KWALITEIT BY DESIGN

Focus op onderhoudbaarheid als methode om de aanpasbaarheid van de applicatie te borgen.

Door:
Diederik Wentink, Mendix Solution Architect (zelfstandig)
Rolf Bolt, co-founder en Mendix Solution Architect EGALiT

ONDERHOUDBAARHEID EEN ISSUE?

Organisaties besteden noodgedwongen een groot deel van hun IT-budget aan beheer en onderhoud van applicaties. Helaas brengt de komst van low-code oplossingen zoals Mendix hier geen verandering in. Sterker nog: wanneer een low-code project niet goed aangepakt wordt, zal de onderhoudbaarheid zelfs slechter worden dan bij traditionele softwareontwikkeling.

Wanneer je als organisatie focust op de onderhoudbaarheid van software heeft dit veel positieve effecten. Maar wat zijn de factoren waar deze focus zich op moet richten? Welke effecten heeft de focus?

In dit artikel geven we antwoorden op deze vragen. Naast de antwoorden leest u ook een concrete oplossing om in Mendix ontwikkelde software onderhoudbaar en flexibel te maken. Deze oplossing is in de afgelopen jaren door EGALiT bedacht en omvat een lagenarchitectuur in de software.

SOFTWARE DE DRAAK IN DE KELDER

Op een dag ga je naar de dierenwinkel en bestel je een leuk klein hagedisje voor in je terrarium. Dit diertje voed je met meelwormen en laat je trots aan je kinderen en aan vrienden zien. Maar het reptiel wordt groter en heeft nu krekels nodig om in leven te blijven.

Enige tijd later moet je het dier muizen gaan voeren en past het niet meer in je terrarium. Je besluit om een grotere kooi in je kelder te bouwen en het draakje daar naartoe te verhuizen. Inmiddels eet het ratten en konijnen en vraag je je af of je het nog wel zo'n leuk huisdier vindt.

Op het moment dat je het schapen moet gaan voeren wil je het liefst van je huisdier af. Maar je bent bang dat als je stopt met voeren, de draak uit de kelder komt en jou opvreet!

OORZAKEN VAN GEBREK AAN KWALITEIT

Uit een onderzoek van Deloitte uit 2018 blijkt dat meer dan de helft van het IT-budget van organisaties opgaat aan onderhoud van applicaties[1]. Geldt dit ook voor applicaties die in een low-code development omgeving zijn gerealiseerd?

SPEED SPEED SPEED…

De laatste 10 jaar heeft low-code development met o.a. Mendix een vlucht genomen ten opzichte van traditioneel ontwikkelen in bijvoorbeeld Java of .Net[2]. Vanwege de out-of-the-box functionaliteit en het genereren van programmatuur gaat ontwikkelen veel sneller met low-code. Mendix claimt dat het ontwikkelen van software 6x sneller gaat dan in traditionele ontwikkeling[3].

Met deze snelheid van ontwikkelen kun je razendsnel een eerste versie van je applicatie neerzetten. Het ontwikkelen in low-code sluit dan ook naadloos aan op de DevOps gedachte: hoe kan ik zo snel mogelijk een idee van de business omzetten in een werkende "service" voor de eindgebruiker? Helaas vergeten de teams en het

[1] IT Spending: From Value Preservation to Value Creation
[2] Enterprises are flocking to low-code tools, Mendix reports
[3] Rapid Application Development Tools & Software - Low-Code Platform

management daarbij vaak maatregelen te treffen zodat ook in vervolgfasen van het project de snelheid van het voortbrengingsproces hoog blijft...

CITIZEN DEVELOPERS

Een Mendix-project wordt vaak, net zoals traditioneel maatwerk, steeds moeilijker te onderhouden naarmate de omvang toeneemt. Een verklaring hiervoor kan zijn dat de Mendix-modelleurs vaak geen klassiek geschoolde programmeurs zijn. Het ontwikkelen in Mendix is laagdrempelig zodat dit door mensen gedaan worden die meer affiniteit met de business hebben dan met IT (*citizen developer*[4]).

