die ook niet tot gevolg heeft dat veranderingen die het verdienen te worden opgenomen, uitgesloten blijven. Er zijn verschillende manieren om dit te doen. Het is vrij eenvoudig een systeem te ontwikkelen waarmee aan deze criteria wordt voldaan zodra het team aan deze criteria voldoende belang heeft toegekend. Ik geef hier een beschrijving van de meest populaire systemen aan de beide uitersten van het spectrum, maar laat uw project zich vooral niet ontmoedigen om creatief te zijn. Er zijn vele andere oplossingen mogelijk. Dit zijn er twee waarvan ik heb gezien dat ze in de praktijk goed werkten.

De eigenaar van de release als dictator

De groep komt overeen één persoon aan te wijzen als *eigenaar van de release*.

Deze persoon heeft het laatste woord over welke veranderingen in de release worden opgenomen. Natuurlijk is het normaal en ook te verwachten dat er over wordt gediscussieerd en geargumenteerd, maar uiteindelijk moet de groep de release-eigenaar voldoende autoriteit geven om de laatste beslissing te kunnen nemen. Om dit systeem te laten werken is het noodzakelijk iemand te kiezen met de technische competentie om alle veranderingen te begrijpen en de sociale status en handigheid in de omgang met mensen om de discussies te sturen zonder op al te veel tenen te trappen.

Het is gebruikelijk dat een eigenaar van een release zegt: "Ik denk niet dat er iets mis is met deze verandering, maar we hebben nog niet genoeg tijd gehad om het te testen. Daarom zou het nog niet in deze release moeten worden opgenomen." Het helpt veel als de release-eigenaar brede technische kennis heeft over het project en redenen kan geven waarom een verandering mogelijk destabiliserend zal werken (bijvoorbeeld door interacties met andere delen van de software of problemen met de portabiliteit). Mensen zullen soms verwachten dat dergelijke beslissingen worden beargumenteerd, of ze zeggen dat een verandering niet zo risicovol is als het lijkt. Dergelijke discussies hoeven niet confronterend te zijn, zolang de release-eigenaar in staat is om de argumenten objectief te beoordelen en niet meteen de hakken in het zand zet.

De release-eigenaar hoeft overigens niet perse dezelfde persoon hoeft te zijn als de projectleider (in die gevallen waar er sprake is van een projectleider, zie het gedeelte ‘Vriendelijke dictators’ in Hoofdstuk 4, *Sociale en politieke infrastructuur*). In feite kan het zelfs goed zijn om *niet* dezelfde persoon te nemen. De vaardigheden die een goede ontwikkelingsleider dient te hebben, zijn niet noodzakelijkerwijs dezelfde als die van een goede release-eigenaar. Bij een belangrijk aspect als het releaseproces kan het verstandig zijn iemand te nemen die tegenwicht kan bieden aan de inzichten van de projectleider.

Vergelijk de rol van de release-eigenaar met de minder dictatoriale rol zoals beschreven in het gedeelte ‘Releasemanager’ verderop in dit hoofdstuk.

Stemmen over veranderingen

Diametraal tegenovergesteld aan dictatorschap van de release-eigenaar staat natuurlijk democratie: ontwikkelaars kunnen stemmen over welke veranderingen in de release worden opgenomen. Omdat de belangrijkste functie van releasestabilisatie echter is om veranderingen *uit te sluiten*, is het belangrijk om het stemsysteem zo op te zetten dat er positieve actie van meerdere ontwikkelaars nodig is om een verandering in de release opgenomen te krijgen. Om een verandering op te nemen moet wel meer nodig zijn dan een gewone meerderheid van stemmen (zie het gedeelte ‘Wie mag er stemmen?’ in Hoofdstuk 4, *Sociale en politieke infrastructuur*). Anders zou één stem voor en geen enkele stem tegen een bepaalde verandering voldoende zijn om deze in de release op te nemen. Dat zou kunnen leiden tot de ongewenste situatie dat iedere ontwikkelaar voor zijn eigen veranderingen stemt en niet tegen andermans veranderingen durft te stemmen uit angst voor wraak. Om dat te voorkomen moet het systeem zo worden opgezet dat subgroepen van ontwikkelaars moeten samenwerken om een bepaalde verandering in de release te krijgen.

Dit betekent niet alleen dat meer mensen een verandering moeten reviewen, het zorgt er tevens voor dat een ontwikkelaar minder snel zal aarzelen om tegen een verandering te stemmen, omdat hij weet dat niemand van de voorstemmers zijn tegenstem als een persoonlijke belediging zal voelen. Hoe groter het aantal betrokkenen is, des te meer de discussie gaat draaien om de verandering en niet om het individu.

Het systeem dat we bij het Subversion-project gebruiken, lijkt een goede balans te hebben gevonden. Daarom kan ik dit hier van harte aanbevelen. Om een verandering aan de release-branch te kunnen toevoegen, moeten minimaal drie ontwikkelaars voor hebben gestemd en mag niemand tegen hebben gestemd. Eén enkele ‘nee’-stem is genoeg om te voorkomen dat een verandering wordt opgenomen, dat wil zeggen dat een ‘nee’ binnen de context van releases gelijk staat aan een veto (zie het gedeelte ‘Veto’s’). Natuurlijk moet een dergelijk ‘nee’ beargumenteerd worden. En in theorie kan een veto worden genegeerd als maar genoeg mensen vinden dat het onredelijk is en daarom een bijzondere stemming daarover afdwingen. Dit is in de praktijk echter nog nooit gebeurd en ik verwacht ook niet dat het ooit zal gebeuren. Mensen zijn sowieso conservatief als het op releases aankomt en als iemand dermate overtuigd is van zijn zaak, dat hij een veto uitspreekt over de opname van een verandering in de release, dan is daar meestal een goede reden voor.

Omdat de releaseprocedure met opzet enigszins conservatief is, zijn de argumenten die voor veto’s worden gegeven soms meer procedureel dan technisch van aard. Iemand kan bijvoorbeeld vinden dat een verandering goed is geschreven en dat de kans klein is dat deze voor nieuwe bugs zal zorgen, maar toch tegen opname in een microrelease stemmen gewoon omdat de verandering te groot is, misschien omdat het nieuwe functies toevoegt of omdat het op subtiele wijze voorbijgaat aan de richtlijnen voor compatibiliteit. Ik heb het ook wel eens meegemaakt dat ontwikkelaars hun veto gebruikten omdat ze instinctief aanvoelden dat de verandering nog meer getest zou moeten worden, hoewel ze tijdens de inspectie geen bugs konden ontdekken. Men mopperde hier wel over, maar het veto was al uitgesproken en de verandering werd niet in de release opgenomen (ik kan me echter niet meer herinneren of er later nog bugs zijn gevonden).

