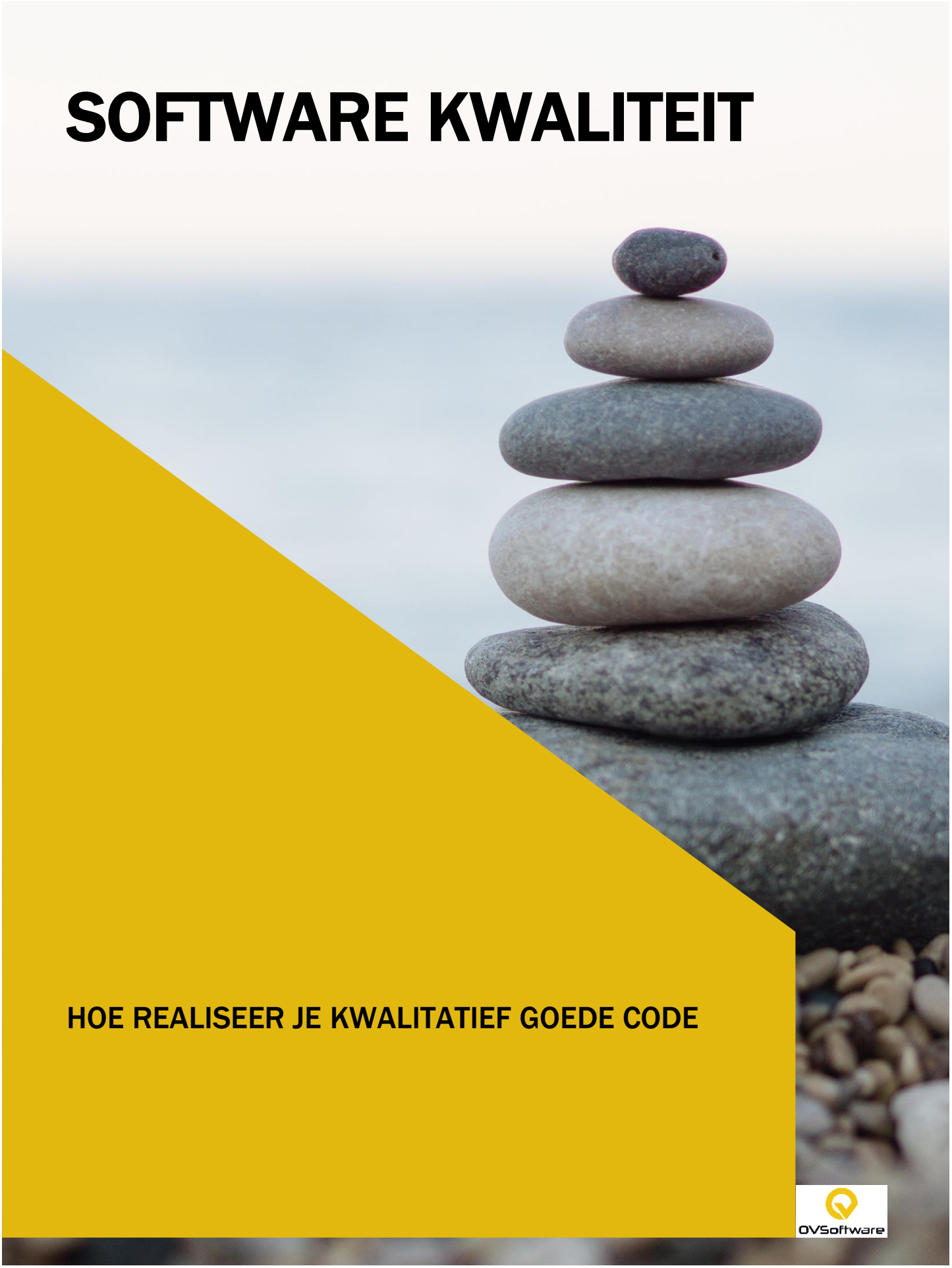
SOFTWARE KWALITEIT



HOE REALISEER JE KWALITATIEF GOEDE CODE

OVSoftware

SOFTWARE KWALITEIT

Een hoge kwaliteit van maatwerksoftware is belangrijk, maar niet vanzelfsprekend. De kwaliteit van de ontwikkelde software bepaalt de kosten voor onderhoud en aanpassingen en in welke mate je risico loopt op uitval en storingen.

Het belang van codekwaliteit voor maatwerkoplossingen

Voor veel processen en automatiseringsvraagstukken zijn standaard software applicaties beschikbaar waar een gemiddeld bedrijf of organisatie prima mee uit de voeten kan. Specifieke vraagstukken vragen echter om een specifieke aanpak en dito oplossingen. Voor de realisatie van een maatwerkoplossing is een maatwerkproces nodig. Dit proces vormt een belangrijke basis voor een succesvolle uitvoering. Hierbij worden namelijk noodzakelijke randvoorwaarden gerealiseerd. Ook moet er een eenduidig beeld worden vastgesteld over de aanpak en verwachte resultaten. Bij de realisatiefase speelt de kwaliteit van het eindproduct een essentiële rol. Zowel voor het op te leveren product als voor de tevredenheid van de klant op lange termijn. Daarbij is het belangrijk om transparant te zijn over hoe je ervoor zorgt dat de kwaliteit van de softwarecode aan de hoogste eisen voldoet.

Waarom is codekwaliteit belangrijk en niet vanzelfsprekend?

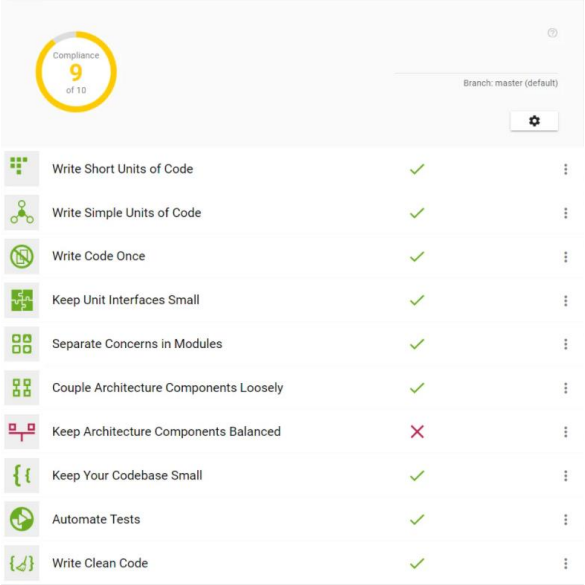
Hoge kwaliteit van de software die je ontwikkelt is belangrijk voor de lange termijn relatie met klanten. Daarmee zijn klanten niet alleen op het moment van de oplevering tevreden, maar dan blijven ze dat ook. Wanneer softwarecode kwalitatief goed is opgebouwd en gedocumenteerd, kunnen aanpassingen achteraf eenvoudiger worden doorgevoerd.

Daarnaast kost het reguliere onderhoud beduidend minder dan de software waarvan de kwaliteit te wensen over laat. Kwalitatief hoogwaardige softwarecode beperkt daarnaast het risico op uitval en storingen, wat de continuïteit van bedrijfsprocessen zekerstelt. Voor de klant betekent dit een kostenbesparing over de gehele levensduur van een applicatie.

Kwalitatief goede softwarecode is niet vanzelfsprekend. Als je hier niet vanaf het eerste moment de nodige aandacht aan besteedt, ontstaat complexe software die je achteraf alleen met veel extra inspanning kwalitatief goed kunt krijgen. Onder druk van deadlines en het ontbreken van voldoende budget wordt deze stap dan ook vaak overgeslagen. Het gevolg hiervan is dat het eindproduct zo goed mogelijk functioneert, maar de kosten voor beheer en doorontwikkeling hoger uit zullen vallen dan nodig is.

SIG Richtlijnen

Bij de ontwikkeling van maatwerk softwareoplossingen maken wij gebruik van de tien richtlijnen voor codekwaliteit van de Software Improvement Group (SIG).



Deze zijn in lijn met de ISO 25010 standaard. Een door TÜViT1*) gecertificeerde tool is beschikbaar om te toetsen of aan de richtlijnen wordt voldaan. Door deze kwaliteitsrichtlijnen onderdeel te maken van de criteria waaraan elk stukje functionaliteit moet voldoen, zorgen we ervoor dat elk stukje functionaliteit pas gereed is als de kwaliteit van de code voldoet aan de richtlijnen.