GEBREK AAN ONTWERP, ARCHITECTUUR EN CODESTANDAARDEN

Een andere oorzaak is dat teams binnen de 'Agile' aanpak onvoldoende rekening houden met de architectuur. Door de hoge ontwikkelsnelheid is een Mendix-team qua omvang vaak beperkt. Daarom wordt gewerkt met *T-shaped* medewerkers die zowel software bouwen, als de ontwerp- en architectuurrol invullen. Het ontwerp wordt 'on the fly' gedaan waarbij vaak onvoldoende aandacht is voor het goed structureren van het systeem en het borgen van kwaliteit. Daarnaast zorgt het gebrek aan codestandaarden voor wildgroei en code duplicatie.

GEVOLGEN

Deze factoren maken de kans groot dat een substantieel deel van het IT-budget aan onderhoud opgaat. Door de eigenschappen van een low-code platform is de kans zelfs groter dan bij traditionele software-ontwikkeling.

Dit artikel benoemt een aantal uitgangspunten waar de oplossing aan moet voldoen om te voorkomen dat uw bedrijfssoftware een oncontroleerbare en onstuitbare kostenpost wordt. Daarna benoem ik een aantal belangrijke effecten van deze uitgangspunten die u kunt verwachten.

MISVERSTANDEN

Low-code platforms worden vaak gezien als productiviteitsverhogend, omdat het ontwikkelen in het platform zo 'gemakkelijk' gemaakt is: iedereen met enige affiniteit met softwareontwikkeling kan meewerken.

SOFTWARE COMPLEXITEIT IS NIET HETZELFDE ALS BUSINESS COMPLEXITEIT

Wat vergeten wordt is dat de tooling wel eenvoudig is, maar de businessproblematiek waarvoor de automatisering een oplossing biedt behoorlijk complex kan zijn. Om die te kunnen doorgronden is een sterk ontwikkeld vermogen tot conceptueel denken nodig en een drive om te zoeken naar eenvoudige oplossingen. Het is dus een misverstand dat het inzetten van Mendix in complexe omgevingen met slechts citizen developers en junior ontwikkelaars een goed resultaat gaat opleveren.

BUSINESS PROCESSTAPPEN NIET STAPSGEWIJS ONWIKKELEN

Het is ook een misverstand dat je goede software kunt ontwikkelen zonder intensief aandacht te besteden aan de structuur van gegevens en de opbouw van de code. Low-code tooling nodigt de ontwikkelaar uit om te denken in processen in plaats van in datastructuur, functies en eigenschappen. Veelal bouwt de low-code ontwikkelaar software door een (complex) business proces 'stapje voor stapje' te realiseren. De benodigde eigenschappen worden on-the-fly toegevoegd aan het gegevensmodel. Hierdoor verdwijnt het overzicht over de juiste werking en

[4] Definition of Citizen Developer - IT Glossary

uitkomst van het business proces. Het gevolg is dat toevoegen van nieuwe eigenschappen veel (analyse) tijd kost en tot regressie problematiek leidt. Een ander gevolg is dat er vrijwel geen uitspraken gedaan kunnen worden over de juistheid van de uitkomst, anders dan door het programmeren van een groot aantal testgevallen.

UITGANGSPUNTEN OPLOSSING

Het belangrijkste principe om maatwerkbedrijfssoftware onder controle te krijgen en te houden is door nadruk te leggen op het kwaliteitsaspect 'onderhoudbaarheid'[5]. Deze focus vraagt om aandacht voor de volgende uitgangspunten:

- Separation of concern (scheiden en helder opdelen van bedrijfslogica en interactie / navigatie) – Bouw bedrijfslogica maar een keer en zorg dat deze op verschillende plekken vanuit bijvoorbeeld schermen aanroepbaar is.
- Reductie complexiteit van code (KISS) – Deel complexe code op in kleinere, minder complexe eenheden.
- Uniformiteit in codering – Zorg dat alle code van alle modelleurs dezelfde structuur heeft.
- Garanderen van datakwaliteit – Scheid bedrijfsregels van de applicatielogica en zet ACID (atomicity, consistency, isolation, durability) principes in om deze bedrijfsregels af te dwingen.
- Traceerbaarheid van requirements – Leg in de code is een relatie naar het requirement vast.
- Laagdrempelig automatisch testen – Voer automatische testen niet alleen op de user interface uit, maar ook op de onderliggende logica.