Het managen van coöperatieve releasestabilisatie

Als uw project ervoor kiest om over veranderingen te stemmen, dan is het van het allergrootste belang dat het fysieke mechanisme van het sturen van de stembiljetten en het stemmen zo gemakkelijk mogelijk is. Hoewel er genoeg open source-stemsoftware beschikbaar is, is het in de praktijk het makkelijkst om gewoon een tekstbestand te maken in de releasebranch met de naam `STATUS` of `STEMMEN` of iets dergelijks. In dit bestand worden de voorgestelde veranderingen (iedere ontwikkelaar mag een verandering voorstellen voor opname in de release) weergegeven, samen met alle stemmen voor en tegen, plus opmerkingen. (Een verandering voorstellen betekent trouwens niet per definitie dat erover gestemd wordt, hoewel de twee wel vaak samen gaan.) Een entry in zo’n bestand kan er als volgt uitzien:

```
* r2401 (issue #49)
  Voorkomen dat een client/server-handshake twee keer plaatsvindt.
```

```
Argument:
  voorkomt extra turnaround-tijd van het netwerk; kleine ver-
  andering en makkelijk te controleren.
Opmerkingen:
  dit is besproken in http://.../mailing-lists/message-7777.
  html en andere berichten binnen die thread.
Stemmen:
  +1: jsmith, kimf
  -1: tmartin (verbreekt de compatibiliteit met sommige -1.0
      servers, toegegeven, deze servers bevatten nogal wat
      bugs, maar waarom incompatibel zijn als dat niet hoeft?)
```

In dit geval kreeg de verandering twee positieve stemmen, maar werd deze weggestemd door tmartin, die de reden gaf voor zijn veto in een extra opmerking. Het exacte formaat van de entry doet er niet toe. Wat uw project hier ook over afspreekt is prima. Misschien moet de uitleg van tmartin voor zijn veto bijvoorbeeld geplaatst worden onder 'Opmerkingen:'-sectie, of misschien moet de beschrijving van de verandering een kop met 'Opmerkingen:' krijgen om bij de andere secties te passen. Het belangrijkste is dat alle benodigde informatie voor het beoordelen van de verandering bereikbaar is en dat het mechanisme voor het indienen van stemmen zo licht mogelijk is. Er wordt aan de voorgestelde verandering gerefereerd met het revisienummer in de database (in dit geval een enkelvoudige revisie, r2401, hoewel een voorgestelde verandering net zo goed uit meerdere revisies kan bestaan). De revisie wordt verondersteld te refereren aan een verandering aan de trunk. Als de verandering al in de release-branch zou zijn opgenomen, zou het niet nodig zijn erover te stemmen. Als uw versiebeheersysteem geen duidelijke syntaxis heeft voor referenties aan afzonderlijke veranderingen, dan moet het project er een opzetten. Om het stemmen werkbaar te maken, moet iedere verandering waarover gestemd moet worden ondubbelzinnig identificeerbaar zijn.

Degenen die een verandering voorstellen of erover stemmen, moeten ervoor zorgen dat deze zuiver en alleen betrekking heeft op de release-branch, dat wil zeggen zonder conflicten van toepassing is (zie *conflict*). Als er sprake is van conflicten, dan moet de entry of verwijzen naar een aangepaste patch die eenduidig van toepassing is, of naar een tijdelijke branch die een aangepaste versie van de verandering bevat, bijvoorbeeld:

```
* r13222, r13223, r13232
  Herschrijven libsvn_fs_fs's auto-merge algoritme
  Argument:
    onacceptabele prestatie (>50 minuten voor een kleine commit)
    in een repository met 300.000 revisies
  Branch:
    1.1.x-r13222@13517
  Stemmen:
    +1: epg, ghudson
```

Dit voorbeeld is direct uit de praktijk overgenomen. Het komt uit het `STATUS`-bestand voor het releaseproces van Subversion 1.1.4. U ziet hoe het de oorspronkelijke revisies gebruikt als handvaten voor de verandering, hoewel er ook een branch is met een versie van de verandering die is aangepast voor het conflict (de branch combineert ook de drie trunk-revisies in één, r13517, om het makkelijker te maken de verandering in de release in te voegen, als daar goedkeuring voor komt). De oorspronkelijke revisies worden ook gegeven, omdat deze nog steeds het makkelijkst te beoordelen zijn aangezien ze de oorspronkelijke logberichten bevatten. De tijdelijke branch beschikt niet over deze logberichten, om duplicatie van informatie te voorkomen (zie het gedeelte 'Enkelvoudigheid van informatie' in Hoofdstuk 3, *Technische Infrastructuur*) moet het logbericht van de branch voor r13517 alleen 'Aanpassen r13222, r13223 en r13232 voor backport naar 1.1.x-branch' zijn. Alle overige informatie over de veranderingen kan worden opgespoord bij de oorspronkelijke revisies.

Releasemanager

Het feitelijke samenvoegingsproces (zie *merge (ook wel port geheten)*) van goedgekeurde veranderingen in de release-branch kan door iedere ontwikkelaar worden gedaan. Er hoeft niet één persoon te worden aangewezen wiens taak het is de veranderingen samen te voegen. Als er veel veranderingen zijn, is het beter de taak over meerdere mensen te verdelen.

Hoewel zowel het stemmen als het samenvoegen echter op een gedecentraliseerde manier gebeurt, zijn er in de praktijk meestal één of twee mensen die het releaseproces aansturen. Deze krijgt soms de formele titel *releasemanager*, maar dat is heel iets anders dan een release-eigenaar (zie het gedeelte 'De eigenaar van de release als dictator' eerder in dit hoofdstuk) die het laatste woord heeft over de veranderingen. Releasemanagers houden in de gaten over hoeveel veranderingen er op dat moment gestemd wordt, hoeveel er zijn goedgekeurd, hoeveel er waarschijnlijk goedgekeurd gaan worden enz. Als ze bang zijn dat belangrijke veranderingen niet genoeg aandacht krijgen en door een gebrek aan stemmen buiten de release zouden vallen, kunnen ze andere ontwikkelaars voorzichtig aan hun jasje trekken om de verandering te beoordelen en erover te stemmen. Wanneer een cluster van veranderingen is goedgekeurd, zullen mensen vaak zelf de taak op zich nemen om deze in de release-branch te voegen. Het is prima als anderen deze taak aan hen overlaten, zolang iedereen wel begrijpt dat ze niet verplicht zijn al het werk te doen, behalve wanneer ze zich daartoe expliciet hebben verplicht. Wanneer de tijd is aangebroken om de release uit te brengen (zie het gedeelte 'Testen en uitbrengen' verderop in dit hoofdstuk) kan de releasemanager ook de zorg op zich nemen voor de logistiek van het maken van de definitieve releasepakketten, het verzamelen van digitale handtekeningen, het uploaden van de pakketten en de publieke aankondiging.

7.4 HET MAKEN VAN DOWNLOADPAKKETTEN

De meest geaccepteerde vorm voor de distributie van open source-software is als broncode. Dit geldt onafhankelijk van het feit of de software normaal gesproken in de vorm van de broncode draait (d.w.z. dat het kan worden geïnterpreteerd, zoals

Perl, Python, PHP enz.) of dat hij eerst gecompileerd moet worden (zoals C, C++, Java enz.). Met gecompileerde software zullen de meeste gebruikers de broncode waarschijnlijk niet zelf compileren, maar in plaats daarvan de installatie doen met een kant-en-klaar binair pakket (zie het gedeelte 'Binaire pakketten' verderop in dit hoofdstuk). Deze binaire pakketten worden echter nog steeds onttrokken uit distributie van de masterbron. Het doel van het broncodepakket is de release ondubbelzinnig te definiëren. Als het project 'Scanley 2.5.0' distribueert, betekent dit in feite 'de tree van broncodebestanden die, wanneer gecompileerd (wanneer nodig) en geïnstalleerd, Scanley 2.5.0 voortbrengt.'

Er zijn nogal strenge normen voor hoe een broncoderelease eruit zou moeten zien. Zo nu en dan zijn er wel afwijkingen te zien van deze norm, maar dit zijn de uitzonderingen en niet de regel. Tenzij er een zwaarwegende reden is om dit niet te doen, zou ook uw project deze norm moeten hanteren.

