POLSAG:
Code review

4. januar 2012

Martin Strandbygaard, Mikkel Christensen, Jesper Hvid, Jakob Røjel & Morten Strunge Nielsen

POLSAG Code Review

Ledelsesresumé

Dette dokument har til formål at samle de væsentligste tekniske informationer om POLSAG-applikationen, der er fremkommet i forbindelse med review'et af programkoden. Disse informationer er indsamlet over to separate forløb:

1. Det tekniske review af POLSAG, som blev gennemført i forbindelse med POLSAG-undersøgelsen, som Boston Consulting Group (BCG) udførte for Rigspolitiet, Justitsministeriet og Finansministeriet i foråret 2011 med Globeteam som underleverandør.
2. POLSAG Code review som blev gennemført af Globeteam i anden halvdel af november måned 2011. Code review'et byggede videre på det tekniske review af POLSAG. Code review'et er formelt set udført på bestilling af Rigspolitiet med Devoteam som leverandør med Globeteam som underleverandør. Efter aftale med Rigspolitiet har Devoteam ikke været involveret i udarbejdelsen og kvalitetssikringen af POLSAG Code review.

I begge tilfælde har reviewet været afgrænset til POLSAG med tilhørende grænseflader. Randsystemerne , ESB'en samt evt. andre fritstående applikationer og komponenter er således ikke omfattet af review'et.

Formål og afgrænsning af opgaven

Opgaven har til formål at gennemgå og dokumentere POLSAG-programkoden samt relevant afledte tekniske områder, såsom dokumentationen og arkitekturen.

Afdækningen af den tekniske side har været koncentreret om at få belyst følgende hovedområder:

- Arkitekturen for løsningen
- Evaluering af den POLSAG-specifikke kode (baseret på stikprøver)
- Udviklingsdokumentation

Opgaven har som sådan haft til formål at analysere POLSAG-programkoden, ikke afteste eller udføre rettelser i samme. Det har ligeledes ikke været en del af opgaven at forholde sig til historikken for udviklingen af systemet.

Beskrivelserne i dette dokument er baseret på en gennemgang af den fremlagte POLSAG-dokumentation samt interviews med en stor del af projektets nøglepersoner. Alle interviews blev udført i forbindelse med det tekniske review i foråret 2011.

POLSAG Code Review

Ledelsesresumé

Dette dokument har til formål at samle de væsentligste tekniske informationer om POLSAG-applikationen, der er fremkommet i forbindelse med review'et af programkoden. Disse informationer er indsamlet over to separate forløb:

1. Det tekniske review af POLSAG, som blev gennemført i forbindelse med POLSAG-undersøgelsen, som Boston Consulting Group (BCG) udførte for Rigspolitiet, Justitsministeriet og Finansministeriet i foråret 2011 med Globeteam som underleverandør.
2. POLSAG Code review som blev gennemført af Globeteam i anden halvdel af november måned 2011. Code review'et byggede videre på det tekniske review af POLSAG. Code review'et er formelt set udført på bestilling af Rigspolitiet med Devoteam som leverandør med Globeteam som underleverandør. Efter aftale med Rigspolitiet har Devoteam ikke været involveret i udarbejdelsen og kvalitetssikringen af POLSAG Code review.

I begge tilfælde har reviewet været afgrænset til POLSAG med tilhørende grænseflader. Randsystemerne , ESB'en samt evt. andre fritstående applikationer og komponenter er således ikke omfattet af review'et.

Formål og afgrænsning af opgaven

Opgaven har til formål at gennemgå og dokumentere POLSAG-programkoden samt relevant afledte tekniske områder, såsom dokumentationen og arkitekturen.

Afdækningen af den tekniske side har været koncentreret om at få belyst følgende hovedområder:

- Arkitekturen for løsningen
- Evaluering af den POLSAG-specifikke kode (baseret på stikprøver)
- Udviklingsdokumentation

Opgaven har som sådan haft til formål at analysere POLSAG-programkoden, ikke afteste eller udføre rettelser i samme. Det har ligeledes ikke været en del af opgaven at forholde sig til historikken for udviklingen af systemet.

Beskrivelserne i dette dokument er baseret på en gennemgang af den fremlagte POLSAG-dokumentation samt interviews med en stor del af projektets nøglepersoner. Alle interviews blev udført i forbindelse med det tekniske review i foråret 2011.

Konklusion

Code reviewet har afdækket følgende overordnede problemer i POLSAG-koden:

