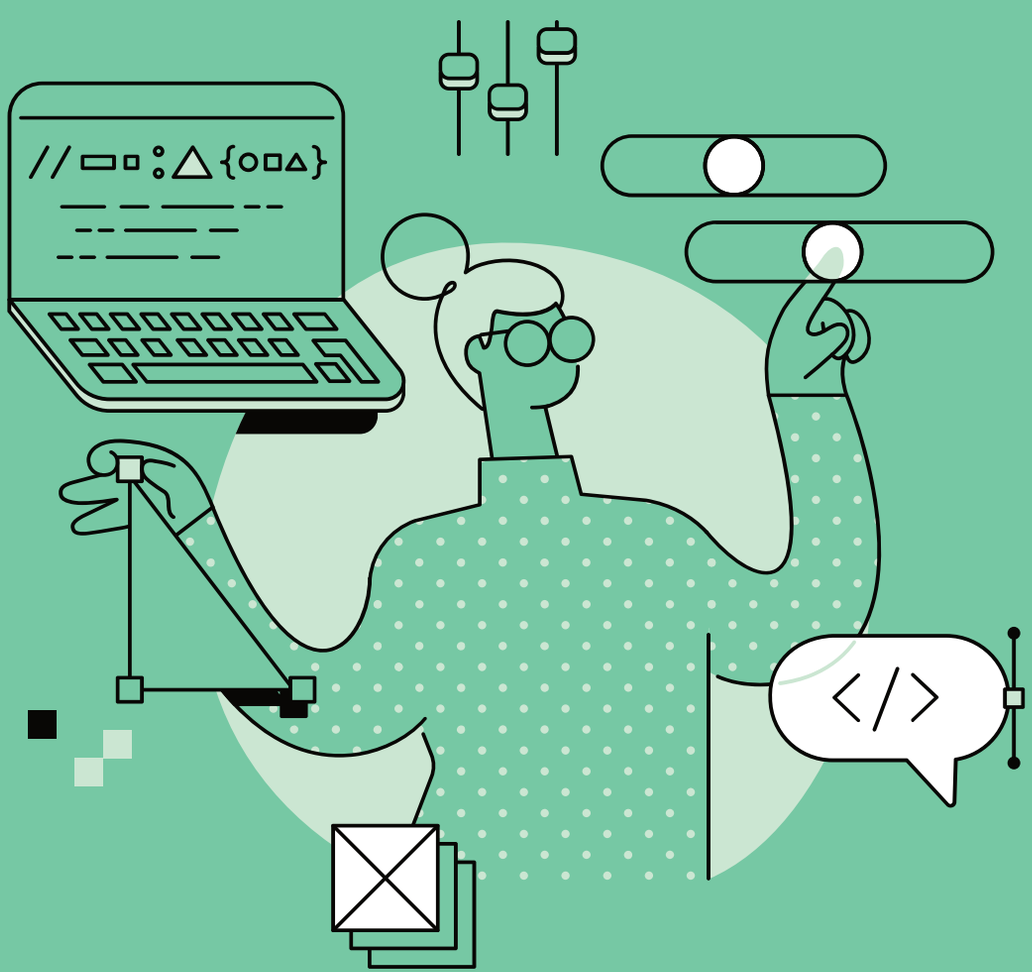
85 kansen en signalen om je softwareontwikkeling nog beter aan te pakken



aviva solutions

Met bijna 25 jaar ervaring achter de kiezen kom ik als Principal Consultant bij veel verschillende organisaties om ze te helpen met het professionaliseren van haar softwareteams. Daarbij kijk ik onder andere naar de ontwikkelmethodieken, ontwerptechnieken, tools, teamdynamiek en planningsprocessen. In deze serie korte posts neem ik jullie mee langs een aantal veelvoorkomende pijnpunten.

Uiteraard zijn er altijd ruim voldoende aandachtspunten, maar deze keer beginnen we met de moeite die veel organisaties hebben om de complexiteit onder controle te houden. Hier een greep uit wat ik zoal tegenkom.

Dennis Doomen
Principal Consultant Aviva Solutions

8 signalen dat je de technische complexiteit niet onder controle hebt

1. Er is een architectuur gekozen die helemaal niet past bij de functionele en technische eisen. Soms is zo'n keuze gedreven door een hype op het internet. Soms omdat de leidende architect iets opgepikt heef op een populaire conferentie. Maar ik heb ook gezien dat een architectuurteam een centrale keuze maakt waar alle systemen zich aan moeten conformeren. controle te houden. Hier een greep uit wat ik zoal tegenkom.