De voordelen:

- Hoge betrouwbaarheid (door beperken van complexiteit)
- Betere onderhoudbaarheid
- Lagere total cost of ownership
- Sneller resultaten
- Betere samenwerking met de developers van onze klanten.

Code reviews en documentatie

Naast het feit dat de software aan de SIG-richtlijnen moet voldoen, is de standaard om ook het vier-ogen-principe te gebruiken om ervoor te zorgen dat er aan de coding-standaarden wordt voldaan. Dit betekent dat elk stukje functionaliteit door een collega wordt bekeken en wordt gevalideerd. Daarnaast moet je zorgen dat relevante documentatie wordt bijgewerkt naar aanleiding van gerealiseerde functionaliteit. De check op SIG richtlijnen moet een vast onderdeel zijn van de ‘Definition of Done’ (DoD).

Testen

De correcte werking van elk stukje code wordt zeker gesteld door de software componenten getest worden. Daarmee voorkom je bovendien dat bij latere aanpassingen regressie optreedt, ofwel dat de functionaliteit onbedoeld omvalt.

Naast het testen van de losse onderdelen, moet de gehele applicatie ook functioneel worden getest en geaccepteerd. De klant bepaalt uiteindelijk of de geleverde oplossing goed genoeg is. Daarom speelt zij ook een cruciale rol bij de uitvoering van deze testen.

Agile en transparant

In de realisatiefase is Agile/Scrum tegenwoordig een gangbare werkwijze. Na elke sprint vindt een review van gerealiseerde functionaliteit plaats en wordt de prioriteit van nog te realiseren functionaliteiten en eventuele aanpassingen bepaald. Naar aanleiding van een review kan er worden besloten om te stoppen of om door te gaan. Voor elke nieuwe sprint wordt bepaald welke functionaliteit zal worden gerealiseerd, zodat het voor iedereen duidelijk is wat ze kunnen verwachten. Het gehele proces wordt bij OVSoftware begeleid door ervaren, technische consultants en functionele Lean/Scrum-gecertificeerde consultants.

De Agile-werkwijze voor softwareontwikkeling vraagt ook iets van klanten. Met name een goede invulling van de rol: 'Product Owner' is essentieel om tot het gewenste eindproduct te komen. Vaak wordt deze rol echter als lastig ervaren. Om deze reden ondersteunen OVSoftware consultants deze rol bij de meeste projecten of nemen ze deze deels over.

Gedurende het gehele proces dient de klant conform de Scrum-methodiek betrokken te worden bij de voortgang. Hierdoor is er na elke sprint sprake van een opgeleverd en werkend product. Door middel van rapportages kan er transparantie geboden worden over de resultaten en de kwaliteit van de software. De kwaliteitsscore van de software is onderdeel van deze rapportage.

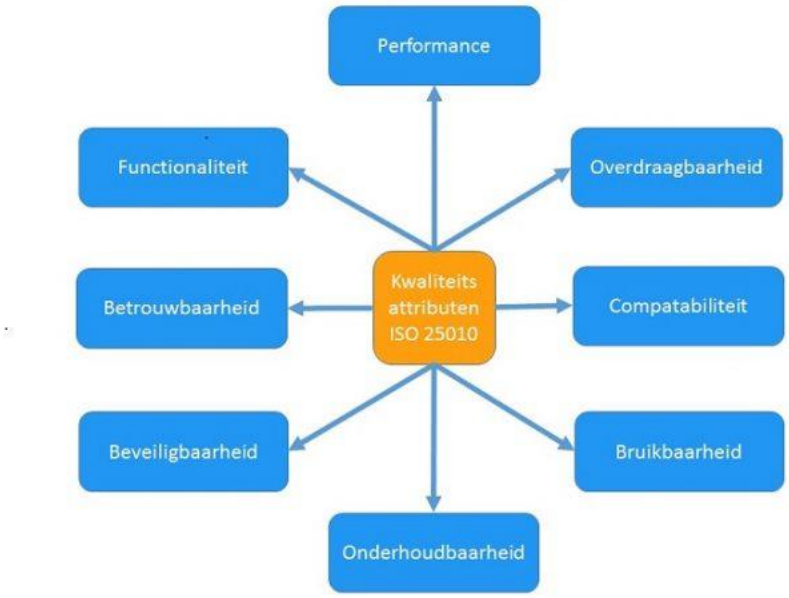
Toepassing standaard devops-componenten

Bij het bouwen van goede software gaat het om waarde toevoegen aan (primaire) processen. OVSoftware maakt gebruik van bestaande functionaliteit en services binnen de Azure DevOps-omgeving. Aan deze omgeving heeft OVSoftware daarnaast eigen, veel gevraagde functionaliteit toegevoegd in de vorm van standaard componenten die hun kwaliteit reeds bewezen hebben in andere oplossingen. Hierdoor kunnen wij ons volledig concentreren op de specifieke functionaliteit die voor de individuele klant wordt gerealiseerd.

De 10 SIG Richtlijnen in detail

ISO25010

De ISO-norm 25010 beschrijft de kwaliteitskenmerken van software en systemen. Daarbij zegt het iets over de kwaliteit van het product en de geschiktheid voor gebruik. Deze vallen uiteen in een aantal componenten:



SIG heeft deze kwaliteitskenmerken geaccepteerd en deel gemaakt van haar 10 software kwaliteitsrichtlijnen:

1. Write short units of code
2. Keep architecture components balanced
3. Write code once
4. Keep your codebase small
5. Keep unit interfaces small
6. Automate tests
7. Write simple units of code
8. Separate concerns in modules
9. Couple architecture components loosely
10. Write clean code

In de volgende hoofdstukken wordt dieper ingegaan op deze 10 richtlijnen.

1. Schrijf kleine functies

‘write short units of code’

Schrijf kleine functies – wat houdt dat in?

De eerste richtlijn in deze serie gaat over ‘write short units of code.’ Het geeft aan dat je bij het schrijven van functies erop moet letten dat je het aantal coderegels beperkt houdt. Divers onderzoek van de SIG heeft uitgewezen dat de menselijke hersenen maar een beperkt aantal regels kunnen bevatten. Is er sprake van te veel regels, dan kun je het niet meer overzien en begrijpen. En code wordt nu eenmaal door mensen geschreven en onderhouden. Uit diverse studies en vergelijkingen van onder meer de SIG, blijkt dat 15 coderegels het optimale aantal is voor een mens om te bevatten. Er is zelfs een rekenmodel op losgelaten waardoor men deze conclusie met concrete cijfers heeft kunnen onderbouwen.

De richtlijn toegepast in de praktijk

Onderhoud aan software kun je vergelijken met onderhoud aan je auto; door intensief gebruik, voortschrijdende ontwikkelingen (inzicht) en veranderende wet- en regelgeving moet je onderdelen van de auto aan blijven passen en vervangen voor een optimale performance. Om vergelijkbare redenen moet ook software worden onderhouden. Daarbij is het prettig dat dit op een efficiënte manier kan gebeuren. Een voorbeeld hiervan is een opdracht waarin wij werken aan het onderhoud van legacy code die zo’n 10 tot 12 jaar geleden is ontwikkeld. Daarin kom met grote regelmaat veel te grote functies voor met soms wel meer dan 30 regels. Daarvan merken wij dat het een stuk lastiger is en dat het meer tijd vraagt om mee te werken dan wanneer de functies kort zijn. Het verkorten van code om software beter onderhoudbaar te maken, doe je door bepaalde regels code uit de grote functie te halen (extracten) en onder te brengen in een nieuwe functie. Visual studio kan dat automatisch doen. De aanpassing op zich is dus eenvoudig, maar het vraagt om discipline om dit structureel door te kunnen voeren.