EFFECTEN VAN ONDERHOUDBAARHEID

Dit hoofdstuk beschrijft een aantal belangrijke effecten die optreden als een organisatie bovenstaande uitgangspunten toepast.

KOSTEN EN PRODUCTIVITEIT ONTWIKKELING

Om onder een gestandaardiseerde en gestructureerde architectuur te werken, zijn extra inspanningen nodig. Dit lijkt in eerste instantie meer tijd te kosten dan ongestructureerd werken. Echter, zelfs bij het initieel ontwikkelen van een systeem zal er tijdswinst optreden. Dit komt doordat verreweg het grootste deel van de tijd gaat zitten in het ontwikkelen van de meest complexe delen van het systeem. En de architectuur zorgt voor minder complexiteit.

Vergelijk een legpuzzel van 100 stukjes met een puzzel van 1000 stukjes. De kleinere puzzel zult u in uw eentje in een half uurtje tijd wel op kunnen lossen. De puzzel van 1000 stukjes zal wellicht een aantal mandagen kosten. Wanneer u een methode vindt die een half uur voorbereidingstijd toevoegt aan het oplossen van een puzzel, maar vervolgens de oplostijd halveert (bijvoorbeeld puzzelstukjes sorteren), dan zult u zoveel tijdwinst boeken bij het oplossen ervan, dat de extra tijd zich gemakkelijk terugverdient

Standaardisatie van naamgeving vergemakkelijkt het vinden van gerealiseerde code en stimuleert daardoor hergebruik van code. Standaardisatie en een juiste opdeling vergemakkelijkt ook het uitbreiden of wijzigen van een bestaand requirement: een ontwikkelaar hoeft niet alle code te analyseren maar kan vertrouwen op de juiste werking van het systeem na zijn wijziging.

Een bijkomend voordeel is dat met gestandaardiseerd gebouwde software veel makkelijker automatische unit- en API-tests gedaan kunnen worden dan met ongestructureerde software. Deze testen vormen

[5] ISO 25010 'Maintainability'

het grootste (en goedkoopste) deel van de testpyramide[6].

Bij doorontwikkeling van software worden de voordelen in kosten en productiviteit nog groter:

- Door het scheiden van logica en interactie / navigatie zal minder regressie optreden. Door de mogelijkheid om automatisch te controleren op dataconsistentie, worden fouten in productiedata met kostbare herstelacties voorkomen.
- Traceerbaarheid van requirements zorgt voor een snellere probleemanalyse bij requirementwijzigingen.
- Geautomatiseerde unit- en API-tests zijn makkelijker te onderhouden dan automatische UI tests.

TIME-TO-MARKET

De voordelen die genoemd zijn bij de kostenreductie leveren ook een bijdrage aan een kortere time-to-market van wijzigingen aan het systeem.

Dit begint bij traceerbaarheid van requirements: deze maakt een snelle analyse van de impact van een voorstel tot wijzigen mogelijk. Dit leidt tot minder fouten tijdens de realisatie. Ook zijn deze fouten sneller oplosbaar: doordat de teams op een gestructureerde wijze code met een lagere complexiteit bouwen kunnen ze profiteren van de voordelen die in de vorige paragraaf beschreven zijn.

Verder hoeft een release minder lang getest te worden voordat deze naar productie gezet kan worden. De controle op data-integriteit voorkomt dat een uitgebrachte release teruggetrokken moet worden vanwege fouten in data (*rollback* van de release). Als er onverhoopt toch een fout in de release zit, dan zorgt de goede analyseerbaarheid van de gestructureerde code ervoor dat fouten snel gevonden en opgelost kunnen worden.