2. Technologie die wordt gekozen "omdat het gaaf is", aantrekkelijk is voor het vinden van personeel, of simpelweg omdat het management geloofd in de gouden bergen die ontwikkelaars, de grote software vendors of architecten beloven.

3. Het aanbieden van HTTP APIs of technische koppelpunten aan andere teams of partijen zonder na te denken over de verantwoordelijkheden die daarbij komen kijken rondom versionering, backwards compatibiliteit, documentatie en de vrijheid om in de toekomst andere keuzes te maken.

4. Teams die complexiteit bouwen zoals abstracties en lagen die nog helemaal niet nodig zijn. Vaak gebeurt dat vanuit de gedachte van genericiteit, het systeem "beter testbaar" te maken of om "meer toekomstvast" te zijn.

5. Technische leads of architecten die te pas en te onpas gooien met architectuurprincipes, design patterns en heuristieken om zo betere software te bouwen, maar daar dan heel dogmatisch in zijn omdat ze nog niet ervaren genoeg zijn om in te schatten wanneer zoiets juist wel of juist niet handig is.

6. Een onbalans tussen tactische architectuur en strategische architectuur, waarbij het ene eind van de schaal resulteert in te kortetermijndenken, terwijl het andere eind eindigt in extreem ambitieuze (en onrealistische) luchtkastelen.

7. Teams die veel moeite hebben om de afhankelijkheden tussen de vele componenten, services en producten in kaart te krijgen met lastig te plannen teamoverstijgende activiteiten als gevolg.

8. En als laatste, maar niet minder belangrijk; ontwikkelaars die het maar niet voor elkaar krijgen om samen een stuk functionaliteit te bouwen en uiteindelijk in isolatie verder werken, of het omgekeerde, waarbij de ontwikkelaars elkaar steeds in de weg lopen.

12 kansen om de productiviteit van je softwareteams te verhogen

1. Ontwikkelteams die refactoren geen onderdeel vinden van hun softwareontwikkel-praktijken, of waarvan hun manager vindt dat dit tijdsverspilling of te veel risico's met zich meebrengt. Vaak is dat het gevolg van een gebrek aan kennis, refactoring tools of b.v. de keuze voor dynamische talen als JavaScript. Als gevolg hiervan zie je veel "dode" code, het gebruik van de verkeerde benaming, onnodig grote blokken code, of andere zogenaamde "code smells" die de onderhoudbaarheid in gevaar brengen.

2. Technical debt die niet onder controle is en waar niet continue aan gewerkt wordt. Soms is het de manager die dit probleem niet goed snapt, soms is het de druk om zo snel mogelijk nieuwe functionaliteit op te leveren en soms is het een gebrek aan de kant van de ontwikkelaars omdat ze het grotere plaatje niet zien. Maar meestal is het een combinatie.

3. Ontwikkelaars wiens enige doel is om de boel werkend te krijgen en te weinig nadenken over de structuur van hun code, b.v. door het toepassen van de S.O.L.I.D. principes of het refactoren naar Design Patterns. Dit heeft dan vaak het gevolgen dat het introduceren van nieuwe functionaliteit lastig is. Aan de andere kant van de medaille zie je ook tech leads die zo gefocused zijn op dit soort principes en patronen dat er onnodig rework gedaan wordt of dat er zo veel abstracties (lagen) geïntroduceerd worden dat niemand het meer snapt.

4. Een organisatie die een bepaalde architectuur nastreeft, met daarin hele specifieke afgebakende gebieden en koppelvlakken, maar haar teams niet op zodanige (fysieke) manier hebben georganiseerd. Of, vanuit een ander perspectief gezien, organisaties die architectuur, tools, mensen, processen en infrastructuur als onafhankelijke ingrediënten van software ontwikkeling zien.

5. Pull Requests (als die al gebruikt worden) met honderden files met wijzigingen, veelal met niet gerelateerde wijzigingen en commits met teksten waaruit het gebrek aan structuur in de aanpak doorsijpelt. Soms zitten daar ook refactorings tussen waardoor het vinden van functionele wijzigen als het zoeken van een spelt in een hooiberg voelt.

6. Een logisch gevolg hiervan is dat het reviewen hiervan ofwel helemaal niet gebeurd, ofwel dagen of zelfs weken duurt. Maar ik zie ook net zo goed teams die helemaal geen review doen en hun code direct mergen met de centrale code base, zonder een enkel vangnet.