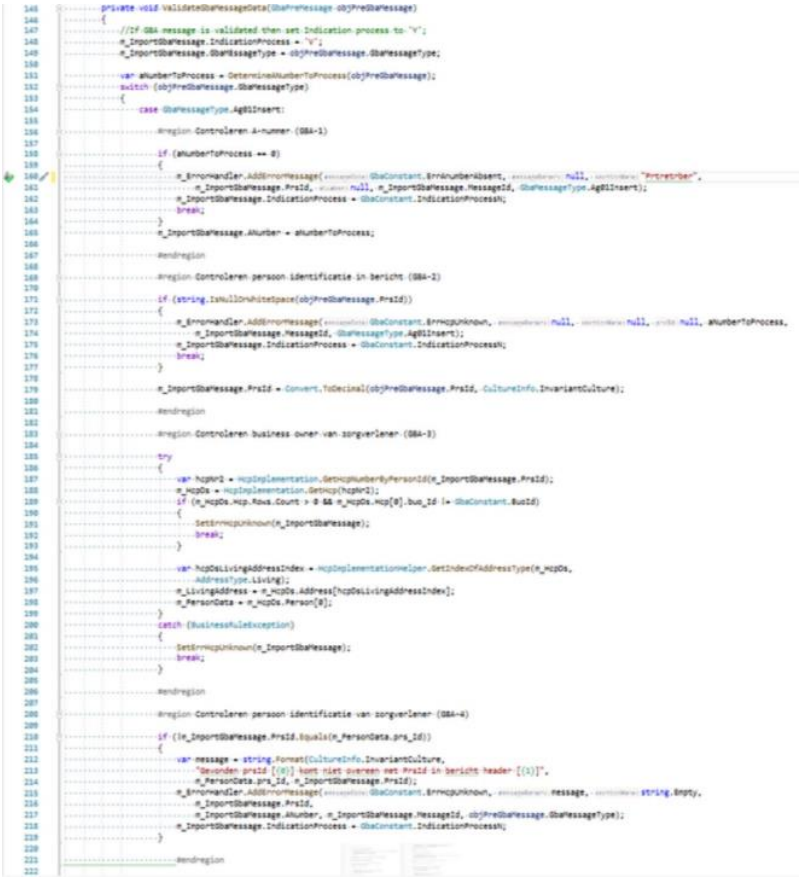
Dit levert de richtlijn op voor de klant

Al is het toepassen van deze richtlijn nog zo belangrijk, de klant ziet er in eerste instantie niets van. Wanneer je op slechte banden rijdt zal ook niet iedereen dit zien, maar de kans op een ongeluk is groter. Veel schade en kosten zijn hiervan gevolg. Zo is het ook met te lange functies die door de complexiteit de kans op fouten vergroten en de herstelkosten verhogen. Je wilt niet dat belangrijke software slecht(er) functioneert of dat er door een beveiligingslek gegevens op straat komen te liggen. Ook deurtjes die door slimme hackers kunnen worden opengezet, wil je zo snel mogelijk weer dicht doen. Kleine functies schrijven maakt de software inzichtelijker waardoor iedereen het snapt en ermee kan werken. Het maakt het ook eenvoudiger om onderhoud en aanpassingen door wisselende personen uit te laten voeren omdat de opbouw voor zich spreekt en goed ‘behapbaar’ is. Op termijn leveren kleine functies tijd en geld op.

Zo helpt de richtlijn de ontwikkelaar

Lazko Kessels – IT Professional OVSoftware

"Ik had nooit gedacht dat het maximum aantal regels met een vast getalletje aangegeven zou kunnen worden. Natuurlijk is 50 regels te veel, maar dat 15 dan het specifieke streefgetal is, had ik niet zelf kunnen bedenken. Maar het is fijn om te weten.



Ik heb genoeg lange functies gezien die onmogelijk waren om te doorgronden. Het is niet alleen onoverzichtelijk, maar bij te lange functies kun je ervan uit gaan dat er dan ook andere dingen niet goed zijn. De lengte van code heeft namelijk ook effect op de werking van de architectuur; bij lange code gebeurt er te veel in een specifieke functie, terwijl bepaalde dingen misschien veel beter op een andere plek zouden passen. Om nog even in het voorbeeld van de auto's te blijven; stel dat één en hetzelfde motortje de wielen én de ruitenwissers aanstuurt. Rijdt de auto snel, dan gaan de ruitenwissers snel, zelfs wanneer de zon schijnt. Het motortje dat de ruitenwissers beweegt moet in dit geval natuurlijk anders zijn dan het motortje die de wielen in gang zet. Voor coderegels in functies geldt hetzelfde. De richtlijn van maximaal 15 regels per functie geeft je als developer houvast. Getallen zijn voor technici hartstikke fijn: ze zijn duidelijk, logisch, het is goed of niet goed, ja of nee. Ik word er in elk geval blij van."

2. Hou je code componenten in balans

'keep architecture components balanced'

Code componenten in balans houden – wat betekent dat?

Een code component is een aantal pagina's met code bij elkaar die verschillende dingen doen. Neem ruitenwissers, die kunnen aan en uit staan, langzaam, snel of extra snel gaan of bewegen met interval. Het snel bewegen en extra snel bewegen zouden qua code in één pagina staan. De getrapte beweging is ook een pagina op zich. Alle pagina's, of 'klassen' zoals we dat in techtaal noemen, gaan over hetzelfde onderwerp en vormen samen het component. Het in balans houden van code componenten doe je door hen onderling niet te veel van elkaar te laten verschillen qua omvang. Door de hoeveelheid code per component zo goed als gelijk te houden en er goed over na te denken welke onderdelen je bij elkaar zet en welke niet. Goed groeperen dus. Als je alles op een onlogische manier in één grote en drie hele kleine componenten zou zetten, dan heb je onbalans. Deze onbalans is slecht voor de overzichtelijkheid van de code. De richtlijn dat je code componenten in balans houdt, is niet heel strikt, maar het is een guideline die je bij het ontwikkelen in je achterhoofd houdt.

De richtlijn toegepast in de praktijk

Als de balans in de code ver te zoeken is ben je veel tijd kwijt om te achterhalen wáár in de code je moest zijn. Zijn de componenten wel in balans? Dan kun je veel sneller en makkelijker achterhalen waar de specifieke code zit die je nodig hebt om bijvoorbeeld een bug op te lossen. Als je één stuk code aanpast, kun je in theorie ook andere stukjes code beïnvloeden. Het is dus goed om te weten welke stukken code elkaar raken. Als de componenten klein en logisch gegroepeerd zijn, is dat ook veel eenvoudiger vast te stellen. Volgens deze richtlijn ligt het ideale aantal componenten rond de negen. Dit blijkt uit het onderzoek van SIG.

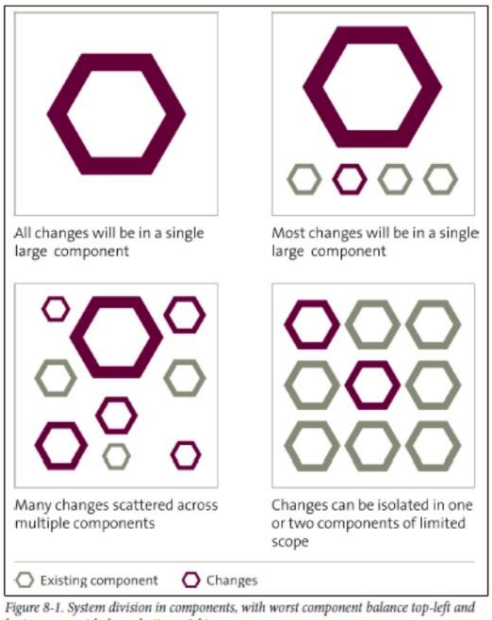


Figure 8-1. System division in components, with worst component balance top-left and best component balance bottom-right

Dit levert de richtlijn op voor de klant

Ook deze richtlijn heeft impact op de onderhoudbaarheid van software. Het zorgt ervoor dat onderhoud minder tijd en geld kost; dat je sneller in kunt grijpen als het fout gaat waardoor de applicatie weer sneller beschikbaar is. Als je code componenten in balans zijn, kun je vlot traceren waar de problemen zitten, preciezer ingrijpen en sneller corrigeren. Als de componenten niet te groot zijn, wordt testen ook minder complex. Je hoeft alleen díe specifieke kleine delen te testen waar het om gaat en de delen die mogelijk beïnvloed zijn. In plaats van alle delen weer opnieuw door de molen te halen. Als je een nieuw onderdeel van de defecte ruitenwisser hebt geplaatst, ga je toch ook de bandenspanning niet checken?