- Kompleks kildekode i POLSAG-klienten – 18% af kildekoden til POLSAG-klienten er svær at fejlsøge, teste og vedligeholde, fordi den er for kompleks, og heraf er 31% meget kompleks og dermed meget svær at fejlsøge, teste og vedligeholde. Det forhold at JavaScript er et typesvagt sprog medfører at øget kompleksitet har en væsentligt større negativ impact for udvikleren end ved andre sprog. Kodekompleksiteten er opstået under udviklingsprocessen og vurderes at være helt unødvendig givet kompleksiteten af POLSAG-løsningen.
- Kompleks kildekode i POLSAG-serverapplikationen – 34%[1] af kildekoden til POLSAG-serverapplikationen er svær at fejlsøge, teste og vedligeholde, fordi den er for kompleks[2]. Heraf har har 56% af kildekoden (20% af kildekoden til POLSAG-serverapplikationen) en alt for stor kompleksitet[3], hvilket betyder at den må betegnes som værende meget svær at fejlsøge, teste og vedligeholde. Dette vurderes som værende særdeles kritisk for POLSAG-projektet, idet den komplekse hhv. meget komplekse kildekode er koncentreret i kerneforretningslogikken (BusinessProcessHandlers), hvor 42% hhv. 25% af kildekoden til kerneforretningslogikken er kompleks hhv. meget kompleks. Kodekompleksiteten er opstået under udviklingsprocessen og vurderes at være unødvendig givet kompleksiteten af POLSAG-løsningen.
- Dårlig organisering af kildekoden i POLSAG-klienten – 50% af kildekoden til POLSAG-klienten er samlet i kun 18% af metoderne, hvilket betyder at kildekoden til POLSAG-applikationen mangler organisering. Webklienten (Javascript) er overvejende baseret på procedural implementeringsteknik. Begge dele vil uværgerligt medføre forøgede omkostninger til at fejlsøge, teste og vedligeholde systemet.
- Dårlig organisering af kildekoden i POLSAG-serverapplikationen – 50% af kildekoden til POLSAG serverapplikationen er samlet i kun 5% af metoderne (19% af kildekoden er samlet i kun 0,7% af metoderne), hvilket betyder at kildekoden til POLSAG applikationen mangler organisering. Kerneforretningslogikken (de funktioner som er beskrevet i SDD'erne og udstillet til klienten gennem Business Process Handlers) er overvejende baseret på procedural implementeringsteknik. Begge dele vil uværgerligt medføre forøgede omkostninger til at fejlsøge, teste og vedligeholde systemet.
- Meget lavt testniveau – Antallet af unit tests til POLSAG er virkelig lavt (kontrol og funktionsklasser har mindre end een unit test per klasse, hvor normen er flere tests per metode, f.eks. 20-30 tests for en klasse med 10 metoder), og POLSAG vurderes at være opbygget på en måde, der gør det svært at lave unit tests til forretningslogikken uden en vis refactoring af samme. Det meget lave antal unit tests medfører forøgede omkostninger ved implementering af ændringer og nyudvikling, fordi en udvikler ikke løbende i den daglige udviklingsproces kan verificere om der introduceres nye fejl. Med den nuværende udviklingsproces, vil en udvikler tidligst dagen efter blive gjort opmærksom på, at der er introduceret en fejl, samtidig med at den fejlbehæftede kode er distribueret til alle øvrige udviklere. Der er ingen garanti for at den funktionelle test (QTP) finder

[1] Alle procentangivelser ifht. kildekode, er beregnet ud fra SLOC

[2] Defineret ved cyklomatisk kompleksitet > 15 eller lavt maintainability indeks og mange advarsler i værktøjer (FxCop, Code Analysis og JSLint) til statisk analyse af kildekode.

[3] Defineret ved cyklomatisk kompleksitet > 30

fejlen, da denne formodes at være langt fra at være fuldt funktionelt dækkende[4], idet det erfaringsmæssigt er meget vanskeligt at teste fejlhåndtering automatisk fra en brugergrænseflade.

- Mangler i webklientens transaktionsstyring (Javascript) – Der er under code review'ets stikprøver blevet identificeret ni steder i koden, hvor den anvendte transaktionsstyring ikke sikrer effektivt mod inkonsistens i data i forbindelse med fejl i webklienten.
- Kildekoden har et lavt dokumentationsniveau – Kildekoden indeholder et særdeles lavt niveau af kommentarer og øvrig information. SDD'erne, som er den primære dokumentation for POLSAG-funktionaliteten, er meget implementeringsnære (de indeholder eksempelvis detaljerede udspecificeringer af algoritmerne). Det vurderes at være svært at udlede de forretningsmæssige krav og ønsker på basis af SDD'erne, især i forbindelse med ændringer, der involverer mange SDD'er, hvilket gør dem væsentligt mindre brugbare i den fremadrettede udvikling.
- Dateret udviklingsproces og værktøjer – Udviklingsprocessen er ikke løbende blevet opdateret med forbedrede udviklingsværktøjer. Specielt giver de anvendte værktøjer giver kun meget begrænset understøttelse af Javascript-programmeringssproget, selvom webklienten er implementeret i Javascript og JavaScript udgør ca. halvdelen af POLSAG-kodebasen. Dette betyder at man giver afkald på de kvalitetsmæssige og produktivitetsmæssige gevinster, det ligger i de nyere processer og værktøjer.
- Løbende opdatering af teknologisk platform – Flere af de anvendte Microsoft kerneteknologier er baseret på ældre versioner, hvilket betyder at projektet ikke høster de forbedringer, der er sket på Microsofts udviklingsplatform i form at højere udviklerproduktivitet, behov for reduceret kodemængde eller øget fleksibilitet.

Baseret på kvantitative parametre som antal kodelinjer, cyclomatisk kompleksitet og vedligeholdelsesindeks samt det kvaliteten parameter af dokumentationen vurderes kildekoden samlet set at være svær til meget svær at vedligeholde. Statisk kildekodeanalyse med udbredte værktøjer (Microsoft Code Analysis, FxCop og JSLint) angiver i øvrigt, at kildekoden ikke overholder de gængse practices for navngivning, struktur og organisering.

Givet problemdomænets natur og omfang vurderes det sammenfattende, at POLSAG-løsningen samlet set er væsentligt mere kompleks end behovene tilsiger..

Løbende drift og vedligeholdelse

Baseret på antallet af automatiserede unit tests kan det konkluderes, at der kun kan ske en yderst behersket aftestning af POLSAG-løsningen i forbindelse med udviklingen, hvilket betyder at den efterfølgende funktionelle test bliver unødig dyr og ressourcekrævende, når den skal gardere mod fejl og mangler i applikationens funktionalitet.

Det er således langt mere omkostningsfyldt at opnå samme grad af testdækning af kildekoden med en rent funktionel test: Et tænkt eksempel: Hvis POLSAG funktionalitet anvender 3 metoder/funktioner som hver har 4 mulige input parametre, så kræver det 12 unit tests (4+4+4) at opnå fuld testdækning, mens den funktionelle test i værste fald vil kræve 64 tests (4 * 4 *4). I praksis vil unit tests dog skulle suppleres med en integrationstest og det kan tænkes at nogle af de 64 varianter i den funktionelle test kan elimineres på basis af en evaluering af problemdomænet.