7. Teams waar individuele ontwikkelaars allemaal op zichzelf werken, waardoor user stories onvoldoende voortgang maken en het al helemaal te veel gevraagd is om stukken functionaliteit in een sprint op te kunnen leveren. Vaak wordt er als reden gegeven dat pair programming dan geen zin heeft, of dat het niet mogelijk is om op het werk te "swarmen"

8. Repositories waarvan het niet meteen duidelijk is wat de releasestrategie is en wat de versienummers precies betekenen betreffende backwards compatibiliteit, bug fixes en nieuwe functionaliteit.

9. Ontwikkelaars zijn mensen, en mensen hebben persoonlijke voorkeuren waar ze zich het prettigs bij voelen. Dus het is niet meer dan logisch dat ontwikkelaars graag de tools gebruiken waar ze het meest efficiënt in zijn. Ik zie bij veel organisaties dat er een sterkte behoefte is aan standaardisatie, terwijl ze geen idee hebben hoeveel geld ze dat uiteindelijk kost aan softwareontwikkelaars die onproductief zijn of uit frustratie het bedrijf verlaten.

10. Diezelfde ontwikkelaars willen zich ook nog wel eens beperken tot alleen maar command-line tools of juist alleen maar tools met een grafische user interface. Ik zie deze uitersten zowel bij bedrijven die helemaal in de Microsoft Azure Devops hoek zitten alsmede bij bedrijven die meer de open-source kant gekozen hebben. Beide oplossingen hebben voordelen en kunnen je juist samen heel effectief maken.

11. Een mooi voorbeeld hiervan is het overmatig gebruik van email, het gebruik van chattools zoals Slack en Teams voor gestructureerde communicatie, en het toepassen van Confluence, Sharepoint en andere wiki-achtige oplossing voor ongestructureerde documentatie. Elke tool heeft voor- en nadelen. Het nastreven van één enkele tool voor een specifiek probleem is vaak heel naïef.

12. Een gebrek aan innovatie, bijvoorbeeld door hier geen tijd voor vrij te maken. Dit is vaak niet alleen een klacht vanuit de ontwikkelteams, maar ook een klacht vanuit het management. Soms zie je dan nog wel eens een aparte afdeling die zich als enige met innovatie bezighouden, terwijl die innovatie juist uit de ontwikkelteams komt. Je ziet dan ook regelmatig dat ontwikkelaars niet de vrijheid hebben om nieuwe tools of technieken te proberen.

8 tekenen dat ontwikkelaars elkaar in de weg zitten door verkeerde architectuurkeuzes

1. Een monolithisch systeem is niet persé slecht en zegt alleen maar dat het als geheel uitgerold wordt. Maar dat betekent nog niet dat het intern een spaghetti zou moeten zijn. Maar helaas is dat wel wat je vaak ziet. Een monoliet die intern opgebouwd is als mooi geïsoleerde modules zie je helaas minder vaak, met alle gevolgen van dien. Het is in zo'n systeem dan ook lastig om met meerdere teams tegelijkertijd te werken zonder dat ze elkaar in de weg zitten. Dit soort situaties wordt regelmatig gezien als (naar mijn mening) verkeerd argument om naar een micro-services systeem te migreren.

2. Organisaties die last hebben van de schaalbaarheid van haar ontwikkelteams wijten dit vaak aan de architectuur zoals de monoliet waar we het net over hadden. Een veel voorkomend gevolg hiervan is dat de organisatie overweegt om het systeem opnieuw te bouwen, veelal met support van de ontwikkelaars, want die doen niets liever dan nieuwe dingen bouwen. Ik ben van mening dat herbouw in meeste gevallen de slechtste keuze is die je kan maken, en er vaak veel betere manieren zijn om vernieuwingen door te voeren.

3. Bugs die ontstaan door codewijzigingen op plekken in het systeem die helemaal niets met de code waar de bug zit te maken lijkt te hebben. Soms zijn die afhankelijkheden nog wel redelijk snel te vinden, maar soms is het ook veel subtieler en wordt dit soort bugs veroorzaakt door (globale) variabelen die b.v. over threads of meerdere modules gebruikt worden. Dan zit een ontwikkelaar al snel uren in de zogenaamde debugger hell.