Format
De broncode moet worden verzonden in de standaardformaten voor het transporteren van directory trees. Voor Unix en op Unix lijkende besturingssystemen wordt over het algemeen het TAR-format gebruikt, gecomprimeerd met **compress**, **gzip**, **bzip** of **bzip2**. Voor MS Windows is de standaardmethode voor het distribueren van directory trees het *zip*-format. Dit comprimeert de tree ook, zodat het niet meer nodig is om het archief te comprimeren nadat u het hebt gemaakt.

TAR-bestanden
TAR staat voor 'Tape ARchive', omdat het tar-format een directory tree weergeeft als een lineaire gegevensstroom. Daardoor is het ideaal voor het opslaan van directory trees op tapes. Dezelfde eigenschap maakt dit ook de norm voor het distribueren van directory trees in de vorm van een enkelvoudig bestand. Het maken van gecomprimeerde tar-bestanden (ook wel *tarballs* genoemd) is nogal eenvoudig. Op sommige systemen kan de **tar-**opdracht zelf een gecomprimeerd archief produceren; op andere systemen wordt een afzonderlijk comprimeerprogramma gebruikt.

Naam en lay-out
De naam van het pakket moet bestaan uit de naam van de software, plus het releasenummer, plus de format-extensies die bij het archieftype horen. Scanley 2.5.0 bijvoorbeeld, verpakt voor Unix met behulp van GNU Zip (gzip)-compressie, zou er zo uit zien:

scanley-2.5.0.tar.gz

of voor Windows met zip-compressie:

scanley-2.5.0.zip

Beide archieven zouden, wanneer ze uitgepakt worden, een enkele nieuwe directory tree moeten creëren met de naam `scanley-2.5.0` in de huidige directory. Onder de nieuwe directory moet de broncode zijn ingedeeld in een lay-out die gereed is voor compilatie (wanneer compilatie nodig is) en installatie. Op het hoogste niveau van de directory tree moet er een `README`-bestand zijn opgenomen waarin wordt uitgelegd wat de software doet en welke release het is, met verwijzingen naar andere informatiebronnen zoals de website van het project, andere interessante bestanden enz. Bij deze andere bestanden moet een `INSTALL`-bestand zitten, een zusje van het `README`-bestand, waarin voor alle besturingssystemen die het ondersteunt instructies worden gegeven over hoe de software moet worden gebouwd en geïnstalleerd. Zoals eerder gezegd in het gedeelte 'Hoe u een licentie toepast op uw software' in Hoofdstuk 2, *Aan de slag*, moet er ook een bestand zijn met de naam `COPYING` of `LICENSE`, waarin de distributievoorwaarden van de software zijn beschreven.

Ook moet er een `CHANGES`-bestand zijn opgenomen (dat soms ook wel `NEWS` wordt genoemd), waarin wordt uitgelegd wat er nieuw is aan deze release. Het `CHANGES`-bestand is een totaaloverzicht van alle veranderingen van alle releases, in omgekeerd chronologische volgorde, zodat de lijst voor deze release bovenaan in het bestand staat. Het maken van deze lijst wordt meestal als laatste gedaan bij het stabiliseren van een release-branch. Sommige projecten schrijven deze lijst stukje bij beetje tijdens de ontwikkeling, anderen hebben er de voorkeur voor alles voor het laatst op te sparen zodat één persoon deze kan schrijven, op basis van informatie uit de logbestanden van het versiebeheer. De lijst ziet er ongeveer zo uit:

```
Versie 2.5.0
(20 december 2004, van /branches/2.5.x)
http://svn.scanley.org/repos/svn/tags/2.5.0/

Nieuwe functies, verbeteringen:
   * Regular expression queries toegevoegd (issue #53)
   * toegevoegde ondersteuning voor UTF-8- en UTF-16-documenten
   * Documentatie vertaald in het Pools, Russisch en Malagasi
   * ...

Bugfixes:
   * Reindexing bug gerepareerd (issue #945)
   * Enkele querybugs gerepareerd (issues #815, #1007, #1008)
   * ...
```

De lijst kan zo lang zijn als nodig is, maar u hoeft niet te veel moeite te doen door iedere kleine bugfix en functieuitbreiding te vermelden. Het doel is louter om gebruikers een overzicht te geven van wat ze erop vooruit zouden gaan als ze upgraden naar de nieuwe release. In feite wordt de lijst met veranderingen gewoonlijk opgenomen in de aankondigingsmail (zie het gedeelte 'Testen en uitbrengen' verderop in dit hoofdstuk), dus schrijf het met deze doelgroep in het achterhoofd.

CHANGES versus ChangeLog
Vanouds wordt een bestand met de naam *ChangeLog* gebruikt om een overzicht te geven van alle veranderingen die ooit in het project zijn doorgevoerd; dat wil zeggen iedere revisie die is gecommit in het versiebeheersysteem. Er zijn verschillende bestandsindelingen mogelijk voor ChangeLog-bestanden. De details van deze for-

mats zijn hier niet belangrijk, omdat ze allemaal dezelfde informatie bevatten: de datum van de verandering, de auteur en een korte samenvatting (of alleen het logbericht van de verandering).

Een `CHANGES`-bestand is iets anders. Ook dit is een lijst met de veranderingen, maar alleen die veranderingen die belangrijk worden geacht voor een bepaalde doelgroep en vaak met metagegevens zoals de exacte datum en de auteur verwijderd. Om verwarring te voorkomen zou u de termen niet door elkaar moeten gebruiken. Sommige gebruiken 'NEWS' in plaats van 'CHANGES'. Hoewel dit mogelijke verwarring met 'ChangeLog' voorkomt, is het een enigszins verkeerde benaming, omdat het CHANGES-bestand informatie bevat over alle releases en dus ook veel oud nieuws bevat naast het nieuwe nieuws bovenaan de lijst.

ChangeLog-bestanden verdwijnen zo langzamerhand sowieso. Ze waren nuttig in de tijd dat CVS het enige beschikbare versiebeheersysteem was, omdat gegevens over veranderingen niet zo makkelijk uit CVS gehaald konden worden. Bij de meer recente versiebeheersystemen kunnen de gegevens die voorheen werden opgeslagen in ChangeLog op ieder moment worden opgevraagd uit de versiebeheerdatabase, waardoor het zinloos wordt een statisch bestand bij te houden met deze gegevens. In feite is dit zelfs nog slechter dan zinloos, omdat het ChangeLog-bestand alleen de logberichten dupliceert die al in de repository zijn opgeslagen.

De feitelijke lay-out van de broncode binnen de tree moet dezelfde zijn, of in ieder geval zo goed mogelijk lijken op de lay-out van de broncode die te zien zou zijn wanneer het project direct wordt bekeken in de repository van het versiebeheersysteem. Meestal zijn er een paar verschillen, bijvoorbeeld omdat het pakket enkele bestanden bevat die zijn gegenereerd voor de configuratie en de compilatie (zie het gedeelte 'Compilatie en installatie' verderop in dit hoofdstuk), of omdat er externe software in is opgenomen die niet door het project wordt onderhouden, maar die wel vereist is en waarvan de kans klein is dat gebruikers die al hebben. Maar zelfs wanneer de gedistribueerde tree exact overeenkomt met een ontwikkelingstree in de repository van het versiebeheersysteem, zou de distributie zelf geen werkende kopie moeten zijn (zie *werkende kopie*). De release wordt geacht een statisch referentiepunt te zijn, een afzonderlijke, niet wijzigbare configuratie van de bronbestanden. Als het een werkende kopie zou zijn, bestaat het gevaar dat de gebruiker het updatet en achteraf denkt dat hij nog steeds de release heeft, terwijl hij in feite iets anders heeft.