Zo helpt de richtlijn de ontwikkelaar

Lazko Kessels – IT Professional OVSoftware

"Het naleven van deze richtlijn zorgt ervoor dat je als developer niet eindeloos aan het zoeken bent in enorme brokken code voordat je problemen kunt achterhalen en repareren. Ook het testen kun je beperken tot de specifieke delen die direct of indirect geraakt zijn. Dit scheelt gewoon veel tijd en vaak ook ergernis. Deze richtlijn van negen componenten is geen wetmatig getal. Er zit bewegingsvrijheid in, maar het is desalniettemin een goede check om uit te voeren."

3. Schrijf code één keer

'write code once'

Schrijf code één keer – wat houdt dat in?

Deze richtlijn gaat over het eenmalig schrijven van code die je vervolgens op meerdere plaatsen kunt hergebruiken. Dat doe je niet door het stukje code keer op keer te kopiëren. Zodra je een stuk code ergens opnieuw wilt gebruiken, haal je dit specifieke deel uit een eerder geschreven functie en maak je hier een nieuwe functie van. Een functie is maximaal 15 regels code die ervoor zorgt dat een bepaalde actie wordt uitgevoerd. Vervolgens verwijs je op elk punt waar je deze code wilt gebruiken naar de nieuwe functie. Op deze manier kun je de functionaliteit hergebruiken, zonder de code te kopiëren.

De richtlijn toegepast in de praktijk

Zodra je een bestaande functionaliteit wilt hergebruiken, pas je de boven beschreven richtlijn toe. Er is echter pas sprake van gekopieerde code, wanneer het om minimaal 6 regels code gaat. Er zijn ook projecten waar gekopieerde code is blijven staan. Dat gebeurt bijvoorbeeld als developers de werking van een stuk code snel willen testen en achteraf vergeten om deze code in een nieuwe functie te zetten. Gelukkig kunnen we gebruikmaken van de BCH (Better Code Hub) van de SIG, een automatische tool die continue checkt of de ontwikkelde code aan de 10 SIG richtlijnen voor codekwaliteit voldoet*. Bij elke codewijziging zie je direct op een schaal van 1 tot 10 aan hoeveel richtlijnen je voldoet en kun je direct je code aanpassen om de hoogste score te realiseren. BCH checkt ook op dubbele code zodat je direct ziet of je codewijziging gekopieerde code bevat.

Dit levert de richtlijn op voor de klant

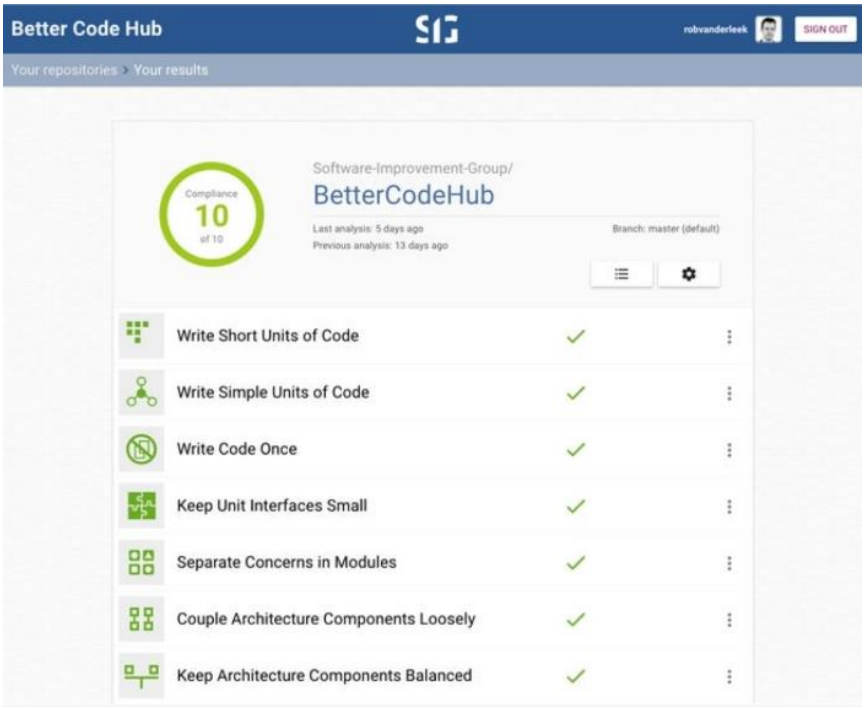
Het eenmalig schrijven van code en deze hergebruiken in plaats van kopiëren, resulteert in makkelijker onderhoudbare code. Tijdens het ontwikkelen zelf kan het iets meer werk zijn, omdat je code uit de functie moet halen. En je moet het als een aparte functie opslaan om deze vervolgens weer op een nieuwe plek te kunnen gebruiken. Daar staat tegenover dat wanneer er een probleem in dit stukje code wordt ontdekt, je maar op één plek wijzigingen door hoeft te voeren. Je bent dus veel minder tijd kwijt met het zoeken en aanpassen van code. Het toepassen van de 10 richtlijnen is niet in alle gevallen even makkelijk aan een opdrachtgever uit te leggen, maar in dit geval is het niet moeilijk om te begrijpen dat de initiële investering zich al snel terugverdient.

Zo helpt de richtlijn mij als ontwikkelaar

Sylvia Straub – IT Professional OVSoftware

"De richtlijn zorgt ervoor dat er uiteindelijk minder regels code worden geschreven en dat heeft een aantal voordelen, ook voor mij als software developer. Als je nieuw bent binnen een project is het werk overzichtelijker en snap je sneller waar het over gaat met behulp van deze richtlijn. Daardoor kun je zelf ook eerder een bijdrage gaan leveren.

Doordat eventuele fouten maar op één plek optreden en niet (meer) in gekopieerde code, kost het mij minder tijd om eventuele fouten in de code te vinden en te corrigeren. Daardoor hou ik meer tijd over om nieuwe functionaliteit te ontwikkelen en implementeren en dat is uiteindelijk toch wel het leukste aspect van mijn werk. Wanneer de door jou ontwikkelde applicatie daadwerkelijk in de praktijk wordt gebruikt en ook nog eens goed wordt ontvangen, dan geeft dat echt veel voldoening. Voor een van onze klanten heb ik bijvoorbeeld een bijdrage mogen leveren aan www.rijksoverheid.nl – als je bedenkt hoeveel mensen er dan uiteindelijk gebruikmaken van de stukken code die jij hebt ontwikkeld, dan is dat best bijzonder."



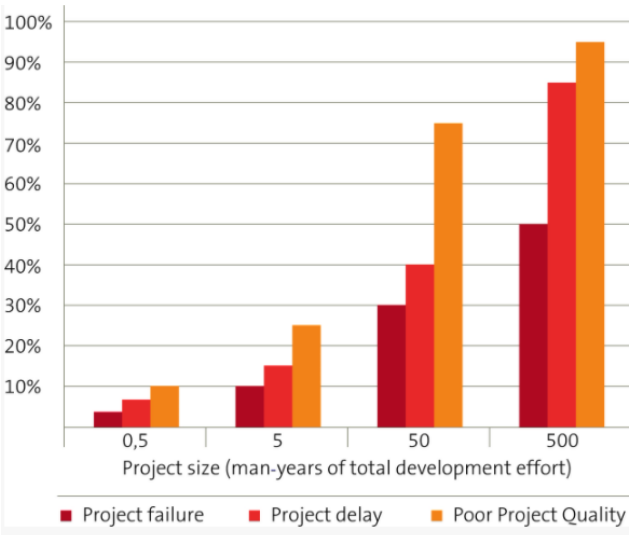
Better Code Hub checkt ontwikkelde code om te zien of deze voldoet aan de 10 benchmark richtlijnen voor kwalitatieve softwareontwikkeling. Het geeft direct feedback waar eventuele aanpassingen nodig zijn om maximale kwaliteit te kunnen realiseren. BCH ondersteunt 17 verschillende moderne programmeertalen.

4. Hou de codebasis klein

'keep your codebase small'

Keep Your Codebase Small – wat houdt dat in?