[4] Der findes ikke nogen tal for hvor stor en del af kodebasen den funktionelle test dækker.

Der er etableret en leverandørspecifik funktionel test (QTP-testen), som leverandøren selv angiver kommer i berøring med 75% af POLSAGs funktionalitet. Der findes ikke nogen evaluering af hvor stor en del af kildekoden (code coverage) der omfattes af den funktionelle test.

Dette forhold kombineret med det lave antal automatiserede unit tests vurderes bl.a. at være en vigtig faktor for antallet af fejl på systemet og udvirker en betydelig større usikkerhed på antallet af fejl, som endnu ikke er blevet fundet (samt risikoen for at der introduceres regressionsfejl i forbindelse med fejlretningen). De manglende unit tests vanskeliggør ligeledes muligheden for at sætte effektivt ind for at nedbringe kodekompleksisten gennem refactoring.

De automatiserede tests (unit tests) som findes i POLSAG dækker som nævnt kun en meget lille del af den samlede kildekode. Verifikation af systemets funktionalitet må således nødvendigvis henlægges til den manuelle tests eller den leverandør-specifikke funktionelle tests.

Kildekoden i POLSAG er karakteriseret ved at indeholde en del kompleks forretningslogik og et meget lavt dokumentationsniveau. Selvom funktionaliteten er beskrevet i de tilhørende SDD'ere vurderer vi at der behov for gode kommentarer i koden for at muliggøre at andre udviklere kan overtage vedligeholdelsen af systemet på en sikker og effektiv måde.

POLSAG-løsningen vurderes samlet sét at indebære en væsentlig forøgelse i omkostningerne til support og vedligeholdelse. Denne vurdering bunder i høj grad på erkendelserne i dette afsnit såvel som følgende observationer:

- De fleste fejl i POLSAG må formodes tidligst at blive opdaget relativt sent i udviklingsforløbet og i mange tilfælde i forbindelse med de efterfølgende testforløb eller efter idriftsættelse.
- Løsningen indeholder et ukendt antal resterende fejl grundet den meget lave unit test-dækning og de deraf følgende problemer med sikring mod regressionsfejl og lignende.
- POLSAG er karakteriseret ved en vanskelig fejlfindingsproces i de områder af løsningen, der er behæftet med en høj kodekompleksitet.
- POLSAG må vurderes at være behæftet med en lang indlæringskurve for nye udviklere på projektet. Det forhold at koden er "spredt" over i alt fire programmeringssprog (JavaScript, C#, C++ og PL-SQL) gør det kun yderligere svært for nye udviklere at arbejde sig ind i løsningen.

POLSAG-løsningen hviler på følgende teknologier:

- .Net Framework v2 – En ældre version af Microsoft .Net Framework, som er udgået af almindelig support (support i form af Service Packs ophørte allerede i januar 2009), og kun er omfattet af Microsofts forlængede supportfase ("extended support phase"), hvilket betyder at der kun udgives sikkerhedsopdateringer samt at løsningen er underlagt et supportprogram af begrænset omfang og dækning. Denne version mangler i øvrigt en række af de nyere kerneteknologier i .Net platformen, f.eks. LINQ, som besidder potentialet til at forbedre effektiviteten og kvaliteten i udviklingsprocessen. Nyeste version af .Net Framework er version 4 fra 2010. Der er tre versioner (3.0, 3.5 og 4.0) ned til .Net Framework v2.
- ASMX – Web Service-løsning, der er koblet til ASP.Net. ASMX indeholder ingen indbygget funktionalitet til transaktionshåndtering og garanteret meddelelsesaflevering og er afgrænset til XML (den kan som sådan eksempelvis ikke håndtere binær serialisering). Selvom ASMX endnu

bliver supporteret så angiver Microsoft selv at det er en legacy-teknologi, som er blevet erstattet af WCF.
- JavaScript – Et programmeringssprog, som primært benyttes som scriptsprog i webbrowsere.

Teknologivalget har også dannet grundlag for det grundlæggende pattern, som karakteriserer hele løsningen: Kald sker med XML, der som typisk er direkte afledt af indholdet af skærmbilledet, via et RPC-agtigt kald over http.

ScanJour fremhæver, at dette pattern danner grundlag for et meget fleksibelt system under henvisning til at det bliver relativt nemmere at tilføje ekstra funktionalitet til XML'en. Det betyder imidlertid også, at klientsiden bliver koblet til serversiden på en ikke-type-stærk måde, som tillige ikke altid er reflekteret i dokumentationen. I et kodegenbrugsscenarie må det anbefales at arbejde på bedre dokumentation af denne kommunikationsmodel eksempelvis ved at introducere et message repository, der også kan benyttes til at compile time validering af sammenhængen. Relativt til behovet for væsentligt bedre tests på udviklerniveau mener Globeteam som minimum det er nødvendigt at etablere en løsning, som validerer at klienten anvender den korrekte XML og omvendt for at afbøde de risici, som den ikke-type-stærke fremgangsmåde er behæftet med.

Samlet set må vi karakterisere POLSAG som en traditionel database-centreret applikation, der er udvidet med XML og http teknologi. POLSAG kernesystemet er som sådan ikke en SOA-baseret løsning med løst koblede delkomponenter, modularisering, indkapsling, metadata, etc.