Vergeet niet dat het pakket hetzelfde is, ongeacht de manier van verpakken. De release, dat wil zeggen de exacte entiteit waaraan wordt gerefereerd als iemand het over 'Scanley 2.5.0' heeft, is de tree die wordt gecreëerd door het uitpakken van een zip-bestand of tarball. Het project kan dus de onderstaande bestanden aanbieden om te downloaden ...

scanley-2.5.0.tar.bz2
scanley-2.5.0.tar.gz
scanley-2.5.0.zip

... maar de brontree die wordt aangemaakt door ze uit te pakken moet dezelfde zijn. Die bron-tree vormt de distributie. De vorm waarin deze wordt gedownload is alleen een kwestie van gemak. Bepaalde onbeduidende verschillen tussen bronpakketten kunnen wel worden toegelaten: in het Windows-pakket moeten de regels in tekstbestanden bijvoorbeeld eindigen met CRLF (Carriage Return en Line Feed), terwijl Unix-pakketten alleen een LF gebruiken. De trees kunnen ook worden opgezet met kleine verschillen tussen de bronpakketten voor de verschillende besturingssystemen, indien er voor de compilatie bij deze besturingssystemen verschillende lay-outvormen worden gebruikt. Dit zijn achter allemaal onbeduidende omzettingen. De basis van de bronbestanden moet dezelfde zijn voor alle pakketten van een bepaalde release.

Wel of geen hoofdletters

Wanneer mensen verwijzen naar de naam van een project gebruiken ze daarvoor meestal een hoofdletter, zoals voor een eigennaam gebruikelijk is. Ze gebruiken ook hoofdletters voor acroniemen: 'MySQL 5.0', 'Scanley 2.5.0' enz. Of dit gebruik van hoofdletters ook wordt toegepast in de naam van het pakket is aan het project om te beslissen. Zowel `Scanley-2.5.0.tar.gz` als `scanley-2.5.0.tar.gz` zijn prima (hoewel ik zelf de voorkeur heb voor de tweede, omdat ik mensen zo weinig mogelijk wil dwingen de shift-toets te gebruiken. Veel projecten gebruiken echter wel pakketten met hoofdletters). Het belangrijkste is dat de directory die wordt aangemaakt door het uitpakken van de tarball hoofdletters op dezelfde manier toepast. Er mogen geen verrassingen zijn: de gebruiker moet exact kunnen voorspellen wat de naam van de directory zal zijn die wordt aangemaakt wanneer hij een distributie uitpakt.

Voorreleases

Bij het versturen van een voorrelease of een kandidaat-release is het achtervoegsel een echt deel van het releasenummer. Neem dit dus op in de naam van het pakket. De gesorteerde volgorde van alfa- en bèta-releases zoals eerder genoemd in het gedeelte 'Componenten van releasenummers' zou bijvoorbeeld resulteren in pakketnamen als:

scanley-2.3.0-alpha1.tar.gz
scanley-2.3.0-alpha2.tar.gz
scanley-2.3.0-beta1.tar.gz
scanley-2.3.0-beta2.tar.gz
scanley-2.3.0-beta3.tar.gz
scanley-2.3.0.tar.gz

Het eerste bestand wordt uitgepakt naar een directory met de naam `scanley-2.3.0-alpha1`, de tweede naar `scanley-2.3.0-alpha2` enz.

Compilatie en installatie

Voor software die vanaf de broncode moet worden gecompileerd of geïnstalleerd is er meestal een standaardprocedure die ervaren gebruikers verwachten te kunnen volgen. Voor programma's geschreven in C, C++ of bepaalde andere gecompileerde talen is de norm onder op Unix lijkende systemen bijvoorbeeld dat de gebruiker het

volgende moet intypen:

```
$ ./configure
$ make
# make install
```

De eerste opdracht detecteert automatisch zo veel mogelijk informatie over de omgeving en bereidt deze voor op het build-proces. De tweede opdracht bouwt de software op de juiste plaats (maar installeert deze niet) en de laatste opdracht installeert deze op het systeem. De eerste twee opdrachten worden uitgevoerd als normale gebruiker, de derde als root. Voor meer informatie over het opzetten van dit systeem kunt u het uitstekende boek van Vaughan, Elliston, Tromey en Taylor, *GNU Autoconf, Automake, and Libtool* raadplegen. Het is in gedrukte vorm uitgegeven door New Riders en de inhoud ervan is ook gratis online beschikbaar via http://sources.redhat.com/autobook/.

Dit is niet de enige norm, maar het is wel één van de meest gebruikte. Het build-systeem Ant (http://ant.apache.org/) wordt steeds populairder, met name voor projecten die in Java zijn geschreven. Het heeft eigen standaardprocedures voor bouwen en installeren. Ook adviseren bepaalde programmeertalen, zoals Perl en Python, dezelfde methode te gebruiken voor de meeste programma's die met deze taal zijn geschreven (Perl-modules gebruiken bijvoorbeeld de opdracht **perl Makefile.pl**). Als u niet precies weet welke normen er van toepassing zijn op uw project, vraag dat dan aan een ervaren ontwikkelaar. U kunt er gevoeglijk vanuit gaan dat er in ieder geval *sommige* normen van toepassing zijn, zelfs als u in eerste instantie niet weet welke.

Wat de juiste normen voor uw project ook mogen zijn, wijk daar niet van af tenzij u echt geen andere keuze hebt. Het doorlopen van standaardinstallatieprocedures wordt door veel systeembeheerders tegenwoordig bijna instinctmatig gedaan. Als ze bekende termen zien in het `INSTALL`-bestand van uw project hebben ze er direct al vertrouwen in dat uw project zich algemeen bewust is van de gebruiken en dat de kans groot is dat ook andere dingen kloppen. Zoals besproken in het gedeelte 'Downloads' in Hoofdstuk 2, *Aan de slag* stelt het hebben van een standaard bouwprocedure ook potentiële ontwikkelaars tevreden.

Voor Windows zijn de normen voor bouwen en installeren wat minder vast. Voor projecten waarvoor compilatie nodig is, lijkt het inmiddels gewoonte te zijn om een tree zo aan te bieden dat deze past in het werkruimte-/projectmodel van de standaard-Windows-ontwikkelomgevingen (Developer Studio, Visual Studio, VS.NET, MSVC++ enz.). Afhankelijk van de aard van uw software kan het mogelijk zijn een op Unix lijkende build-optie aan te bieden voor Windows via de Cygwin-omgeving (http://www.cygwin.com/). En natuurlijk zou u, als u een taal of programmeerkader gebruikt dat zijn eigen build- en installatieconventies heeft (bijv. Perl of Python) de standaardmethode moeten gebruiken voor dat kader, of het nu voor Windows, Unix, Mac OS X of een ander besturingssysteem is.

Wees bereid veel extra energie te spenderen om uw project aan te passen aan de van toepassing zijnde build- of installatienormen. Bouwen en installeren is een eerste kennismaking met de software. Het is niet erg als het daarna moeilijker wordt, als dat niet anders kan, maar het zou jammer zijn als de eerste kennismaking tussen de gebruiker of de ontwikkelaar en de software uit onverwachte stappen bestaat.