4. Een systeem met een gelaagde architectuur lijkt op papier een goed idee met een mooie scheiding tussen het functionele domein en de technische aspecten in b.v. een data laag. Maar in de praktijk is dit een nogal achterhaalde architectuurstijl die resulteert in heel veel generieke code. Vaak is het niet duidelijk hoe die bijdraagt aan het functionele probleem, en kan dus ook niet geoptimaliseerd worden. Bovendien zijn de hogere lagen enorm afhankelijk van de lagere lagen, met als gevolg een overvloed aan koppelingen. Voor een ontwikkelaar is het meestal niet meer te overzien waar de impact is van functionele wijzigingen.

5. De meeste ervaren ontwikkelaars zullen SOLID, een set van ontwerpprincipes die helpen om je code beter begrijpelijk en makkelijk aanpasbaar te maken, inmiddels wel kennen. Maar net als met elk ander patroon of principe kun je ook hier weer te ver gaan. Te veel lagen en abstracties, simpelweg door de SOLID principes te dogmatisch te volgen, maken het juist veel lastiger om de relaties tussen de code in je systeem te begrijpen. Het vinden van de juiste balans is en blijft een lastige zaak.

6. Een typische eigenschap van een ontwikkelaar is dat ze iets niet meerdere keren willen doen. Ze zijn al snel geneigd om een stuk code dat op meer plaatsen voorkomt te refactoren naar een stuk herbruikbare code dat centraal beheerd wordt. Een veel voorkomend argument hiervoor is dat een bug in die code maar op één plek hoeft te worden opgelost. Dit principe, ook wel Don't Repeat Yourself, of kortweg DRY, is één van de vele technieken die jonge ontwikkelaars al heel vroeg wordt aangeleerd. Maar dit gedrag leidt ook vaak tot veel te generieke code, en dus (weer) te veel afhankelijkheden tussen modules in het system.

7. Een wijs iemand heeft ooit eens gezegd dat als het voor een nieuwe ontwikkelaar meer dan vijf minuten duurt om de code te downloaden vanaf de bron en de applicatie werkend te krijgen, je als bedrijf iets verkeerds doet. Eén van de oorzaken hiervan is wanneer je organisatie te afhankelijk is van een specifiek product voor het bouwen van software. Een mooi voorbeeld hiervan is de build pipeline van Microsoft's Azure DevOps. Dat de code online in een Git repository staat is prima en vooral heel verstandig, maar voor de rest moeten de ontwikkelaars vooral zo onafhankelijk mogelijk van online diensten kunnen werken.

8. Een andere manier om de onafhankelijkheid te meten is te kijken hoe makkelijk het is voor een team om een codewijziging door te voeren op een module of service waarvan dat team geen eigenaar is, en dit ook nog te kunnen testen op een echte testomgeving. Het zou je verbazen hoe vaak je ziet dat zo'n team volledig geblokkeerd wordt door lastige prioriteitsdiscussies en uiteindelijk volledig vastloopt. Dit is één van de grootste problemen om een organisatie schaalbaar te maken, helemaal als je overweegt om wat met micro-services te gaan doen.

15 signalen om te bepalen of je systeem testbaar genoeg is

1. Het in dienst hebben van professionals die heel goed zijn in het "breken" van applicaties en het verbeteren van de testbaarheid van een systeem, maar die dan "testers" noemen i.p.v. b.v. test engineers.

2. Het gebrek aan onderscheid tussen gestructureerd en ongestructureerd testen waardoor de unieke skills en mindset van een test engineer verspild wordt aan handmatig testen van scenario's die volledig geautomatiseerd hadden kunnen worden.

3. Test engineers die niet voldoende betrokken zijn bij het uitwerken van de functionele wensen waardoor de testbaarheid van de functionaliteit niet expliciet in de breakdown wordt meegenomen.

4. Het handmatig testen van een webapplicatie tegen meerdere browsers of onvoldoende gebruik maken van zogenaamde geautomatiseerde snapshot testen om de visuele verschillen te detecteren als gevolg van onverwachte wijzigingen.

5. Geautomatiseerde browser testen die onstabiel zijn, of na een tijdje stabiel lijken te zijn geweest, toch weer onstabiel worden. Ook al gebruik je de modernste UI automation tools, een goed ontworpen Test Automation Framework is cruciaal om dit onder controle te krijgen. En dan nog is de kans op succes niet zo heel hoog.