Anbefalinger

Rapporten indeholder en lang række delkonklusioner med tilhørende anbefalinger. Nedenfor er de væsentligt findings opsummeret sammen med en angivelse af, hvor i rapporten, anbefalingen stammer:

- Døgnrapporten bør gennemgå en omfattende refactoring (evt. omskrives) efter almindelige principper for softwareudvikling, så det sikres at den kan testes og vedligeholdes samt at performance er acceptabel for almindelig brug, jf. afsnittet "Døgnrapporten".
- Implementeringen af error logning er usammenhængende og sikrer ikke i alle tilfælde et tilstrækkeligt grundlag til at udføre fejlsøgning. Hele tilgangen til logging bør ændres så der indsamles tilstrækkelig information til fejlsøgning kan ske hurtigt og effektivt jf afsnittet Fejllogning.
- Javascript-kildekoden i webklienten bør refaktoreres efter almindelige principper for udvikling af løsninger i Javascript (se afsnittet Javascript).
- Mønster for transaktionsstyring i Javascript-kildekoden i webklienten skal ændres, så f.eks. nedbrud eller nedlukning af webklienten ikke kan medføre forretningstransaktioner der kun er delvist gennemført og dermed potentiel inkonsistens i data, jf afsnittet om Transaktionsstying.
- De kodemæssigt komplekse dele POLSAG bør refaktoreres. I den arbejdsgang bør man opdatere koden således at bliver lettere at teste de enkelte dele hver for sig, i stedet for at det som nu kræver et komplet kørende miljø (se afsntittet Test-drevet udvikling).

Det anbefales at rette op på POLSAG-koden jvf anbefalingerne. Alle væsentlige risici på kodeniveau i relation til udrulning og driftsstabilitet (døgnrapport, error logning og transaktionsstyring) bør være elimineret forud for implementeringen hos Rigspolitiet. De øvrige anbefalinger (refaktorering og

automatiseret test, herunder unit testing) bør ligeledes gennemføres af hensyn til vedligeholdelsen og dermed indirekte driftsstabiliteten.

Muligheder for forbedringer

Globeteam vil endvidere fremhæve følgende ting, som vi anser for at være oplagte og væsentlige forbedringer af POLSAG:

- Opgradering af kerne-teknologier til tidssvarende versioner. Dette giver højere produktivitet samt bedre muligheder for udvikling af POLSAG
- Indførelse af en ensartet udviklingsproces og krav til kodestandarder, herunder ikke mindst i relation til automatiseret test ved hjælp af unit tests. Det vurderes også at derer behov for bedre kommunikation af gældende practices for udvikling på POLSAG suppleret med en peer-review proces.
- Bedre anvendelse af udviklingsværktøjer til at understøtte udviklingsprocessen.
- Kvalitetssikring af det udførte arbejde gennem løbende automatiseret monitorering af kvaliteten i kodebasen.

Forudsætninger for læsning af rapporten

Rapporten forudsætter at læseren er bekendt med POLSAG-løsningen, herunder den overordnede arkitektur for POLSAG-løsningen og opbygningen af en løsning baseret på ScanJour CAPTIA.

Det forudsættes derudover at læseren er bekendt med Captia 4.2 samt POLSAG-projektet, herunder SOM, SOMASP og Captias extensibility punkter.

Såfremt læseren ikke opfylder ovenstående forudsætninger kan vi oplyse at den væsentligste dokumentation omkring disse emner er placeret i 'Captia Datamodel', 'Captia Guidelines and documentation', 'Overordnet POLSAG arkitektur' samt 'POLSAG Guidelines and documentation' mapperne på Baseline DVD'en.

Baggrund for dokumentet

Omdrejningspunkterne for det tekniske review var interviews med nøglepersoner hos ScanJour (SJ), CSC og Rigspolitiet (RP) samt inspektion og analyse af den POLSAG-specifikke kildekode og den dertilhørende evaluering af systemets arkitektur og platformens sammensætning.

Der har været afholdt interviews med nøglepersoner hos ScanJour, CSC og Rigspolitiet:

- ScanJour: Dennis Lauritzen (løsningsarkitekt), Jan Olsen (de facto udviklingslead), Lene Alhed Augenstenborg (tidligere ansvarlig for analyse-afdelingen).
- CSC: Lars Bech Rasmussen, Tom Jørgensen (drift), Mikkel Nielsen og Dorte Krogsgaard (funktionel test), Peter Koch Jensen og John G Thomassen (performance / tuning/ arkitektur)
- Rigspolitiet: Michael Englev (funktionel test), Brian Nielsen (leverandøransvarlig), Tommy Rueskov Sørensen (ansvarlig for performanceprojektet) og Peter Greifenstein (Chefarkitekt).

I forbindelse med inspektionen og analysen af kildekoden i det tekniske review var Globeteam i dialog med Jan Olsen, Martin Glob (ekstern konsulent hos ScanJour), Torben Frandsen (ekstern konsulent hos

ScanJour), Matthew Robert Graham (intern primær ressource på C++ og PL-SQL delen af POLSAG hos ScanJour) samt Johnny Sabinsky (QA hos ScanJour).

POLSAG Code Review er sket isoleret på basis af den nyeste release af POLSAG-kildekoden, der blev udleveret af ScanJour medio november 2011.

Risici og afgrænsninger

Analyserne og konklusionerne i code review'et er behæftet med følgende afgrænsninger/risici:

- Code review'et er afgrænset til den POLSAG-specifikke del af løsningen (excl. eksterne systemer og komponenter, herunder ESB'en og randsystemerne). Code review'et omfatter ikke nogen dele af ScanJour Captia, som POLSAG-løsningen bygger på.
- I forbindelse med gennemgang kildekoden kunne leverandøren ikke stille et komplet kørende udviklingsmiljø til rådighed, hvilket væsentligt har begrænset mulighederne for at analysere systemets virkemåde og udføre yderligere validering af en række detaljer.
- Gennemgangen medtager eksempler på de observationer der er gjort, men indeholder ikke udtømmende lister med alle forekomster af de enkelte observationer.
- Al rettighedsstyring i POLSAG sker i det underliggende Captia system og rettighedsstyring er derfor ikke omfattet af gennemgangen.
- POLSAG integrerer med en række af Scanjours COM-komponenter, f.eks. til opdatering af data i registermodellen. Kildekoden til disse COM-komponenter var ikke en del af baseline materialet og er derfor ikke omfattet af gennemgangen.
- POLSAG er baseret på et samspil med Captia register model med indbygget sikkerhed og en del PL-SQL. Der har været dialog med Scanjour omkring at denne custom kode overholder Captias regler og krav, men det er ikke i praksis muligt at verificere uden adgang til Captia-kildekoden.
- Code reviewet er afgrænset til det udleverede baseline materiale og omfatter ikke et fuldt kørende POLSAG-system, hvorfor der principielt ikke er nogen sikkerhed for at Globeteam har set al kode, der er omfattet af POLSAG-projektet.