Binaire pakketten

Hoewel de formele release een broncodepakket is, zullen de meeste gebruikers deze installeren vanaf binaire pakketten, die worden aangemaakt door het softwaredistributiemechanisme van hun besturingssysteem, of eigenhandig kunnen worden verkregen vanaf de website van het project of een externe partij. 'Binair' betekent in dit geval niet per definitie 'gecompileerd'. Het houdt alleen een voorgeconfigureerde vorm in van het pakket waarmee de gebruiker het op zijn computer kan installeren zonder de gebruikelijke build- en installatieprocedures voor de broncode te hoeven doorlopen. Voor RedHat GNU/Linux is dit het RPM-systeem, voor Debian GNU/Linux is dit het APT (`.deb`)-systeem, voor MS Windows zijn het over het algemeen `.MSI`-bestanden of zelf-installerende `.exe`-bestanden.

Of deze binaire pakketten nu worden samengesteld door mensen die direct bij het project zijn betrokken of door derden op afstand, gebruikers zullen ze wel gaan *behandelen* als equivalent van de officiële releases van het project en zullen issues in de bug tracker van het project invoeren die zijn gebaseerd op het gedrag van de binaire pakketten. Daarom is het in het belang van het project om makers van pakketten heldere richtlijnen te geven en nauw met hen samen te werken om erop toe te zien dat wat zij produceren de software eerlijk en nauwkeurig weergeeft.

Het belangrijkste dat pakketmakers moeten weten, is dat ze hun binaire pakketten altijd moeten baseren op een officiële release van de broncode. Soms laten makers van pakketten zich verleiden om een latere vorm van de code uit de repository te halen, of bepaalde veranderingen op te nemen die pas zijn toegevoegd nadat de release is uitgebracht, om gebruikers bepaalde bugfixes of andere verbeteringen te kunnen bieden. De maker van de pakketten denkt dat hij de gebruikers een plezier doet door ze recentere code te geven, terwijl deze manier van werken in feite voor grote verwarring kan zorgen. Projecten zijn erop voorbereid rapporten te ontvangen over bugs die worden gevonden in uitgebrachte releases en in de recente code van de trunk en de hoofd-branch (dat wil zeggen, gevonden door mensen die met opzet niet-uitontwikkelde code gebruiken). Als er van die mensen een bugrapport komt, zal degene die het rapport beantwoordt vaak wel in staat zijn te bevestigen dat de aanwezigheid van de bug bekend is voor die momentopname, en misschien ook dat het sindsdien is gerepareerd en dat de gebruiker moet upgraden of wachten op de volgende release. Als het om een voorheen onbekende bug gaat, maakt het weten van de exacte release het veel eenvoudiger om de bug te reproduceren en te categoriseren in de bug tracker.

Projecten zijn echter niet voorbereid op bugrapporten die gebaseerd zijn op niet-gespecificeerde tussentijdse of hybride versies. Het kan moeilijk zijn deze bugs te reproduceren. Ze kunnen ook het gevolg zijn van onverwachte interacties tussen de geïsoleerde veranderingen die uit een latere ontwikkeling zijn gehaald en die onjuist gedrag van de software veroorzaken waar de projectontwikkelaars niet op

aangekeken zouden mogen worden. Ik heb het zien gebeuren dat verbijsterende hoeveelheden tijd werden verspild omdat een bug *afwezig* was, terwijl deze er wel had moeten zijn. Iemand gebruikte namelijk een enigszins opgelapte versie, gebaseerd op (maar niet identiek aan) een officiële release, en toen de genoemde bug niet tevoorschijn kwam, moest iedereen blijven spitten om uit te zoeken waarom.

En toch zullen er soms situaties zijn waarbij een pakketmaker erop staat dat modificaties aan de release van de broncode doorgevoerd worden. Makers van pakketten zouden moeten worden aangemoedigd om dit met de ontwikkelaars van het project te bespreken en hun plannen voor te leggen. Misschien krijgen ze er toestemming voor, misschien niet. Maar zelfs in dat laatste geval hebben ze het project in ieder geval op de hoogte gesteld van hun bedoelingen, zodat het project kan uitkijken naar ongebruikelijke bugrapporten. De ontwikkelaars kunnen hierop reageren door een disclaimer op de website van het project te plaatsen en de maker van het pakket te vragen hetzelfde te doen op een daarvoor geschikte plaats, zodat gebruikers van het binaire pakket weten dat wat ze krijgen niet precies hetzelfde is als wat het project officieel heeft uitgebracht. Het is in zulke situaties helemaal niet nodig om vijandig te doen, hoewel dat helaas wel vaak gebeurt. Het probleem is dat de doelstellingen van pakketmakers vaak enigszins afwijken van die van ontwikkelaars. Pakketmakers willen voornamelijk een optimaal kant-en-klaarpakket voor hun gebruikers. Natuurlijk willen de ontwikkelaars dat ook, maar zij moeten er ook zeker van zijn dat ze weten welke softwareversies er beschikbaar zijn, zodat ze coherente bugrapporten kunnen ontvangen en garanties kunnen geven ten aanzien van de compatibiliteit. Soms is er een conflict tussen de verschillende doelen. Wanneer dat het geval is, moet u niet vergeten dat het project geen controle heeft over de makers van de pakketten en dat verplichtingen wederzijds zijn. Het klopt dat het project de pakketmakers een plezier doet door de software te produceren. Maar de pakketmakers doen het project ook een plezier door de weinig aantrekkelijk taak op zich te nemen om de software voor een breder publiek beschikbaar te maken, vaak in orde van grootte. Het is geen probleem als u het niet met de pakketmakers eens bent, maar jaag ze niet tegen u in het harnas. Probeer gewoon de best mogelijke oplossing te vinden.

7.5 TESTEN EN UITBRENGEN

Zodra de bron-tarball is gemaakt van de gestabiliseerde release-branch begint het publieke deel van het releaseproces. Maar voordat de tarball aan iedereen beschikbaar wordt gesteld, moet deze worden getest en goedgekeurd door een minimumaantal ontwikkelaars, meestal drie of meer. Deze goedkeuring bestaat uit meer dan alleen het controleren van de release op overduidelijke gebreken. In het ideale geval zullen de ontwikkelaars de tarball downloaden, deze op een schoon systeem bouwen en installeren, en een regressietestpakket erop loslaten (zie het gedeelte 'Geautomatiseerd testen') in Hoofdstuk 8, *Managen van vrijwilligers* en enkele handmatige tests. Ervan uitgaande dat de tarball deze tests goed doorloopt en dat deze ook voldoet aan andere checklistcriteria voor releases die het project heeft, ondertekenen de ontwikkelaars hem vervolgens digitaal met GnuPG (http://www.gnupg.org/), PGP (http://www.pgpi.org/), of een andere programma dat PGP-compatibele handtekeningen kan produceren.

In de meeste projecten gebruiken de ontwikkelaars hun eigen persoonlijke digitale handtekening, in plaats van een gezamenlijke projectsleutel, en er mogen zo veel ontwikkelaars ondertekenen als maar willen. Er is dus alleen een minimumaantal, geen maximumaantal. Hoe meer ontwikkelaars het ondertekenen, des te uitgebreider de release is getest en des te groter de kans dat een gebruiker die zich van de veiligheid van de software bewust is, kan zien hoe digitaal betrouwbaar de tarball is.