6. Performance testen die niet realistisch genoeg zijn. Dit kan meerdere oorzaken hebben. Wachttijden van gebruikers worden niet goed gesimuleerd, de interactie tussen de browser en de backend van de applicatie is niet realistisch genoeg, de database gebruikt in de testen bevat geen productiedata, caching effecten omdat er maar met één gebruiker getest wordt.

7. Onvoldoende begrip van wat "concurrent users" precies betekent, of onbekendheid met termen als de gemiddelde responsetijd, de 90 percentiel en Apdex scores. Ook het verkeerd interpreteren van performancetesten, loadtesten en regressietesten kom ik vaak tegen.

8. Geen gebruiken maken van microbenchmarks om de performance van kritische modules en subsystemen automatisch te testen als aanvulling op echte performance tests.

9. Webapplicaties die op een ontwikkelmachine getest moet worden, omdat het team niet in staat is om testversies van de software volledig automatisch in de cloud uit te rollen.

10. Automatische testen die niet als onderdeel van de pull requests worden uitgevoerd. Soms komt dit omdat de infrastructuur niet op orde is, maar soms ook omdat het uitvoeren van die testen te lang duurt.

11. Ontwikkelaars die unit testen achteraf pas of zelfs helemaal niet schrijven, of die hun aanpassingen "over de muur gooien" zonder minimaal vijf minuten te besteden om hun eigen wijzigingen te testen.

12. Het niet geautomatiseerd testen van de interactie met de database "omdat dat te lastig is met onze gedeelde database server", met bugs als gevolg die pas in productie boven water komen. Of het niet geautomatiseerd testen van front-end code want "daar zit toch geen bedrijfslogica in"

13. Een test coverage die ruim onder de algemene norm van 90 procent ligt of juist het gebruiken van diezelfde 90 procent als een heilig doel in plaats van een kwaliteitsindicator.

14. Het testen op maar één niveau, ofwel alleen maar op class niveau waardoor het refactoren heel erg lastig wordt, ofwel op component/module niveau waardoor het analyseren van falende tests en het voldoende coveren van alle mogelijke paden een enorme hoeveelheid werk veroorzaakt.

15. Testen die herschreven moeten worden als de geteste code gerefactord wordt. Vaak wordt dit veroorzaakt door architectuurkeuzes waardoor bijvoorbeeld de scope van de geautomatiseerde testen te fijnmazig wordt.

7 symptomen van onvoldoende traceerbaarheid

1. Je zult versteld staan van hoe vaak ik een team vraag om de achterliggende gedachte van een architectuur of technische keuze, en dat niemand meer precies weet waarom, zelfs als de oorspronkelijke ontwikkelaar er nog steeds werkt. En als ze al een keuze hebben vastgelegd, dan is het niet duidelijk welke andere opties overwogen waren en waarom er uiteindelijk voor de uiteindelijke optie gekozen is. Een gevolg hiervan is dat technische keuzes toch weer ter discussie gesteld worden, vooral als nieuwe mensen het team komen versterken. Als de onderbouwing dan ver te zoeken is, begint het hele onderzoekstraject weer opnieuw.

2. Het onvoldoende communiceren van technische veranderingen en de consequenties daarvan op andere teams d.m.v. interne blog posts, notificaties op interne communicatietools zoals Slack en Teams, technische sessies en groepsreviews. Bij kleine teams is het overzicht nog wel te houden, maar zodra jouw organisatie groeit zul je merken dat dezelfde problemen meerdere keren worden opgelost, of dat er onvoldoende gebruik wordt gemaakt van de meest recente kennis.

3. Het schrijven van documentatie zonder dat er is nagedacht voor wie die bedoeld is en wat de actualiteit daarvan gaat zijn. Dit vertaalt zich dan in documentatie die ofwel te technisch is, ofwel op een te hoog niveau is, ofwel een rare mix van beide. Hoe dan ook, de doelgroep kan er niet mee uit te voeten. Daarnaast is het cruciaal om vooraf te bedenken of het een stukje tekst is dat eenmalig geschreven wordt en een natuurlijk verloop in tijd kent, of dat het iets is dat continu moet worden bijgewerkt. En dan is het kiezen van de juiste tool ook nog een aspect wat bijna altijd onderschat wordt.