Det er i øvrigt vigtigt at understrege at POLSAG er et meget stort og komplekst projekt, hvorfor der kan være informationer og detaljer i koden, som er forbigået vores opmærksomhed. Den manglende adgang til et kørende udviklingsmiljø, det forhold at koden ikke kunne compileres samt ikke mindst den manglende indsigt i kildekoden til Captia opsætter nogle naturlige begrænsninger for review'ets evne til at fange alle vitale kodedetaljer.

Indhold

Code review

Dette kapitel indeholder en overordnet analyse med tilhørende konklusioner af alle de områder, som er undersøgt ved inspektion eller afdækket under interviews. Detaljer og kodeeksempler er henlagt til de efterfølgende appendices.

Platform

De primære komponenter i POLSAG er:

- Windows XP (klient), Windows Server 2003 (applikationsserver)
- IE6 (klient), IIS 6 (applikationsserver)
- VS2008 (udviklingsværktøj)
- ASMX web services (dog WCF services til Sagsstyringreol)
- Captia 4.2
- Javascript
- .NET 2 (application server), dog .NET 3.5 for Sagsstyringreol
- C++
- PL-SQL

Nogle af disse er betinget af den anvendte version af Captia[5], andre er kontraktuelt fastlagt.

Det basale call pattern er RPC[6] mellem browser og web server. Klienten anvender en proprietær scripting syntax til at tilgå og opdatere data. Dataudveksling og fortolkning er baseret på magiske strengværdier, XML-parsing af HTML-forms og omfattende brug af late binding[7].

Der er observeres en meget høj grad af hård kobling fra klient til server på grund af disse valg.

Den eneste mulighed, der p.t. eksisterer til at udføre impact-analyser af ændringer består af tekst-baserede filsystemsøgninger i henholdsvis kodebasen og SDD-dokumenterne.

Konklusion: POLSAG er baseret på en teknologisk platform, der er inhomogen og har et væsentligt antal år på bagen. Det bemærkes specielt at dette tjener til at gøre den væsentlig mindre produktivt og uattraktivt at arbejde med for udviklerne.

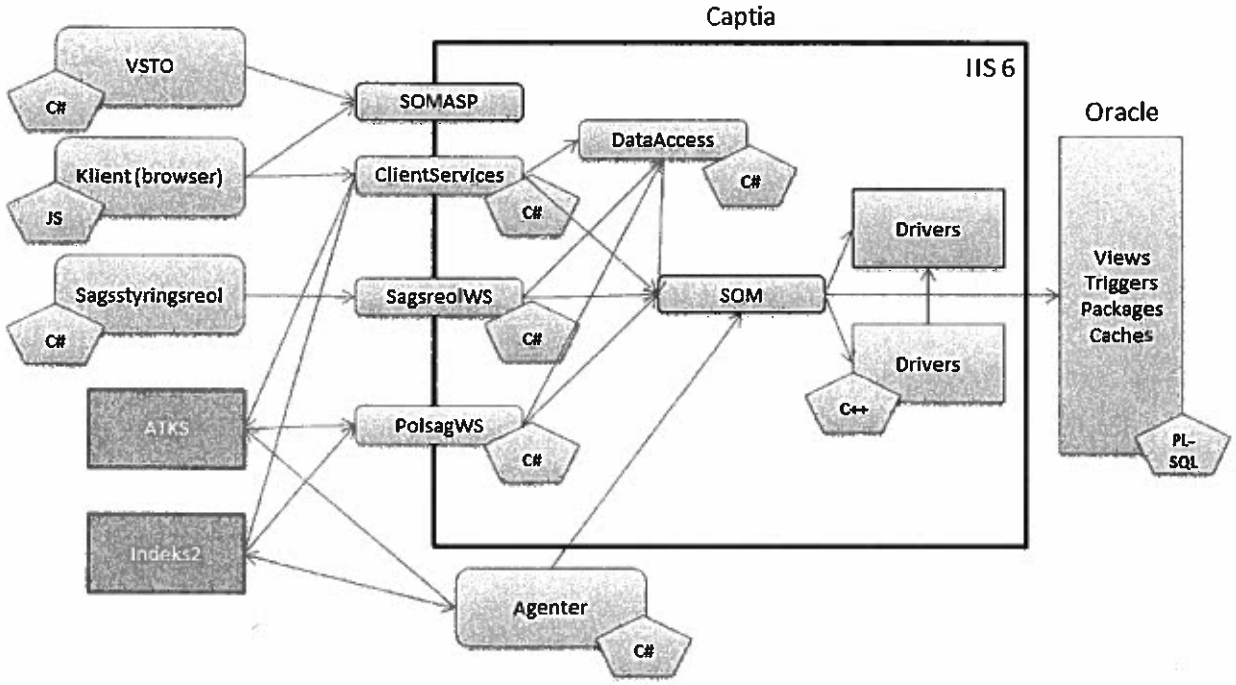
Mest iøjefaldende er bindingerne til .Net Framework 2 og IE6. Særligt anvendelsen af ASMX web services er problematisk, da ASMX-teknologien ikke er pålidelig og ikke indeholder replay detection. Løsningen er som nævnt karaktiseret ved mange små kald mellem browser og web server i en RPC-stil, hvilket erfaringsmæssigt kan give grobund for performanceproblemer og resultere i øgede hardwarebehov.