De release (dat wil zeggen alle tarballs, zip-bestanden en welke andere formaten er dan ook worden gedistribueerd) moet, zodra deze is goedgekeurd, op de downloadsectie van de website worden geplaatst, voorzien van de digitale handtekeningen en MD5/SHA1-checksums (zie http://en.wikipedia.org/wiki/Cryptographic_hash_function). Er bestaan hiervoor verschillende normen. Eén manier is om ieder releasepakket te voorzien van een bestand met de bijbehorende digitale handtekeningen en een bestand met de checksum. Als een van de uitgebrachte pakketten bijvoorbeeld `scanley-2.5.0.tar.gz` is, zet dan in dezelfde directory een bestand `scanley-2.5.0.tar.gz.asc` met de digitale handtekening voor de tarball, een ander bestand `scanley-2.5.0.tar.gz.md5` met de MD5-checksum en eventueel een derde bestand, `scanley-2.5.0.tar.gz.sha1`, met de SHA1-checksum. Een andere manier om te laten zien dat de release gecontroleerd is, is om alle handtekeningen voor een uitgebracht pakket samen te voegen in één enkel bestand, `scanley-2.5.0.sigs`. Hetzelfde kan met de checksums worden gedaan.
Het maakt in feite niet zo veel uit hoe u het doet. Maak een eenvoudig systeem, beschrijf dit duidelijk en gebruik het consistent voor alle releases. Het doel van deze handtekeningen en checksums is om de gebruikers een middel in handel te geven waarmee ze kunnen verifiëren dat er met de kopie die ze hebben ontvangen niet geknoeid is. Gebruikers zullen deze code op hun computer laten draaien. Als ermee geknoeid is, kan een aanvaller ineens toegang hebben tot al hun gegevens. Zie het gedeelte 'Beveiligingsreleases' verderop in dit hoofdstuk voor meer informatie over paranoia.

Kandidaat-releases
Voor belangrijke releases met veel veranderingen geven veel projecten er de voorkeur aan eerst een *kandidaat-release* uit te brengen, `d.w.z. scanley-2.5.0-beta1` voorafgaande aan `scanley-2.5.0`. Het doel van een kandidaat-release is dat deze uitvoerig kan worden getest voordat deze een officiële release wordt. Als er problemen worden gevonden, worden ze gerepareerd op de release-branch en wordt er een nieuwe kandidaat-release uitgerold (`scanley-2.5.0-beta2`). Deze cyclus wordt doorlopen totdat er geen onacceptabele bugs meer worden gevonden. Op dat moment wordt de kandidaat-release een officiële release. Dat wil zeggen dat het enige verschil tussen de laatste kandidaat-release en de feitelijke release is dat het achtervoegsel uit het versienummer wordt gehaald.

Wat betreft de meeste andere aspecten moet een kandidaat-release hetzelfde worden behandeld als een feitelijke release. De achtervoegsels *alfa*, *bèta* of *rc* is voldoende om conservatieve gebruikers te waarschuwen om te wachten op de echte release. Daarnaast moeten de aankondigingsmails voor de kandidaat-releases natuurlijk vermelden dat het doel van de release is om feedback te verzamelen. Buiten

deze verschillen om moeten de kandidaat-releases evenveel aandacht krijgen als gewone releases. Uiteindelijk wilt u dat mensen de kandidaat-release gaan gebruiken, omdat actief gebruik de beste manier is om bugs te ontdekken en ook omdat u niet zeker weet welke kandidaat-release uiteindelijk de officiële release zal worden.

Releases aankondigen
Het aankondigen van een release is gelijk aan het aankondigen van een andere gebeurtenis en moet de procedures volgen zoals beschreven in het gedeelte 'Publiciteit' in *Hoofdstuk 6, Communicatie*. Er zijn echter een paar specifieke dingen die u voor een release moet doen.

Telkens wanneer u de URL vrijgeeft naar de tarball van de release die gedownload kan worden, moet u ervoor zorgen dat u de MD5/SHA1-checksums en links naar de digitale handtekeningen vermeldt. Omdat de aankondiging op meerdere forums wordt geplaatst (mailinglijsten, nieuwspagina's enz.) kunnen gebruikers de checksums ook vanaf meerdere locaties krijgen, wat zelfs de gebruikers die zich het meest druk maken om de beveiliging extra kan geruststellen dat er niet met de checksums zelf is geknoeid. Het meerdere malen geven van de link naar de bestanden met de digitale handtekeningen maakt deze handtekeningen niet veiliger, maar het stelt mensen wel gerust (met name de mensen die het project niet van dichtbij volgen) dat het project de beveiliging serieus neemt.

Zorg ervoor dat u in de aankondigingsmail en op de nieuwspagina's die meer dan de minimale informatie over de release bevatten ook het betreffende deel van het CHANGES-bestand opneemt, zodat mensen kunnen zien wat ze aan een upgrade zouden kunnen hebben. Dit is net zo belangrijk voor kandidaat-releases als voor de uiteindelijke release. De aanwezigheid van bugfixes en nieuwe functies is belangrijk om mensen over te halen om de kandidaat-release uit te proberen.

Vergeet aan het eind ook niet om het team van ontwikkelaars, de testers en alle mensen die de tijd hebben genomen goede bugrapporten in te dienen te bedanken. Noem echter geen mensen bij naam, behalve wanneer er iemand is die in zijn eentje verantwoordelijk is voor een enorm deel van het werk, waarvan de betekenis door iedereen binnen het project wordt herkend. Ga gematigd om met complimentjes om te voorkomen dat ze aan waarde verliezen (zie het gedeelte 'Bedankjes' in Hoofdstuk 8, *Managen van vrijwilligers*).

7.6 HET ONDERHOUDEN VAN VERSCHILLENDE RELEASELIJNEN

De meeste volwassen projecten onderhouden meerdere releaselijnen, parallel aan elkaar. Nadat 1.0.0 bijvoorbeeld is uitgebracht, gaat die lijn verder met de microreleases (met bugfixes) 1.0.1, 1.0.2 enz., tot het moment dat het project expliciet besluit de lijn te beëindigen. Het uitbrengen van 1.1.0 alleen is overigens niet genoeg om de 1.0.x-lijn te beëindigen. Sommige gebruikers maken er bijvoorbeeld een gewoonte van nooit te upgraden naar de eerste release van een nieuwe sub- of hoofdlijn. Ze wachten eerst tot anderen de bugs uit 1.1.0 hebben gehaald en wachten op 1.1.1. Dit is niet per definitie egoïstisch (vergeet niet dat ze het ook zonder de bugfixes en de nieuwe functies moeten doen), ze willen alleen om wat voor reden dan ook voorzichtig zijn met upgrades. Dus als het project ontdekt dat er een grote bug zit in 1.0.3, vlak voordat release 1.1.0 wordt uitgebracht, het zou een beetje veel gevraagd zijn de bugfix alleen in 1.1.0 op te nemen en alle oude 1.0.x-gebruikers te vertellen dat ze moeten upgraden. Waarom dan niet zowel 1.1.0 als 1.0.4 uitbrengen om iedereen tevreden te houden?

Nadat de lijn 1.1.x een eind op weg is, kunt u het einde van 1.0.x aankondigen. Dit moet officieel worden aangekondigd. De aankondiging kan op zichzelf worden gedaan, of het besluit kan worden vermeld als onderdeel van de aankondiging van release 1.1.x. Hoe u het ook doet, gebruikers moeten weten dat een oude lijn wordt afgebouwd, zodat ze kunnen beslissen over een upgrade.

Sommige projecten geven een bepaald tijdsbestek aan waarin ondersteuning voor de voorgaande release wordt toegezegd. Binnen een open source-project betekent 'ondersteuning' het accepteren van bugrapporten voor die lijn en onderhoudsreleases maken wanneer er belangrijke bugs zijn gevonden. Andere projecten geven geen exacte periode aan, maar houden de binnenkomende bugrapporten in de gaten om in te kunnen schatten hoeveel mensen de oude lijn nog gebruiken. Wanneer het percentage onder een bepaalde waarde komt, wordt het einde van de lijn aangekondigd en stopt de ondersteuning.