Het klein houden van je codebase is iets heel anders dan de richtlijn: 'Schrijf kleine functies.' Waarin wordt aangegeven dat je maximaal 15 regels per functie schrijft. Een codebase is een verzameling broncode die wordt gebruikt om een bepaald softwaresysteem, applicatie of softwarecomponent te bouwen. De bekende uitspraak: 'Less is more' geldt ook zeker voor softwareontwikkeling. Hoe kleiner de hoeveelheid code, hoe makkelijker het is om iets terug te vinden en om de software te onderhouden. Sommige applicaties/codebases bestaan uit grote hoeveelheden code, omdat voor elke nieuwe functionaliteit honderden regels code zijn geschreven. Dat maakt het geheel al snel complex. In dit soort applicaties moet je lang zoeken voor je de code vindt die een specifieke functionaliteit aanstuurt en kost het bovendien veel tijd om op een goede manier de nodige correcties aan te brengen. Door de onoverzichtelijkheid is bovendien het risico dat er iets fout gaat groter. Als je de code splitst in verschillende codebases die allemaal verantwoordelijk zijn voor hun eigen functionaliteit en zorgt dat ze zelfstandig kunnen functioneren, dan is de software als geheel goed beheer(s)baar en te onderhouden.



Source The Economics of Software Quality by Capers Jones and Olivier Bonsignour (Addison-Wesley Professional 2012)

De richtlijn toegepast in de praktijk

Om ervoor te zorgen dat elke codebase klein blijft, maak je telkens de afweging wat het doel en functionaliteit ervan is. Je stelt jezelf bij elke nieuwe functionaliteit de vraag welke waarde deze toevoegt voor de business of voor de gebruiker. Als de toegevoegde waarde van de functionaliteit groot is ten op zichtte van de hoeveelheid code die ervoor nodig is, dan wordt deze toegevoegd. Maar als de toegevoegde waarde relatief klein is

ten opzichte van de hoeveel code die nodig is, dan wordt er in overleg met de klant besloten of deze functionaliteit (al dan niet in aangepaste vorm) gewenst is. Je moet te allen tijde voorkomen dat je code ontwikkelt puur en alleen omdat het mooi en uitdagend is, of omdat het technisch kan. De wens van de klant geeft in alle gevallen de doorslag. Diegene moet tenslotte bepalen wat bepaalde functionaliteit waard is.

Dit levert de richtlijn op voor de klant

Ook aan deze SIG-richtlijn ligt divers onderzoek ten grondslag om te laten zien wat het uiteindelijke resultaat ervan is. Door projecten te definiëren in manjaren werk, hebben ze aantoonbaar gemaakt wanneer een project groter wordt dan voorgeschreven, de kwaliteit ervan omlaaggaat en het project vertraging oploopt of zelfs faalt. In dat laatste geval wordt de applicatie waaraan is gewerkt nooit in productie genomen. Hou je de codebasis klein, dan neemt de kans van slagen dus toe.

Zo helpt de richtlijn mij als ontwikkelaar

Sylvia Straub – IT Prof OVSoftware

"In sommige gevallen werk je alleen aan bepaalde functionaliteiten en heb je in het geval van meerdere codebases maar met een kleiner deel van de code te maken. Hierdoor is de code makkelijker te begrijpen waardoor je als ontwikkelaar bij de start op het project sneller nieuwe code op kunt leveren. Een kleine codebase betekent voor mij als developer dat specifieke stukken code makkelijker te vinden, sneller te begrijpen en eenvoudiger aan te passen én te testen zijn. Daarnaast loop je met minder code ook minder risico dat er na een aanpassing onverwacht (andere) dingen fout gaan die je van tevoren niet had kunnen voorzien."

5. Houd de interfaces van units klein

'Keep Unit Interfaces Small'

Keep unit interfaces Small – wat houdt dat in?

Als we het hebben over een unit interface is het handig om eerst uit te leggen wat een unit nu eigenlijk is. Een unit is een functie in je softwarecode die een bepaald aantal regels code uitvoert, bijvoorbeeld het versturen van een e-mail. Een e-mail is niet compleet zonder een aantal aanvullende stukjes informatie zoals de aanhef en een onderwerpregel. Dit zijn de zogenaamde parameters van een functie. Een interface is het totaal van de parameters die je mee kunt geven aan een unit. Het gevaar bestaat dat je bij het programmeren van een functie al snel te veel parameters meeneemt; een aantal dat bovendien bij latere aanpassingen steeds groter wordt. Je wilt dan niet alleen gebruik kunnen maken van een aanhef en onderwerpregel, maar ook mensen in cc op kunnen nemen of bijlagen mee kunnen sturen. De richtlijn: 'Keep unit interfaces small' stelt dat je het aantal parameters per functie op maximaal vier houdt.

```
1  //Unit interface met meer dan 4 parameters
2  public double BerekenAfstand(
3      string postcodeVan, string huisnummerVan, string isoCodeLandVan,
4      string postcodeNaar, string huisnummerNaar, string isoCodeLandNaar)
5  {
6      ....
7  }
8
9  //Methode om dit op te lossen is Extract object.
10 //Velden worden in object gezet
11 public Adres
12 {
13     public string Postcode { get; set; }
14     public int Huisnummer { get; set; }
15     public string IsoCodeLand { get; set; }
16 }
17
18 //Unit interface heeft nu minder dan 2 parameters en object kan worden hergebruikt
19 public double BerekenAfstand(Adres van, Adres naar)
20 {
21     ....
22 }
```

De richtlijn toegepast in de praktijk

Voor een klant zijn we op dit moment bezig met een traject waar mensen op basis van hun werkrooster aan moeten geven hoe hun woon-werkverkeer eruit ziet. Door eenmalig aan te geven waar hun huis, het kantoor of andere locaties zich bevinden, worden de gereisde afstanden berekend. Het registreren van deze vertrek- en doeladressen kan op verschillende manieren: met behulp van een postcode, x- en y-coördinaten en ISO-landcodes. Wil je deze gegevens vastleggen in de code van een unit? Dan zit je al snel op meer dan vier parameters. Parameters die bij elkaar passen, kun je samenvoegen in een adresobject en deze vervolgens als één parameter opnemen in je unit. Omdat er in dit voorbeeld telkens sprake is van een vertrek- en aankomstadres, kun je bovendien het adresobject twee keer gebruiken.

Dit levert de richtlijn op voor de klant

Net als de andere tien SIG-richtlijnen voor het realiseren van softwarecode van hoge kwaliteit draagt deze richtlijn bij aan het eenvoudiger kunnen begrijpen, (her)gebruiken en onderhouden van software. Met op termijn lagere kosten als gevolg. Omdat een unit slechts zo'n klein onderdeel is van je softwareoplossing, zou je kunnen denken dat het belang ervan minimaal is. Maar al deze kleine units samen leggen wél de basis voor je hele systeem. Ten slotte kunnen allemaal kleine foutjes bij elkaar kunnen ook leiden tot grotere problemen. De bijdrage die je kunt leveren aan het welzijn en de tevredenheid van je medewerkers is minstens net zo belangrijk. Blije developers zijn productieve en innovatieve developers.

Zo helpt de richtlijn mij als ontwikkelaar

Martin Aarnoudse – IT Prof OVSoftware

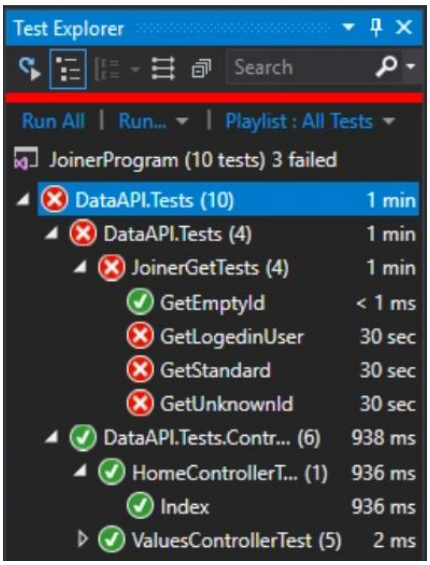
"Door het aantal parameters beperkt te houden, is de code veel eenvoudiger te lezen en te begrijpen. Dat maakt het (her)gebruik van functies ook makkelijker en de kans op het maken van fouten wordt minder. Eigenlijk is het feit dat een functie veel parameters heeft op zich niet het probleem, maar het is eerder een indicatie van een onderliggend issue. Namelijk dat er sprake is van een slecht datamodel of dat code ad hoc wordt aangepast. Als developer is het werken in goed geschreven functies met een beperkt aantal parameters veel fijner, het is een bepalende factor voor het plezier dat ik heb in mijn werk. "

6. Automatiseer testen

'Automate tests'

Automate tests – wat houdt dat in?