Valget af Captia i POLSAGs arkitektur, og sammenhængen mellem de til POLSAG specialudviklede delkomponenter er illustreret i nedenstående diagram.

[5] ScanJour Captia er baseret på legacy-teknologierne COM og ASP, hvilket har konsekvenser for de teknologier, der indgår i POLSAG.
[6] http://en.wikipedia.org/wiki/Remote_procedure_call
[7] http://en.wikipedia.org/wiki/Late_binding

- **SOMASP er ASP interface til SOM.**
- **ClientServices** er en Javascript-konsumerbar frontend til de eksterne systemer (der udstilles over HTTP-protokollen).
- **SagsreolWS er den WCF service som er udstillet til sagsreolklienten (implementeret i Silverlight)**
- **PolsagWS** er de ASMX web services som er udstillet fra POLSAG af hensyn til de eksterne systemer.
- **Batch-agenter** er Windows NT services som hostes af Captia og som udfører periodiske baggrundsjobs af både opdaterende og opryddende natur.
- **Drivers** er SOM-registerudvidelser som anvendes til enkelt-register data-valideringer.
- **'Views, Triggers, Packages, Caches'** dækker over de udvidelser til databasen som ikke nødvendigvis er tilgængelige via SOM Register-modellen, men som bl.a. finder anvendelse til performanceoptimeringer, følgeopdateringer af ændringer gennem Register-modellen, synkronisering af caches og håndtering af skema-migreringer.

De røde pentagoner angiver hvor der forefindes implementeringer af forretningslogik[8] samt i hvilket programmeringssprog, mens de grå bokse angiver komponenterne fra standard Captia.



Figur 1 POLSAG logisk komponent oversigt

Den øverste venstre boks angiver browser-klienten, og dennes indlejrede Javascript. Denne bør ret beset markeres som en delvis specialudviklet komponent, fordi en ikke triviel delmængde af Javascript-koden har relation til anvendelsen af hhv. BPM og ClientServices.

[8] Defineret som kildekode der implementerer regler og logik, der er specifikke for det problemdomæne, som POLSAG udgør.

Overordnet arkitekturvurdering

Systemets arkitektur kræver at en stor del af flow- og valideringslogikken både implementeres på serveren og i klienten, for at brugeren styres korrekt i gennem en transaktion, i stedet for at anvende en arkitektur der kun kræver at logikken implementeres på serversiden.

Systemet har ikke nogen skarpt adskilt lagdeling, når det kommer til implementering af forretningslogik på serveren, der implementeres i enten C#, C++ eller PL-SQL. Valget af placeringen af en given funktionalitet vurderes generelt at være blevet foretaget ad-hoc af den enkelte udvikler.

Den anvendte del af standard Captia's transaktionsstyring-model kan ikke håndtere opdateringer til mere end et register ad gangen, og denne begrænsning ville have gjort systemet uanvendeligt til Rigspolitiets anvendelses-scenarier. Derfor er der implementeret et transaktionslag oven på standard Captia systemet, til at håndtere dette. Det betyder, at arkitekturen indeholder flere niveauer, hvor der kan implementeres forretningslogik.

Der er ikke nogen overordnet strategi for, hvordan man sikrer sig at applikationen leverer den nødvendige svartid og generelle UI-respons. Performanceproblemer forventes løst hen ad vejen, når de opstår (iflg. Dennis Lauritzen, SJ), og CSC meldte i 2008, efter etableringen af performancekommisoriet, ud til RP at man ville vente med at kigge på dette område til senere (iflg. Brian Nielsen, RP). De første performancemålinger blev udført i foråret 2010.

Konklusion: POLSAG-platformens nuværende interne arkitektur pålægger udviklerne et stort ansvar for at træffe det rigtige valg omkring placering af nyudviklet funktionalitet. I forbindelse med dette valg skal der bl.a. ske en afvejning af hensynet til genbrug, vedligeholdelse og performance meget tidligt i udviklingsforløbet. Dette fordrer som minimum et højt niveau af udvikler guidelines og peer reviews, hvis det skal være muligt at sikre ensartethed og god kvalitet i disse beslutninger.

Strategien for hvordan man vil håndtere performance-problemer vil i nogle tilfælde vise sig at være anvendelig når det drejer sig om relativt trivielle fejl i koden. Eftersom den basale client-server kommunikation er baseret på et synkront (RPC-style) call pattern, som ikke besidder de bedste skaleringsegenskaber og som trækker mange ressourcer både netværksmæssigt og server-side må det forventes, at det vil være forbundet med store omkostninger at udbedre evt. performanceproblemer, der udspringer af dette forhold.

Transaktionsstying

Standard Captia har indbygget transaktionsstyring med den begrænsning at man kun kan opdatere ét register ad gangen.

I POLSAG er der implemeret en delkomponent, Business Proces Manager (ashx http handler), der løsner denne restriktion ved at være i stand til at koordinere opdateringer som involverer flere registre. Der er dog ingen centraliseret styring af anvendelsen af transaktioner, hvorfor der er observeret manuel transaktionsstyring spredt rundt omkring i kodebasen (dvs. begin/commit/rollback foretages eksplicit uden nogen form for overordnet koordinering). Dette er problematisk, fordi det gør det vanskeligere (for ikke at sige umuligt) at kunne genbruge eksisterende kode i nye sammenhænge, idet man risikerer at få committet delmængder af data som i nye sammenhænge ikke er konsistente, og man bliver hårdt bundet op på den eksisterende funktions forståelse af, hvornår noget går galt.

Endvidere betyder den manuelle transaktionsstyring, at fejl og uventede exceptions kan medføre at der ikke sker rollback. Implementeringen er derfor helt afhængig af at udviklereren har forudset de mulige fejlscenarier og i alle tilfælde lavet koden tilstrækkelig robust til at uventede fejlscenarier medfører at rollback bliver kaldt. Da den manuelle transaktionsstyring bl.a. sker i metoder på > 200 kodelinjer (f.eks. ClientServices.F0081.ProcessRequest) virker det ikke sandsynligt at denne præmis er opfyldt overalt.