UITBREIDBAARHEID EN SCHAALBAARHEID

Door bedrijfslogica goed op te delen is het mogelijk om delen van de applicatie van elkaar te ontkoppelen. Deze ontkoppeling zorgt ervoor dat de applicatie makkelijker onder te verdelen is in meerdere microservices die elk een eigen doel hebben. Het opdelen van een applicatie in microservices zorgt voor een betere uitbreidbaarheid en schaalbaarheid van de applicatie, omdat bijvoorbeeld de microservices over verschillende servers verdeeld kunnen worden.

Het uniform werken door teamleden zorgt ervoor dat de medewerkers aan alle applicaties kunnen werken met een korte inwerkperiode. Een team kan daardoor snel opschalen.

FOUTGEVOELIGHEID EN CORRECTHEID

In veel applicaties is de bedrijfslogica verweven met de gebruikersinteractie. Daardoor kan het voorkomen dat de applicatie verschillende resultaten geeft als dezelfde functie op verschillende plekken uitgevoerd wordt. Dit voorkom je door bedrijfslogica separaat te realiseren. De bedrijfslogica kan los van de rest van de applicatie (automatisch) getest worden om de werking te blijven waarborgen.

Het terugbrengen van complexiteit zorgt voor minder fouten. Lage complexiteit en apart realiseren van bedrijfslogica zorgt voor het makkelijker opsporen en analyseren van fouten. Ook de uniforme werkwijze draagt hieraan bij. Doordat alle programmatuur dezelfde structuur heeft, kan een medewerker eenvoudig de code van zijn collega reviewen en fouten opsporen.

LEERCURVE NIEUWE MEDEWERKERS

Voor junior medewerkers is het prettig om in een vaste structuur te werken. Dit geeft houvast bij het realiseren van de software. Een organisatie kan

[6] Mike Cohn “Succeeding with Agile: Software Development Using Scrum”

ervoor kiezen om medewerkers met weinig ervaring alleen aan schermen en navigatie te laten werken Dit kunnen junior medewerkers zijn, maar ook mensen die meewerken vanuit de business (*citizen developers).* Omdat er geen bedrijfslogica in schermen zit, zal de onervaren medewerker niet vastlopen in de complexiteit of fouten introduceren die leiden tot verkeerde data.

Door de vaste structuur kan een nieuwe medewerker snel waarde toevoegen aan een ander team. En omdat de requirements traceerbaar zijn, is het voor medewerkers die onbekend zijn in het systeem toch mogelijk om snel te begrijpen hoe de logica van het systeem werkt. Dit biedt flexibiliteit in het schuiven met teamgroottes of het wisselen van de applicaties waar een team aan werkt.

LAGENARCHITECTUUR ALS OPLOSSING

EGALiT heeft in de afgelopen jaren een oplossing bedacht om in Mendix ontwikkelde software onderhoudbaar en flexibel te maken. Deze oplossing is in de basis een architectuur die bestaat uit een lagenmodel waarbij elke laag in de software een specifiek omschreven eigen verantwoordelijkheid heeft (separation of concern).

MVC-LIKE PATTERN

Het lagenmodel is een gedetailleerdere uitwerking van het Model-View-Controller (MVC) patroon dat al lange tijd gebruikt wordt in traditionele softwareontwikkeling.

- In het Model wordt alle bedrijfslogica gedefinieerd: validaties, business rules en bewerkingen / berekeningen op de data.
- De View betreft alle communicatie naar buiten. Dit kunnen schermen zijn, interfaces met andere systemen maar ook gebeurtenissen met een tijd-trigger.
- De Controller verzorgt het in de juiste volgorde uitvoeren van de business rules en het wegschrijven naar de database. Door de Controller worden de ACID-principes afgedwongen.