Als je het hebt over geautomatiseerd testen dan gaat het over het testen van de code units waarbij je kijkt of zij doen wat ze moeten doen. Een unit is een functie in je softwarecode die een bepaald aantal regels code uitvoert, bijvoorbeeld het versturen van een e-mail. Bij het testen van de unit kijk je naar heel veel verschillende aspecten. Denk aan de diverse ontwikkelpaden (beslisbomen voor uiteenlopende scenario's) of aan de werking van de unit met of zonder inzet van parameters2. Bij het testen kijk je niet alleen of de werking van de functie goed uitpakt, maar probeer je vooral dingen uit die fout kunnen gaan. Stel je hebt bepaald dat je een foutmelding wilt ontvangen op het moment dat er geen waardes in de parameter staan. Uit de test blijkt echter dat de unit gewoon doorgaat zonder deze melding af te geven. Dan weet je dat er aanpassingen nodig zijn. Geautomatiseerde testen worden vaak als regressietest ingezet. Bij elke aanpassing of uitbreiding van de code wordt er gecontroleerd of er niet onbedoeld iets is stukgegaan in de code. Blijkt uit de test dat dit wel het geval is? Dan kun je dit direct controleren en waar nodig corrigeren.



De richtlijn toegepast in de praktijk

Zelfs bij het bouwen van een unit waarbij je nog niet handmatig kan testen, omdat je bijvoorbeeld op dat moment nog niet via de applicatie bij de betreffende unit kan komen, kan je alle mogelijke unittesten al schrijven. Deze kan je op de unit testen waarbij het resultaat honderd procent 'goed' is. De geschreven code kon dan met een gerust hart opgeleverd worden, want wat er ook gevraagd zou worden, de unit zou doen wat hij moest doen. Nu vraag je je wellicht af hoe je er zeker van kunt zijn dat je alles hebt getest wat er maar te testen valt. Dat doe je door gebruik te maken van een Test Framework dat in elke ontwikkelomgeving beschikbaar is. Zodra je je testen hebt uitgevoerd, kleurt het onderdeel van de unit dat hiermee geraakt is en positief

bevonden op groen en je ziet of je nog delen hebt gemist. De kwaliteit van de ontwikkelaar is natuurlijk bepalend voor de opzet van de tests die worden gedraaid en uiteindelijk ook voor de kwaliteit van de opgeleverde code.

Dit levert de richtlijn op voor de klant

Projecten waarbij gebruik wordt gemaakt van automatisch testen, vallen qua kosten hoger uit omdat het opzetten en uitvoeren van testen initieel meer tijd kost. Helaas wordt er vaak om die reden besloten om het onderdeel te laten vallen. De negatieve impact op de onderhoudbaarheid van de software op langere termijn, wanneer aanpassing van code nodig is, wordt daarbij vergeten. Als je code niet getest is, weet je nooit of een aangepaste variant goed of fout zal gaan. Wanneer je dan handmatig moet gaan testen, kom je er al snel achter dat dat bijna niet te doen is. En juist hierbij ontstaan veel fouten. Dus ons advies is altijd: ga voor automatisch testen want op lange termijn leidt dit altijd tot grotere efficiency, betere onderhoudbaarheid en eenvoudiger aan te passen softwarecode.

Zo helpt de richtlijn mij als ontwikkelaar

Martin Aarnoudse – IT Professional OVSoftware

“Het automatiseren van testen helpt je als developer om te checken of aangepaste software nog steeds doet wat het moet doen. Het helpt je ook wanneer er nog geen andere mogelijkheid is om de unit te testen, zoals in eerder genoemd voorbeeld. Het geeft je houvast, de zekerheid dat het goed is. Dat geldt voor je eigen code en voor de code van anderen die je mogelijk minder goed kent. Je krijgt dan de bevestiging (of niet) dat je niets ongewild hebt aangepast of stukgemaakt. Het toepassen van deze richtlijn wil niet zeggen dat er nooit fouten in je code zullen zitten. Wel stelt het je in staat om gemaakte fouten sneller te ontdekken en te corrigeren. Vergeleken met manueel testen is automatisch testen veel eenvoudiger, sneller en preciezer.”

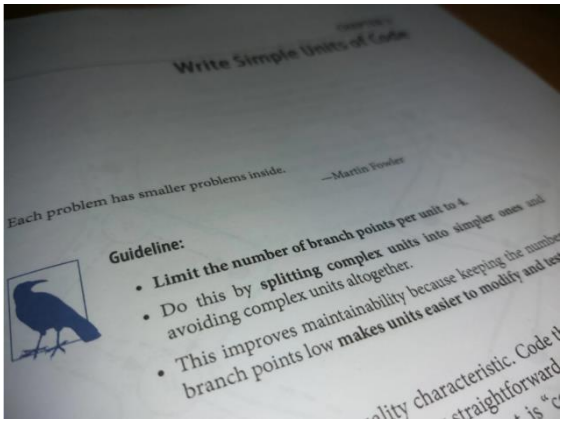
7. Schrijf simpele code units

'Write simple units of code'

'Write simple units of code' – wat houdt dat in?

Hoe beoordeel je of een codebase simpel is of complex? Een stuk code kan voor een buitenstaander of iemand die net als developer begint heel ingewikkeld lijken. Voor de ontwikkelaar van deze code of een heel ervaren persoon is het waarschijnlijk prima te volgen. Of iets simpel of complex is, is dus subjectief. Toch is er een punt waarop een stuk code zo complex is, dat het doorvoeren en testen van aanpassingen risicovol en tijdrovend wordt. Daarom wil je complexiteit meetbaar maken. Een van de manieren waarop je dit kunt doen, is met de eerder beschreven SIG-richtlijn: 'Schrijf kleine functies.' Deze richtlijn schrijft voor dat elke methode (een aantal regels code dat ervoor zorgt dat een actie wordt uitgevoerd) maximaal 15 regels code bevat. Een andere manier om objectief te bepalen hoe ingewikkeld de code al dan niet is, is door te kijken hoeveel ontwikkelpaden (beslisbomen voor uiteenlopende scenario's) een methode bevat. Een ontwikkelpad begint op een zogenaamd branch point waarbij je door gebruik te maken van bijvoorbeeld een 'IF', 'OR' of 'AND' statement verschillende wegen kunt bewandelen. Bijvoorbeeld: IF leeftijd is >18 dan gebeurt X, anders Y. Of een 'OR' statement: IF leeftijd >18 OR leeftijd ==55 (gelijk aan 55)… etc. De uitkomst van het statement bepaalt welke kant je opgaat op een branch point. Door het aantal branch points te beperken tot vier per methode weet je zeker dat je methodes overzichtelijk blijven.

De richtlijn toegepast in de praktijk



De richtlijn is dus om simpele code units te schrijven door je te houden aan een maximaal aantal van vier branch points per methode. Als je nieuwe code aan het schrijven bent, hou je continu in de gaten dat je het aantal niet overschrijdt. Heb je toch meer dan vier branche points nodig? Dan maak je een nieuwe methode aan waar je één of meerdere extra branch points in onderbrengt. De naam van de methode geeft aan welke actie wordt uitgevoerd, hierdoor is de opbouw van de code voor iedereen goed te volgen. Daarom is het belangrijk dat de code die je onderbrengt in een methode altijd past binnen de uit te voeren actie van die methode. Werk je aan verbeteringen van bestaande code en zie je dat er sprake is van te veel branch points per methode? Dan ga je op dezelfde manier te werk door het teveel aan branch points onder te brengen in een nieuwe methode.

Dit levert de richtlijn op voor de klant

De richtlijnen van de SIG dragen bij aan code van hoge kwaliteit waardoor je deze beter kunt onderhouden. Eenvoudige, goede code betekent dat de developer minder tijd

kwijt is aan het zoeken, doorgronden, aanpassen en testen van de codebase. Het zorgen voor een kwalitatief goede codebase betekent dat je gedurende de hele levensloop van de ontwikkelde software profiteert van de voordelen. Deze voordelen zijn onder andere: minder bugs, minder downtime, snel kunnen doorvoeren, testen van correcties en aanvullingen. En hoe minder tijd developers aan dit soort zaken besteden, hoe minder het de opdrachtgever kost.