Zorg er bij iedere release voor dat u een *doelversie* of *doelmijlpaal* in de bug tracker hebt opgenomen, zodat mensen die bugs indienen dat voor de juiste release kunnen doen. Vergeet niet ook een doelversie op te nemen met de naam 'ontwikkeling' of 'laatste' voor de meest recente ontwikkelingscode, omdat sommige mensen, en niet alleen de actieve ontwikkelaars, vaak vooruit lopen op de officiële releases.

Beveiligingsreleases
De meeste bijzonderheden over het omgaan met beveiligingsbugs zijn behandeld in het gedeelte 'Beveiligingskwetsbaarheden aankondigen' in *Hoofdstuk 6, Communicatie*, maar er zijn nog speciale bijzonderheden voor beveiligingsreleases die hier moeten worden besproken.

Een *beveiligingsrelease* is een release die uitsluitend wordt uitgebracht om een beveiligingskwetsbaarheid op te lossen. De code die de bug oplost, kan niet worden gepubliceerd totdat de release beschikbaar is, wat betekent dat de fixes niet alleen niet in de repository kunnen worden opgenomen tot de datum van de release, maar ook dat de release niet publiek kan worden getest voordat hij de deur uitgaat. Natuurlijk kunnen de ontwikkelaars de fix onderling onderzoeken en de release zelf testen, maar uitgebreid testen in de praktijk is niet mogelijk.

Omdat de release niet voldoende kan worden getest, moet een beveiligingsrelease altijd bestaan uit een bestaande release plus de fixes voor de beveiligingsbug, en *geen andere veranderingen*. Dit is omdat hoe meer veranderingen u opneemt die niet zijn getest, des te groter de kans is dat één daarvan een nieuwe bug zal veroorzaken, misschien zelfs wel een beveiligingsbug! Dit conservatisme is ook bedoeld voor beheerders die de beveiligingsfix wel moeten implementeren, maar wiens be-

leid het is geen andere veranderingen op hetzelfde moment te implementeren.

Het maken van een beveiligingsrelease betekent soms ook dat u misschien wat mensen moet misleiden. Het project heeft bijvoorbeeld al een poosje gewerkt aan release 1.1.3, waarbij een aantal bugfixes voor 1.1.2 al zijn aangekondigd. Op dat moment komt er een beveiligingsrapport binnen. Uiteraard kunnen de ontwikkelaars niet over het probleem praten totdat de fix beschikbaar is gesteld. Tot dat moment moeten ze in het openbaar blijven praten alsof 1.1.3 zo zal blijven als gepland. Maar wanneer 1.1.3 werkelijk uitkomt, zal het enige verschil met 1.1.2 de beveiligingsfix zijn. Alle andere fixes zijn uitgesteld tot 1.1.4 (die dan natuurlijk *ook* de beveiligingsfix bevat, net als alle toekomstige releases).

U kunt een extra component toevoegen aan een bestaande release om aan te geven dat het alleen een verandering in de beveiliging bevat. Mensen zouden bijvoorbeeld aan de hand van de cijfers moeten kunnen zeggen dat 1.1.2.1 een beveiligingsrelease is voor 1.1.2 en dat iedere release met een 'hoger' nummer (d.w.z. 1.1.3, 1.2.0 enz.) dezelfde beveiligingsfixes bevat. Voor mensen die er verstand van hebben, bevat dit systeem veel informatie. Aan de andere kant kan het voor mensen die het project van dichtbij volgen een beetje verwarrend zijn wanneer er meestal een releasenummer met drie componenten is met zo nu en dan een vierde component, naar het lijkt willekeurig. De meeste projecten die ik onder ogen heb gehad, kiezen voor consistentie en gebruiken gewoon het volgende geplande nummer voor de beveiligingsrelease, ook als dat betekent dat alle geplande releases een plaats moeten opschuiven.

7.7 RELEASES EN DAGELIJKSE ONTWIKKELING

Het onderhouden van parallelle releases heeft implicaties voor de dagelijks ontwikkeling. Een bepaalde regel, die sowieso wordt aanbevolen, wordt met name in dit geval een verplichting: zorg ervoor dat iedere commit één enkele logische verandering is en voeg veranderingen die niet met elkaar te maken hebben nooit samen in dezelfde commit. Als een verandering te groot is of te verstorend kan zijn, verdeel het dan over meerdere commits, waarbij iedere commit een duidelijk omlijnde subgroep is van de totale verandering en niets bevat dat niks met de grootschalige verandering te maken heeft.

Dit is een voorbeeld van een niet goed doordachte commit:

```
------------------------------------------------------------------
r6228 | jrandom | 2004-06-30 22:13:07 -0500 (Wed, 30 Jun 2004) |
8 lines

Fix issue #1729: de indexer moet de gebruiker vriendelijk waar-
schuwen wanneer een bestand wordt gewijzigd tijdens het indexeren.

* ui/repl.py
  (ChangingFile): nieuwe uitzonderingsklasse.
  (DoIndex): omgaan met nieuwe uitzondering.

* indexer/index.py
  (FollowStream: nieuwe uitzondering creëren wanneer bestand
  wordt gewijzigd tijdens indexeren.
  (BuildDir: enkele zaken die hier niets mee te maken hebben,
  zoals het verwijderen van enkele overbodige opmerkingen, op-
  nieuw formatteren van code en de foutcontrole tijdens het aan-
  maken van een directory repareren.

Andere poetsacties die hier niets mee te maken hebben:

* www/index.html: enkele typefoutjes herstellen, volgende re-
  leasedatum vaststellen.
------------------------------------------------------------------
```

Het probleem hiervan wordt duidelijk op het moment dat iemand de reparatie van de foutcontrole in `BuildDir` over wil brengen naar een branch voor een aanstaande onderhoudsrelease. Degene die dit doet, wil de andere veranderingen niet overbrengen. De fix voor issue #1729 is bijvoorbeeld misschien niet goedgekeurd voor de onderhouds-branch en de verbeteringen van `index.html` zijn gewoon niet relevant hier. Hij kan de verandering in `BuildDir` alleen niet ophalen via de samenvoegfunctie van het versiebeheersysteem, omdat in het versiebeheersysteem is opgenomen dat de verandering logisch gekoppeld is aan de andere veranderingen die hier geen relatie mee hebben. In feite wordt het probleem al duidelijk vóór de samenvoeging. De verandering vermelden om over te stemmen is al een probleem. In plaats van alleen een revisienummer te geven, moet degene die het voorstel maakt een speciale patch of veranderings-branch maken om het voorgestelde deel van de commit te isoleren. Voor anderen is dit nogal een klus om doorheen te worstelen, en dat allemaal omdat de oorspronkelijke committer niet de moeite heeft genomen om de veranderingen op te delen in logische groepen.

In feite had deze commit moeten worden opgesplitst in *vier* verschillende commits: één om issue #1729 te repareren, een tweede om overbodige opmerkingen te verwijderen en de code te formatteren in `BuildDir`, een derde om de foutcontrole in `BuildDir` te repareren en een vierde om `index.html` op te kalefateren. De derde commit zou dan worden gebruikt als voorstel voor de branch van de onderhoudsrelease.