Het Model bestaat uit een aantal sublagen:

- Selecties: ophalen van gegevens uit het geheugen of de database.
- Manipuleren van objecten en gegevens (operaties, functies, afleidingen, processen en processtappen).
- Valideren van gegevens op geldigheid bedrijfsregels.

View bevat onder andere:

- De definitie van de interfaces (webservices, andere koppelingen maar ook schermen).
- De afhandeling van scherm interactie (navigatie, scherm-events etc).

De Controller bestaat typisch maar uit één laag die aangeroepen wordt vanuit de View en die de verschillende lagen uit het Model combineert.

RANDVOORWAARDEN

Het opdelen van de Mendix-code in lagen gebeurt op basis van afspraken. Er is geen extra software nodig om het lagenmodel toe te passen. Wat wel nodig is:

- Heldere afspraken en regels rondom het bouwen van software.
- Goede instructie en opleiding voor alle modelleurs.
- Een reviewproces dat toeziet op het toepassen van de afspraken en regels.
- Een “lagenmodel-champion” in het team: de modelleur/architect die kan helpen bij het correct toepassen van de lagenarchitectuur in complexe en afwijkende gevallen.

WAAROM DOET NIET IEDEREEN DIT AL?

Zoals al eerder genoemd, hebben Mendix-modelleurs vaak geen achtergrond in programmeren en architectuur. De drempel voor werken in Mendix is laag, waardoor citizen developers ook code kunnen bouwen. Door het gebrek aan kennis en achtergrond wordt niet

gedacht aan het opzetten van een uniforme structuur.

Om een goede software architectuur te realiseren is het noodzakelijk dat er voldoende IT- en architectuurkennis in een team aanwezig is. Maar op dit moment is deze kennis schaars waardoor teams van start gaan met te weinig senioriteit en kennis.

Mendix-modelleurs zijn vaak geen architecten. Door een focus op de korte termijn worden architectuurprincipes niet belangrijk genoeg gevonden door modelleurs. Ervaren modelleurs voelen zich beperkt in hun creativiteit en werkwijze door het toepassen van een uniform framework en zullen het belang pas na onderhouds- en uitbreidbaarheidsproblemen gaan onderkennen.

Zelfs in een nog jong project kan al een forse 'technical debt' zijn ontstaan. De ervaring leert dat een bestaande low-code applicatie achteraf lastig onder architectuur te brengen is. Niet alleen vanwege complexe code, maar vaak ook omdat het onder architectuur brengen een investering vraagt van geld en tijd zonder dat dat dit direct nieuwe businessfunctionaliteit oplevert. Dit stuit vaak op onbegrip van de business die toch een applicatie heeft 'die werkt'.

Overigens is het vaak wel mogelijk om een applicatie deels onder architectuur te brengen. Dit kan een oplossing zijn om het lagenmodel te introduceren op reeds bestaande software. Voorwaarde hierbij is dat deze delen weinig interactie hebben en er een heldere aanpak gevolgd wordt.

CONCLUSIE EN VERVOLG

Om meer mensen bewust te maken van het belang van werken onder een uniforme architectuur hebben we dit artikel geschreven. De investering die nodig is om een uniforme architectuur initieel op te zetten is laag en op de lange termijn zijn de voordelen groot. Door bij de opbouw van de Mendix-expertise in je organisatie te kiezen voor werken onder architectuur, kun je de medewerkers vanaf het begin meenemen in deze denkwijze.

In een vervolgartikel gaan we meer in detail in op het lagenmodel en hoe dit toe te passen in Mendix. Ook lichten we een aantal standaard-patronen toe, bijvoorbeeld voor statusovergangen.

Tot die tijd kunt u voor meer informatie contact met ons opnemen.

Diederik Wentink, diederik@wentinksolutions.nl

Rolf Bolt, r.bolt@egalit.nl

Voor meer informatie over de diensten van EGALiT kunt u contact opnemen met Markus Travaille m.travaille@egalit.nl 06-19975135