4. Van een hele andere orde is het vrijgeven van software zonder dat je aan het versienummer kunt zien wat dit betekent voor bestaande gebruikers van die software. Is het meteen duidelijk dat het om een bugfix gaat die zo snel mogelijk uitgerold moet worden? Of bevat die software nieuwe functionaliteit waardoor je zonder risico's kunt upgraden naar de laatste stand van zaken? En als er al een goede versienummering is, zie ik ook nog wel eens dat er een zogenaamd build nummer aan geplakt wordt waardoor dezelfde broncode meerdere versies kan veroorzaken. Terwijl hier zulke simpele en breed geaccepteerde conventies beschikbaar voor zijn.

5. Is de geschiedenis van jouw source control systeem duidelijk genoeg om te begrijpen waarom de wijzigingen nodig waren? Ik zie namelijk nogal regelmatig codewijzigingen waarvan de beschrijving (als je geluk hebt) nog net uitlegt wat er gedaan is. Maar ik zie ook net zo vaak beschrijvingen als "oeps", "fix van fix", "code review comments" en dergelijke. Maar vrijwel nooit helpt de historie van je code om te begrijpen waarom de wijzigingen nodig waren. En misschien nog wel erger, codewijzigingen waarin niet gerelateerde wijzigingen gemixt worden met de te verwachten functionele wijzigingen.

6. Een andere soort van traceerbaarheid is datgene wat je in productie aan informatie verzameld in de vorm van logging. Ik zie hier eigenlijk vooral twee uitersten; te veel logging waardoor de bomen door het bos niet meer te zien zijn, of te weinig logging waardoor het diagnosticeren van productieproblemen bemoeilijkt wordt. En dan heb je ook nog zoiets als logging niveaus, waarbij de ene warning je onnodig op het verkeerde been zet, en de andere error eigenlijk helemaal niet zo kritisch is of zelfs van tijdelijke aard. Zonder goede richtlijnen is dit simpelweg niet op te lossen.

7. En met logging ben je er nog niet. Moderne systemen bestaan uit steeds meer services (of zelfs micro-services) waardoor een simpele gebruikersactie kan resulteren in een berichtenstroom die door meerdere van die services loopt. In zo'n situatie is het bijna niet te meer volgen wat er precies gebeurde, in welke volgorde, en door wie het werd veroorzaakt.

9 codeerpatronen waar een luchtje aan zit

1. Code die misschien wel leesbaar is, maar waarvan de intentie onduidelijk is wordt niemand blij. Als er dan een bug in lijkt te zitten of er moet iets veranderen n.a.v. nieuwe functionele wensen, dan is die intentie natuurlijk cruciaal om te begrijpen wat de oorspronkelijke ontwikkelaar precies bedoeld heeft. En dat is niet per se hetzelfde als wat de code nu doet. Dit zie je terug in de naamgeving, de organisatie van de code, maar ook in codedocumentatie. Vooral naamgeving vinden ontwikkelaars lastig. Of het is te technisch of het herhaalt wat de code doet.

2. Het wordt nog lastiger als de bug in een geautomatiseerde test zit. Als die faalt, is het dan de test die fout was of de te testen functionaliteit? Het is echt geen unicum als ik claim dat er genoeg geautomatiseerde testen zijn die het verkeerde testen. En dat maakt het allemaal niet makkelijker. Kortom, ook testcode is code en het moet dus duidelijk zijn wat de intentie van die test precies was.

3. Gerelateerd aan het (gebrek aan) intentie van de code is het niet voldoende duidelijk maken van het algoritme dat een methode probeert uit te voeren. Methodes abstraheren complexiteit en gebruiken een zinvolle naam om die complexiteit te benoemen. Dus als de bovenliggende methode een combinatie van andere methodes gebruikt en allerlei technische logica is het algoritme niet duidelijk genoeg.

4. Codedocumentatie is een gevoelig onderwerp, en daar kom ik ook heel wat uitersten tegen. Ofwel wordt de code überhaupt niet gedocumenteerd, veelal vanuit waanbeelden als "code is de documentatie" of "als de code veranderd moeten we de documentatie bijwerken. Maar heel dogmatisch alles documenteren is ook geen goed idee. Ook hier is het vinden van een balans de juiste weg, waarbij er goed moet worden nagedacht over het niveau van documenteren.

5. Een wat simpeler pijnpuntje is wanneer je gedwongen wordt om continue op en neer te moeten scrollen om de flow in de code te kunnen volgen. Dat is net zoiets als een boek waarvan de paragrafen in willekeurige volgorde op een hele lange pagina zouden staan.