Zo helpt de richtlijn mij als ontwikkelaar

Murat Yilmaz – IT Professional OVSoftware

"In mijn werk maak ik dagelijks gebruik van de tien SIG-richtlijnen. Net als verschillende andere richtlijnen zorgt de richtlijn om simpele code units te schrijven ervoor dat de code makkelijker te begrijpen, aan te passen en te testen is. Mede doordat je niet meer dan vier branch points in één methode kwijt kunt, blijven de methodes klein. Bovendien kun je aan de naam van de methode precies zien wat deze in de praktijk moet doen. Wil je wat aanpassen aan je code dan is het veel makkelijker om een groot aantal kleine, duidelijk beschreven methodes te doorzoeken dan een hele grote hoop van code. Het is sneller te doorzien waar aanpassingen nodig zijn en ook het testen van aangepaste code kost zo minder tijd en moeite."

8. Scheid concerns binnen modules

'Separate Concerns in Modules'

'Separate Concerns in Modules'– wat houdt dat in?

Voor je mensen zonder kennis van technische aspecten van softwareontwikkeling de richtlijnen van de SIG uit kunt leggen, is het handig om eerst in te gaan op de betekenis van de gebruikte termen. In deze richtlijn komen: 'concerns, modules en klasse' aan bod, maar wat houden ze in? Een concern van een module bepaalt welke belangen deze module heeft en welke acties deze verzorgt. Een module bestaat op zijn beurt uit klassen die qua opbouw bij elkaar horen, waarbij een klasse staat voor een bestand met code. Vergelijken we het met meer toegankelijke techniek, zoals de techniek van een auto. Je kunt zeggen dat er binnen een module code staat voor het ontvangen van energie. In een andere module staat de code voor de accu zodat deze energie kan versturen. Op deze manier zijn de concerns goed gescheiden. Als er daarentegen in de verlichtingsmodule ook klassen zijn die coderen hoé de energie uit de accu komt, dan is dat niet goed gescheiden en geeft het eerder problemen. Als je ervoor zorgt dat je tijdens het software ontwikkelen binnen een module niet meer dan één concern hebt, kun je naderhand wijzigingen in de code doorvoeren of nieuwe functionaliteit toevoegen zonder dat je daarbij het risico loopt dat er ergens anders ongewild (en soms ook ongemerkt) dingen fout gaan.

De richtlijn toegepast in de praktijk

Stel je voor dat je een applicatie hebt gemaakt waar de klant na verloop van tijd nieuwe features of elementen aan toe wil voegen of aan wil passen. Hier zitten natuurlijk kosten aan verbonden. De volgende kenmerken bepalen samen deze kosten:

1) De hoeveelheid code die aangepast moet worden.
2) Hoe eenvoudig (of lastig) is het wijzigen van deze code?
3) Hoe groot is het risico dat jouw wijziging ongewild iets stukmaakt dat door andere developers of de applicatie zelf wordt gebruikt?
4) Hoeveel bestaande code je kunt hergebruiken.

Het goed gescheiden houden van concerns binnen een module is vooral bepalend voor het derde punt; het risico dat er door een wijziging ergens anders in de code iets fout gaat.

Neem een voorbeeld uit de praktijk; wij hebben een opdracht gehad waarbij we een bestaand systeem verder moesten uitbreiden en opstellen voor document wijzigingen en eventuele uitbreiding van die documenten. Doordat het systeem klassen bevatte die niet goed aan de richtlijn voldeden, was het om te beginnen al heel lastig om te achterhalen wat het systeem precies deed en kon. De klassen waren enorm groot, hadden meerdere taken (ofwel te veel concerns) en alles liep in elkaar over. Dit maakte bijna alles onleesbaar. Ook het inwerken van nieuwe collega's was lastig en het duurde lang voordat we aanpassingen aan het systeem door konden voeren. Aanpassingen in één deel van het systeem hadden bijna altijd ongewenste effecten in andere delen van

de applicatie. Als gevolg daarvan werd zo'n project groot en complex, daarnaast kost het ook enorm veel.

Dit levert de richtlijn op voor de klant

Het achteraf aanpassen van software waarbij de concerns binnen modules sterk met elkaar verweven zijn, is nogal een avontuur en kostbaar. Het doorvoeren van wijzigingen is duurder omdat je meer tijd kwijt bent aan het zekerstellen dat jouw wijzigingen geen negatief effect hebben op andere delen van de applicatie. Als je aan een applicatie werkt die al in de lucht is, moet je extra voorzichtig zijn bij aanpassingen. De fouten die je maakt zijn namelijk direct zichtbaar voor de buitenwereld. Tijdsdruk, het beschikbare budget en zeker ook de senioriteit van de betrokken developers zijn bepalend voor de kwaliteit van je softwarecode. Maar het leggen van een goede basis levert je uiteindelijk altijd meer op dan de investering die daar in eerste instantie voor nodig is.

Zo helpt de richtlijn mij als ontwikkelaar

Gökhan Güler – IT Professional OVSoftware

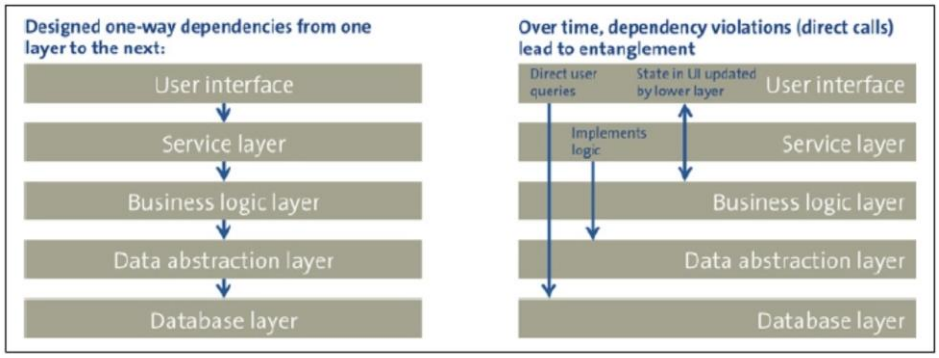
"Als developers zich aan de richtlijn houden om concerns binnen modules gescheiden te houden, dan hoef je bij latere wijzigingen minder code aan te passen. Namelijk alleen díe code waar het op dat moment over gaat, zonder dat dit ergens anders een ongewenst effect heeft. Dat geldt ook wanneer je interfacing en dergelijke toepast. Wanneer je concern beschrijft dat er een koppeling gemaakt kan worden (met welke willekeurige database dan ook) dan kun je de database op elk moment door elke gewenste database vervangen. Je spreekt over een minder felxibele opzet wanneer je in je concern nauwkeurig specificeert met welke databse een integratie gereliseerd moet worden. Het gescheiden houden van concerns betekent ook minder stress voor mij als developer. Je hoeft niet de hele tijd op eieren te lopen in afwachting van wat er waar fout gaat. Je kunt meer code hergebruiken als je met goed gescheiden concerns werkt."

9. Verweef softwarecomponenten op een losse manier met elkaar

'Couple Architecture components Loosely'

'Couple Architecture Components Loosely' – Wat houdt dat in?

Software bestaat uit een aaneenschakeling van componenten, blokken code die elk een eigen doel en functie hebben en daarmee een bijdrage leveren aan het totale systeem. In een goed ontworpen systeem worden deze blokken weergegeven als op elkaar gestapelde lagen. Bij voorkeur gaan ze maar langs één route; van boven naar beneden. Vergelijk het met lego bouwstenen die je normaal gesproken van beneden naar boven op elkaar stapelt. Een gebruiker bekijkt de stapel legostenen altijd van bovenaf; naar de functionaliteit die het systeem te bieden heeft.