Natuurlijk is het stabiliseren van de release niet de enige reden waarom iedere commit één logische verandering zou moeten bevatten. Psychologisch gezien is een semantisch homogene commit makkelijker te evalueren en makkelijker terug te draaien indien nodig (in sommige versiecontrolesystemen is reversie sowieso een bijzonder type samenvoeging). Een beetje zelfdiscipline van iedereen op voorhand kan het project in een later stadium veel hoofdbrekens besparen.

Het plannen van releases

Eén gebied waarop open source-projecten altijd al hebben afgeweken van proprietaire projecten is bij het plannen van de release. De deadlines van propriëtaire projecten zijn vaak veel strakker. Soms is dat omdat klanten is beloofd dat er op een bepaalde datum een upgrade beschikbaar zal zijn, omdat de nieuwe release gecoördineerd moet worden met andere activiteiten of marketingacties, of omdat de kapitaalverstrekkers die erin hebben geïnvesteerd resultaten willen zien voordat ze meer geld in het project stoppen. Open source-projecten werden tot voor kort echter gekenmerkt door amateurisme, in de meest letterlijke betekenis van het woord. Ze werden geschreven uit liefde voor het onderwerp, niet uit commerciële overwegingen. Niemand voelde de behoefte om alle functies uit te brengen voordat ze echt klaar waren, en waarom zouden ze ook? Niemands baan zou erdoor op de tocht kunnen komen te staan.

Tegenwoordig worden veel open source-projecten gefinancierd door bedrijven, waardoor ze meer en meer onder invloed komen te staan van de deadlinegerichte bedrijfscultuur. Dit is in veel gevallen positief, maar het kan ook prioriteitsconflicten veroorzaken tussen de ontwikkelaars die ervoor worden betaald en de ontwikkelaars die het werk vrijwillig doen. Deze conflicten steken vaak de kop op rond het issue van wanneer en hoe releases moeten worden gepland. De betaalde ontwikkelaars, die onder druk staan, willen natuurlijk het liefst gewoon een datum prikken voor de release waaraan iedereen zich dan moet aanpassen. De vrijwilligers kunnen een hele andere agenda hebben, misschien functies die ze willen afmaken, of tests die ze eerst willen doen en waarop naar hun mening de release moet wachten.

Natuurlijk is er geen andere oplossing voor dit probleem dan praten en compromissen sluiten. Maar u kunt de frequentie en de ernst van de problemen minimaliseren door het *bestaan* van een bepaalde release los te koppelen van de datum waarop hij de deur uit moet. Probeer dus de discussie in de richting te sturen van welke release het project op de korte en middellange termijn zal uitbrengen en welke functies er in zullen zitten, zonder direct al over een datum te beginnen, behalve voor een ruwe planning met hele grote marges[23]. Door de op te nemen functies al in een vroeg stadium vast te leggen wordt de discussie rond iedere afzonderlijke release minder complex, waardoor de voorspelbaarheid verbetert. Dit creëert ook een soort passief vooroordeel ten opzichte van mensen die een release willen uitbreiden door nieuwe functies of andere complicaties toe te voegen. Als de inhoud van een release goed is gedefinieerd, moet degene die een uitbreiding daarop voorstelt met goede argumenten komen voor die uitbreiding, zelfs als de datum voor de release nog helemaal niet is vastgelegd.

Dumas Malone schreef een uit meerdere delen bestaande biografie van Thomas Jefferson, *Jefferson and His Time*, waarin hij vertelt hoe Jefferson de eerste bijeenkomst leidde waarin de organisatie van de toekomstige universiteit van Virginia moest worden vastgesteld. De universiteit was in de eerste plaats Jeffersons idee, maar (wat overal en altijd het geval is, niet alleen in open source-projecten) er waren al snel anderen bij gekomen, allemaal met hun eigen belangen en agenda's. Op het moment dat ze bij elkaar kwamen voor de eerste vergadering om alles door te spreken, zorgde Jefferson ervoor dat hij zeer nauwkeurig voorbereide bouwtekeningen, gedetailleerde budgetten voor de bouw en operationele kosten, een voorstel voor een onderwijsplan en de namen van de specifieke faculteiten die hij wilde overbrengen vanuit Europa bij zich had. Niemand anders in de zaal was zo goed voorbereid als Jefferson en de groep had in feite niet veel keus dan te schikken naar zijn visie. Uiteindelijk werd de universiteit min of meer overeenkomstig Jeffersons plannen opgericht. Waarschijnlijk wist Jefferson heel goed dat de bouw uiteindelijk veel meer zou gaan kosten dan was gebudgetteerd en dat veel van zijn ideeën om verschillende redenen helemaal niet zouden werken. Zijn plannen waren strategisch.

Hij kwam opdagen bij een bijeenkomst met zoveel materiaal, dat de anderen niets anders konden dan veranderingen voorstellen op zijn materiaal, zodat de algehele vorm, en daarmee de planning van het project, ruwweg zou zij zoals hij het wilde. Bij open softwareprojecten is er geen afzonderlijke 'bijeenkomst', maar in plaats daarvan een reeks kleine voorstellen die voornamelijk via de issue tracker worden ingediend. Maar als u enige kredietwaardigheid hebt opgebouwd, en verschillende functies, verbeteringen en bugs toe gaat wijzen aan releases in de issue tracker overeenkomstig een vooraf uitgebrachte algemene planning, zullen mensen in de meeste gevallen met u meegaan. Zodra u de dingen ongeveer hebt georganiseerd zoals u het wilt, zullen de discussies over de feitelijke *releasedatum* veel soepeler verlopen.

Het is natuurlijk wel erg belangrijk dat u geen enkele individuele beslissing presenteert als een voldongen feit. Nodig mensen in de opmerkingen bij iedere toewijzing van een issue aan een specifieke toekomstige release uit tot discussie en afwijkende opvattingen en wees werkelijk bereid waar mogelijk uw mening te veranderen. Probeer niet de baas te spelen alleen maar om de baas te kunnen spelen. Hoe meer anderen zijn betrokken bij de planning van het releaseproces (zie het gedeelte 'Zowel de managementtaken als de technische taken delen met anderen' in Hoofdstuk 8, *Het managen van vrijwilligers*), des te makkelijker het zal zijn hen over te halen mee te gaan in uw prioriteiten voor issues die voor u belangrijk zijn.

Een andere manier om de spanning rond het plannen van releases te verminderen, is zeer regelmatig nieuwe releases uitbrengen. Wanneer er veel tijd zit tussen verschillende releases is het belang van iedere afzonderlijke release voor mensen veel groter. Ze zijn nog veel meer teleurgesteld als hun code niet wordt opgenomen in een release, omdat ze weten hoe lang het kan duren voordat ze een volgende kans krijgen. Afhankelijk van de complexiteit van het releaseproces en de aard van het project is een periode van drie tot zes maanden een goede interval tussen releases, hoewel onderhoudslijnen wat vaker microreleases kunnen uitbrengen, als daar behoefte aan is.

23| Voor een alternatieve benadering kunt u het proefschrift van Martin Michlmayr Ph.D. nalezen: *Quality Improvement in Volunteer Free and Open Source Software Projects: Exploring the Impact of Release Management* (http://www.cyrius.com/publications/michlmayr-phd.html). Dit proefschrift gaat over tijdgestuurde releaseprocessen, in tegenstelling tot releases op basis van functies, binnen grote open source-softwareprojecten. Michlmayr heeft ook voor Google een lezing gehouden over dit onderwerp, dat beschikbaar is via Google Video op http://video.google.com/videoplay?docid=-5503858974016723264.