6. Refactoren is ook al zo'n term die voor veel haat en liefde zorgt. Ik zie teams waarbij refactoring onder druk van de commercie helemaal niet gebeurt en teams waarbij de product owner of een manager eerst toestemming moet geven. Maar ik zie ook ontwikkelaars die het verschil tussen refactoren en redesignen niet snappen en helemaal losgaan om de code perfect te krijgen. En dat helpt natuurlijk allemaal niet met het vertrouwen tussen de teams en het management.

7. Op gebied van class design heb ik ook wel wat pareltjes gezien. Denk aan afgeleide classes die alleen te begrijpen zijn zonder dat je op en neer te springen tussen (meerdere lagen aan) base- en afgeleide classes. Maar ook classes met namen als "Manager" of "Helper", de bekende vergaarbakken voor van alles en nog wat. En vooral senior ontwikkelaars zijn erg gek op het introduceren van lagen, abstracties en design patterns. In dat soort code bases zie je heel veel classes die een interface implementeren met dezelfde naam, maar dan met een I ervoor.

8. Maar ook de code zelf ontkomt niet aan vieze geurtjes. Denk aan heel diep geneste structuren, vaak te vinden in hele lange methodes, vol met magische nummers en variabelen waarvan het type niet meteen duidelijk is omdat iemand nogal dogmatisch omgaat met b.v. zoiets als var in C#. En wat denk je van boolean parameters waarvan het voor de aanroepende partij volkomen onduidelijk is wat die true of false nou precies betekende. En ga er maar vanuit dat als er veel puntjes worden gebruikt om via een variabele bij diep geneste property, field of attribuut te komen, er iets niet goed ontworpen is.

9. En toch is dat gek. Er zulke mooie tools beschikbaar zijn om veel van dat soort "geurtjes" te vinden. Gebruik je b.v. iets als SonarQube om je C#, TypeScript en JavaScript code te testen tegen de best practices uit onze industrie? Of gebruik je ze wel, maar heb je alle vervelende regels uitgeschakeld? Maar ook ESLint voor JavaScript/TypeScript, StyleCop voor C# en vele andere gratis tools zie ik nog te weinig gebruikt worden om snelle feedback te krijgen op de kwaliteit, stijlconventies en leesbaarheid.

6 signalen dat je architectuur niet transparant genoeg is

1. Het niet altijd meteen duidelijk zijn waar een bepaald stukje functionaliteit precies in het systeem is geïmplementeerd. Dat is zeker zoals de code op een nogal technische manier is opgebouwd en dus niet georganiseerd is rondom functionele modules. Dit kan komen omdat een ontwikkelaar te weinig zicht heeft op de onderliggende architectuur, of omdat die architectuur niet evident is.

2. Maar ook wanneer je puur naar de code kijkt zou je de onderliggende architectuur eruit moeten kunnen halen. Sterker nog, in een ideale wereld zou de code de architectuur moeten "uitschreeuwen", ook wel screaming architecture genoemd. En dan kun je natuurlijk ook nog vanuit de user interface kijken, ongeacht of het nu een desktop applicatie is of een Single Page (web) Application. Zelfs dan zou je vrij makkelijk aan de visuele structuur moeten kunnen terugleiden waar de bijbehorende code precies zit.

3. En als de ontwikkelaar de juiste plek wel gevonden heeft, merk ik vaak dat ze niet bewust zijn van waar bepaalde logica precies hoort te zitten, en met welke andere delen van het systeem deze gekoppeld mag worden. Zeker bij hele ervaren ontwikkelaars wil de noodzaak nog wel eens ontbreken omdat ze de architectuur in hun hoofd hebben zitten. Ook kan er aversie zijn tegen zoiets omdat het dan duidelijk zou kunnen worden dat een codebase geen fatsoenlijke structuur heeft. Dit helpt zeker niet om de interne architectuurgrenzen te bewaken.

4. Qua documentatie helpt het verder niet als die documentatie achterhaald is. Ik zie bij sommige organisaties dat het schrijven van documentatie onderdeel is van de definition-of-done. Maar dat betekent niet automatisch dat het heel nuttige documentatie is. Ik heb het al eerder gezegd, maar het is heel belangrijk om te snappen voor wie welke documentatie bedoeld is, hoe actueel het hoort te zijn en wat de juiste locatie is om het vindbaar te maken. Dit geldt uiteraard ook voor informatie over de architectuur. Dat is ook de belangrijkste reden waarom ik niet geloof in UML tools als Enterprise Architect om de gehele architectuur vast te leggen. Het gaat altijd achterlopen. Hier zijn veel betere en laagdrempelige oplossingen voor.