F.eks. flg. udsnit af ClientServices.F0081.ProcessRequest:

```
1
2

3
4

5

6
7
8
9

10
11
12
13
14
15
16
17
18
19
20
21
22
23

24
```

ClientServices.F0081

I ovenstående eksempel, vil en fejl i underafvikling af forretningslogikken medføre at første catch-blok (linje 7) rammes. Hvis f.eks. en fejl i metoden "ReturnResult" (linje 14) her kaster en exception, så rammes næste

catch-blok (linje 18), og her vil endnu en excption fra "ReturnResult" medføre, at transaktionen ikke bliver rullet tilbage.

Et tilsvarende problem ses i POLSAG.Common.Functions.F0115.GemAfgorelse:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

POLSAG.Common.Functions.F0115

Problemet i dette eksempel er større end i foregående eksempel. Her skal der blot forekomme een uventet exception efter linje 5 i ovenstående, og så vil transactionen ikke blive rullet tilbage (pga. tjek i linje 14) som den ellers bør blive det. Der forekommer 417 linjer med forretningslogik efter linje 5 inden catch-blokken.

Endvidere ses samme mønster i Javascript kodebasen, hvor den samme Javascript funktion opdaterer POLSAG flere gange fra samme metode (se afsnit **Fejl! Henvisningskilde ikke fundet.**)

Konklusion: Transaktionsstyringen bør centraliseres så de enkelte funktioner mv. ikke har ansvaret for at afgøre, hvornår der skal foretages comit/rollback. Det er en bemærkelsesværdig svaghed i arkitekturen at der ikke er et gennemtænkt og håndhævet transaktionsstyrings-princip, særligt set i lyset af at Business Process Manager-implementeringen er introduceret netop for at håndtere transaktionsstyring.

Opdatering

ScanJour udtaler, at standard Captia i alle tilfælde sørger for at håndtere transaktionsstyringen. Såfremt dette er korrekt i alle tilfælde vil ovenstående afsnit ikke være relevant. Afsnittet er dog af forsigtighedshensyn bibeholdt, da det ikke umiddelbart er muligt at validere udsagnet.

Fejllogning

Logning af systemfejl og uventede fejl sker i POLSAG til Windows eventloggen.

I de klasser hvor der benyttes error logging anvendes to tilgange: Agent-implementeringen anvender den statisk metode på "POLSAG.Agents.AgentLib.LogToEventLog", og øvrige klasser abstraherer funktionaliteten i en lokal metode (en metode per klasse).

Måden der logges til eventloggen og hvad der logges er ikke konsistent for POLSAG som helhed. Der findes således otte implementeringer af metoden "LogToEventLog", der alle sammen udfører den samme handling og flere af metoderne er helt ens defineret (se "Definitioner af LogToEventLog.txt").

Derudover indeholder POLSAG også kode til at udføre skrivning til selve Windows event log mere end 20 gange (se "Forekomst af kald til WriteEntry.txt"), hvilket skyldes at Eventloggen flere steder instantieres og skrives til direkte fra try-catch konstruktioner (se "Forekomst af kald til WriteEntry.txt").

Kode der interagerer med Windows EventLog er således spredt over store del e af POLSAG kodebasen, og selv små ændringer til måden der logges, vil derfor medføre en uforholdsmæssig stor indsats med opdatering og involvere en tilsvarende stor del af POLSAG kodebasen. Implementeringen indikerer at der ikke er etableret nogen arkitekturmæssig tilgang til implementering af errorlogging. Den eneste overordnede styring er, at errorlogging skal ske til Windows Eventloggen, mens den konkrete implementering synes overladt til den almindelige applikationsudvikling. Derfor ses stor forskel i hvordan de enkelte dele af kodebasen implementerer errorlogging.

Mange steder logges ikke stacktrace i forbindelse med errorlogging til Windows Eventlog (dette er f.eks. tilfældet for 94 kald til de otte nævnte definitioner af LogToEventLog). I stedet logges alene beskeden i den exception der blev fanget. Det vil i praksis gøre det svært eller umuligt at benytte det der bliver skrevet til Windows Eventlog til efterfølgende fejlsøgning.

Mange system exceptions indeholder i øvrigt en rent generisk besked, f.eks. "An error occurred" eller "Access Denied" og uden et stacktrace har man derfor ingen mulighed for at identificere hvilket af de 94 kald til LogToEventLog, der resulterede i en besked i eventloggen.

Konklusion: Implementeringen af errorlogning er usammenhængende og sikrer ikke i alle tilfælde et tilstrækkeligt grundlag til at udføre fejlsøgning. Hele tilgangen til logging bør ændres så der indsamles tilstrækkelig information til at fejlsøgning kan ske hurtigt og effektivt. Implementeringen af errorlogning bør samles i et delt bibliotek, der anvendes alle de steder hvor det er nødvendigt at foretage errorlogning, og det bør sikres at der altid logges et stack trace i forbindelse med errrorlogning så det altid er muligt at identificere, hvor fejlen opstod.

Interne integrationspunkter

Business Process Mananger

Business Process Manager-implementeringen består af to ting:

1. En IHttpHandler implementering der via .Net reflektion mapper indkommende kald til en implementering.
2. En model for at foretage opdateringer af flere registre inden for samme transaktion, via funktionsklasser (Fxxxx) og tilhørende business process handlers.