Figuur uit 'Building maintainable software' – *Designed versus implemented architecture*

Vanuit daar werk je naar beneden, waarbij je telkens een niveau dieper komt. Het bovenste blokje presenteert een scherm waarin de gebruiker zijn naam en adres invoert. In het blokje daaronder worden deze gegevend verder uitgesplitst in losse onderdelen: voornaam/achternaam/tussenvoegsel en de verschillende elementen van het adres. In het derde blokje wordt bepaald hoe deze informatie wordt weggeschreven en in welke database dat gebeurt etc.
In deze SIG-richtlijn gaat het vooral om hoe componenten onderling samenhangen en met elkaar communiceren en dat de lijnen waarlangs de communicatie verloopt niet kriskras alle kanten opgaan. Hoe deze richtlijn daar invulling aangeeft is door:

- De communicatie tussen het component en 'de buitenwereld' zo veel mogelijk te beperken.
- De communicatie door middel van interface te definiëren (het opstellen van een contract).
- Te voorkomen dat functies van onderliggende componenten één-op- één worden doorgesluisd via een bovenliggend component.

Met dit beeld in het achterhoofd begint men aan het ontwikkelen van software-systemen, alleen wijst de praktijk uit dat dit gedachtegoed na verloop van tijd steeds minder consequent wordt toegepast.

De richtlijn toegepast in de praktijk

Het primaire uitgangspunt van deze richtlijn is dus om componenten zoveel mogelijk geïsoleerd te laten functioneren. De buitenwereld hoeft niet te weten welke specifieke functies er allemaal in een component beschikbaar zijn. Het is alleen relevant om de functies beschikbaar te stellen waar de buitenwereld (lees de gebruiker/systeem) iets mee moet. En dat beschikbaar stellen van functies doen we door het opstellen van een zogenaamd contract. Hierin wordt precies beschreven hoe de buitenwereld met het component kan communiceren. Zo'n contract noemen we ook wel een interface. Het opstellen van een of meerdere interface(s) is een exercitie. Hierbij moet goed worden nagedacht over wat de component zouden moeten toestaan aan input en aan vragen van buitenaf. Hoe de component dat vervolgens intern organiseert en afhandelt is voor de buitenwereld niet van belang, zolang het maar gebeurt.

Dit levert de richtlijn op voor de klant

Net als de andere richtlijnen van de SIG, draagt ook deze richtlijn bij aan een betere kwaliteit van de software voor onze klanten. Code wordt beter gestructureerd. Daarnaast wordt de impact van een aanpassing verkleind omdat je slechts een onderdeel van het systeem wijzigt. Aanpassingen kunnen hierdoor in een kortere tijd worden gerealiseerd. Hierdoor wordt de foutgevoeligheid gereduceerd en de stabiliteit van de software verbeterd.

Zo helpt de richtlijn mij als ontwikkelaar

Emil Cristen – IT Professional OVSoftware

"De toepassing van deze richtlijn zorgt ervoor dat ik als ontwikkelaar de controle heb over wat er met mijn componenten gebeurt. Elke component staat op zichzelf en heeft zijn eigen doel en functie. Dat betekent dat ik snel door de code kan navigeren als ik opzoek ben naar specifieke functionaliteit. De werking van een component is geïsoleerd. Hierdoor kan de functionaliteit eenvoudig worden toegevoegd of aangepast zonder dat de buitenwereld er iets van merkt.
Ook maakt deze richtlijn de mogelijkheid om een component geheel te vervangen eenvoudiger. Zolang we de interface maar intact laten. Dit zien we vaak bij bijvoorbeeld communicatie met databases. De ene keer gebruiken we SQL Server, de andere keer MySQL of Oracle. Voor de buitenwereld maakt het niet uit waar iets wordt vastgelegd. Tenslotte biedt deze richtlijn mij de mogelijkheid om de component uitgebreid te voorzien van testen. Zowel voor wat betreft de interne werking van de component als de interface met de buitenwereld. Hierdoor weet je zeker dat de communicatie tussen de component en de buitenwereld zonder problemen overeind blijft."

10 Zorg ervoor dat code leesbaar is

'Write clean code'

Write clean code' – wat houdt dat in?

Het schrijven van 'clean code' gaat voornamelijk over leesbare code, maar dat is niet de essentie waar het hier over gaat. Waar men op doelt, is dat je ongebruikte code zoveel mogelijk verwijdert. Ballast weghalen als het ware. Van het overgrote deel van de code die je schrijft, moet elke andere developer zonder verdere uitleg kunnen zien wat het doet. Maar soms zijn er complexe stukken code waar uitleg in de vorm van 'comments' bij nodig is. Op zich is dat geen probleem, maar commentaren die op enig moment geen waarde meer toevoegen moet je weghalen omdat ze de code anders onnodig vervuilen. Neem bijvoorbeeld de veelvuldig geplaatste opmerking "hier moet ik nog wat mee, want dit werkt niet". Op het moment dat deze aantekening wordt gemaakt is het een relevante opmerking waar nog iets mee moet gebeuren, maar zodra het probleem is opgelost moet je niet vergeten om het weg te halen.

De richtlijn toegepast in de praktijk

Padvinders hebben een regel die erop neerkomt dat ze hun kampeerplek schoner achterlaten dan dat ze deze hebben gevonden. Datzelfde geldt voor software developers die de regel van leesbare code naleven: telkens wanneer je ergens een stuk code aanpast, heb je de mogelijkheid om tegelijkertijd kleine verbeteringen op andere plekken in die code door te voeren. Stel je voert een aanpassing door in de code of een bestand en ziet dat een bepaalde functie op een andere plek ook slimmer, beter of efficiënter kan. Je hebt deze code nu toch onderhanden, dan kun je deze verbeteringen zonder al te veel moeite gelijk meepakken. Laat je het zitten dan krijg je vervuiling van je systeem en dat is nou precies wat je níet wilt.

Manieren waarop je de kwaliteit van de code optimaliseert zijn bijvoorbeeld:

- Waar mogelijk geen commentaar achterlaten in code, alleen als het echt niet anders kan.
- Voorkomen dat je code in comments laat staan. Oude versies van herschreven code worden vaak tijdelijk vastgelegd in de vorm van comments, maar zodra de herschreven code in gebruik wordt genomen, kun je de comments verwijderen. Alle versies van de code worden namelijk ook vastgelegd in het versiebeheersysteem, dus kunnen te allen tijde worden bekeken.
- Ervoor zorgen dat je zogenaamde 'dode code' die niet meer wordt gebruikt in het systeem verwijdert.
- Altijd duidelijke namen voor je componenten en formules en dergelijke gebruiken zodat voor iedereen direct duidelijk is wat ze doen.

Dit levert de richtlijn op voor de klant

Leesbare code zorgt ervoor dat deze eenvoudig overdraagbaar is aan collega's. Als opdrachtgever ben je dan niet meer enkel aangewezen op die één of twee specifieke personen die ooit aan de wieg hebben gestaan van een bepaald softwaresysteem. In principe is de code namelijk zo duidelijk dat iedereen er in korte tijd mee aan de slag kan. Aangezien er in de meeste organisaties en bedrijven regelmatig sprake is van een veranderende teamsamenstelling, levert deze richtlijn dus vooral een bijdrage aan de continuïteit van processen en daarmee indirect ook aan de continuïteit van je bedrijf.

Zo helpt de richtlijn mij als ontwikkelaar

Emil Cristen – IT Professional OVSoftware

"Het is voor mij een hele bewuste keuze om naast mijn functie als CTO ook als ontwikkelaar actief te blijven. Ik wil mezelf blijven ontwikkelen en bijhouden wat er om mij heen gebeurt in de wereld. Dat betekent niet dat ik mij met elk detail ga bemoeien, maar ik wil wel weten wat er speelt en wat het op hoofdlijnen betekent. Datzelfde geldt bijvoorbeeld ook voor de richtlijnen van de SIG. In mijn rol als developer ben ik continu alert op mogelijkheden om de door mij of door anderen geschreven code verder te optimaliseren. Bijvoorbeeld door de eerder genoemde 'dode code' weg te halen. Mijn ervaring leert dat juist deze loze code ervoor zorgt dat ik op het verkeerde been word gezet. Dode code dient uiteindelijk nergens meer voor, dus elke minuut die je daaraan besteedt, is er een te veel. Door deze ballast structureel weg te halen, voorkom ik tijdverspilling op een later moment."

Kwaliteit & Ontzorging

Software oplossingen
Beheer & Onderhoud
Detachering