5. En als de architectuur, de code en de documentatie al goed op elkaar afgestemd zijn, dan nog veranderd er veel door nieuwe functionele wensen, technische inzichten, refactoring en dergelijke. Ik zie nog te vaak dat teams te weinig doen om dit wereldkundig te maken. In een van mijn vorige posts heb ik een interne technische blog al genoemd, maar ook technische sessies om een en ander toe te lichten zijn blijkbaar een te grote drempel.

6. Zelfs op codeniveau is dit belangrijk. Heeft elke codeproject (b.v. een Github of AZDO repository) een duidelijke read-me die uitlegt waar dat project voor dient, hoe het gereleased wordt, wie het beheerd, waar de artifacten te vinden zijn, wat je nodig hebt om de code te bouwen en hoe je uiteindelijke oplossing kunt uitrollen? Ik zie zowel bij organisaties als open-source projecten dat het hier nogal aan schroomt.

20 vragen om te bepalen of je teams wel voorspelbaar genoeg zijn

1. Hebben jouw teams al gewerkt volgens Scrum, Kanban of een combinatie daarvan maar lukt het nog steeds niet om een stabiele velocity te krijgen? En vind je dit een probleem of juist inherent aan softwareontwikkeling in het algemeen?
2. Is het voor de teams lastig om het werk in behapbare brokken op te delen waar ze met meerdere ontwikkelaars tegelijkertijd aan kunnen werken?
3. Blijft het lastig om functionele user stories te schrijven voor de meer technische aspecten van een project?
4. Passen grotere technische aanpassingen vaak niet in een user story of zelfs niet in een sprint en lopen de teams dan tegen dogmatische definities aan van wat een user story precies is. Of passen ze principes toe als skeleton stories of stories met project value?
5. Weten de teams precies wat er nodig is voordat het ontwikkelwerk kan beginnen?
6. Zijn schattingen nog steeds een lastig punt, met veel afwijkingen, uitlopers en de neiging om ze toch in uren terug te rekenen?
7. Worden de schattingen überhaupt gezien als een doel op zich of is het vooral een manier om een goede breakdown te kunnen doen?
8. Voelt het definiëren van een sprint goal vooral als een kunstmatig ding of biedt het echt een toegevoegde waarde om het team een doel te geven?
9. Zijn stories toch wel met grote regelmaat aan het einde van de sprint net niet af? En zijn het alleen de grotere stories die niet af zijn, of zijn het toch wel vaak meerdere?
10. Zien de burn down charts eruit als blokdiagrammen of lopen ze juist mooi en regelmatig af?
11. Merk je dat teams toch wel regelmatig de focus op de belangrijkste zaken verliezen of het eigenlijk doel van de story en sprint uit het oog verliezen?
12. Is iedereen even enthousiast betrokken bij de stand-ups? Of zijn het altijd dezelfde mensen die aan het woord zijn?
13. Blijven user stories wel erg lang in dezelfde status hangen zonder dat er voortgang lijkt te zijn? En hoe vaak overtreden je Kanban teams de WIP limit?
14. Lukt het goed om de balans te houden tussen technisch werk, het wegwerken van technical debt en het functionele werk? Of is het elke sprint weer een gevecht tussen de ontwikkelaars en product owners?
15. Zijn de teams voldoende gefocust of worden ze continu lastig gevallen door mensen buiten het team?
16. Leeft het gevoel dat het team de stories samen zo snel mogelijk van links naar rechts over het bord krijgen?
17. Snappen de ontwikkelaars hoe ze bijdragen aan het business doel? Is het een POC of MVP en is de kwaliteit nu even niet belangrijk? Of bouwen ze iets dat nog jaren mee moet gaan?
18. Zien je ontwikkelaars het proces en de ceremonie eromheen als nuttig? Of voelen ze zich beperkt in hun vrijheid om hun werk in te richten zoals ze willen? En zijn het altijd dezelfde mensen die het proces een beetje (mentaal) saboteren?
19. Zijn de retrospecties vooral een saaie routine zonder dat er echt iets concreets uit komt? Of zijn het elke keer verrassende, interessante en nuttige events?
20. En ten slotte (maar niet minder belangrijk), zie jij Scrum of Kanban überhaupt als een nuttige techniek om meer controle te krijgen op de voortgang van jouw teams?

aviva solutions

www.avivasolutions.nl