BPM-implementeringen anvender Reflection API til at oversætte et indkommende kald til den tilhørende Business Process Handler, hvilket sker i ExecuteHandler-metoden:

```
1
2

3
4
5


6
7
8
9
10
11
12
13
14
15
16
17
```

POLBP.BusinessProcessManager

Denne implementering er uhensigtsmæssig, da der sker en enumerering af typer i "ExecutingAssembly" (linje 3) for hver request. På en quad core Intel i7 tager den enumerering ca. 600 ms som er en forsinkelse der vil mærkes direkte i klienten. .Net framework cacher resultatet på tråd-niveau, så det er kun første eksekvering på en tråd der har dette overhead, men det vil med et interval svarende til IIS AppPool recycle.. Resultatet af enumereringen og de efterfølgende første to checks i linje 5 er konstant for alle kald og alle kald til BPM-implementeringen passerer denne logik. Så selv om effekten overordnet set er begrænset, er

det hensigtsmæssigt at opdatere denne implementering med en statisk struktur, der kun opdateres ved det første kald til BPM.

Der findes 119 implementeringer af BusinessProcessManager, som omfatter en væsentlig del af kerne forretningslogikken i POLSAG. Heraf findes der unit tests til at teste 5 af disse implementeringer, mens de resterende 114 BusinessProcessManager-implementeringer er ikke omfattet af nogen form for automatiseret test. Det betyder, at regressioner i forbindelse med nyudvikling og vedligeholdelse af den øvrige kodebase kun opdages i forbindelse med manuel test af funktionaliteten og den leverandør-specifikke funktionelle test (denne omfatter ikke den fulde funktionalitet af systemet, og omfatter ikke fejlhåndtering, hvorfor den ikke vil afdække regressioner til disse områder).

Dette stiller meget store krav til omfang og detaljegrad af den manuelle test der skal gennemføres.

Konklusion: BusinessProcessManager implementeringen er stort set ikke dækket af unit tests. Manuel test og den leverandør-specifikke funktionelle test, er således eneste mulighed for at bekræfte virkemåde og opdage fejl, der måtte være introduceret ved tilretning eller nyudvikling. Dette til trods for at BusinessProcessManager-implementeringen dækker over store dele af den centrale forretningslogik i POLSAG.

Den POLSAG-specifikke SOM-model

Den POLSAG-specifikke logiske datamodel består af 35 entiteter, som er sat sammen ovenpå den underliggende database, der består af 856 tabeller (standard Captia har 294 tabeller). Der er 542 registre i POLSAG datadictionary (standard Captia har 199 registre). Herudover er der cirka 280 specialudviklede PL-SQL views og 30 specialudviklede PL-SQL packages.

Anvendelsen af POLSAG modellen

Der er observeret en meget uensartet brug af POLSAGs SOM-model spændende fra standard-Captia mønsteret med 1 skærmbillede til 1 Register over Business Process Manager-håndteringen af tvær-Register opdateringsmønstre og til views og triggers i PL-SQL som dels benyttes af performancehensyn, dels til at vedligeholde synkronisering af indbliks-indstillingerne mellem personkategori- og sagsrecords.

Beslutningen om hvilket anvendelsesmønster, som skal bruges er angiveligt blevet taget på en case-by-case basis, hvilket ikke bidrager til overblikket.

Forespørgsler gennem SOM

POLSAG er bygget oven på Captias SOM-model, som er en datamodel abstraktion oven på en database, som i POLSAGs tilfælde tager form af en Oracle-database. Ud over fordelene ved den indbyggede funktionalitet som indblik, logging og konfiguration giver det udviklerne et typestærkt C# interface til at udføre forespørgsler mod datamodellen.

Bagsiden af medaljen er (som også er en udfordring i andre O/R mapping frameworks) at udviklerne relativt hurtigt mister fornemmelsen for, hvor mange databasekald der bliver foretaget, samt at dette sjældent manifesterer sig som et problem på små datamængder på en udvikler-pc med tilhørende lokal database. Som udvikler skal man hele tiden holde sig for øje at SOM Register-API'et, ud over de oplagte database-forespørgsler i kaldene til ExecuteSearch, også foretager database-forespørgsler hver gang man skaber en instans af en Register-klasse.

Udtrykket nedenfor giver således anledning til et (implicit) ekstra round trip til databasen.

Et eksempel:

Selve metoden er på ca. 500 linjer, hvorfor det ikke er helt ligetil at overskue, hvor mange kald der f.eks. kommer til søgning efter afgørelser.

Funktionen er også et godt eksempel på hvor svært det i det hele taget er at overskue kode, når ikke der er nogen kommenterer til at guide læseren igennem det store mængde forretningslogik, der er til stede i denne funktion.

I en klassisk client/server-arkitektur ville denne metode formentlig være klaret med et enkelt kald til en Stored Procedure i databasen. I POLSAG er noget af denne funktionalitet flyttet til applikationsserveren med flere database roundtrips til følge. Dette eksempel udgør ydermere en udmærket anskueliggørelse af udfordringen ved performanceoptimeringer af POLSAG: Hvis denne funktion bliver kaldt sjældent med små datamængder vil der næppe opstå nogen problemer og således heller ikke være et presserende behov for optimering, men hvis den bliver kaldt ofte på større datamængder vil dette give en alvorlig belastning af webserveren og måske give timeouts til brugeren.

Teknisk vurdering af integration til eksterne systemer

Al web service funktionalitet i forhold til eksterne systemer i POLSAG bygger på ASMX-teknologien. Dette valg er formentlig taget som en logisk følge af at server-side løsningen for nuværende kun kører på .NET 2.0 samt ud fra en vurdering af at dette gør det relativt enkelt at udstille de eksterne systemer til Javascript-klienten via ordinære HTTP-kald (á la SOMASP).

POLSAG-klienten kalder eksterne systemer direkte gennem ESB. Dvs. POLSAG ClientService virker i realiteten udelukkende som en proxy, der viderestiller beskeder fra klienten til eksterne systemer og omvendt.

F.eks. kaldes Index2 servicen 37 steder i POLSAG-klienten. Implementeringen af Index2 ClientService i POLSAG er en ren facade for interfacet på ESB:

```
1
2
3
4
5
6
7
8
```

Index2.ashx.cs

Her mappes det kald klienten ønsker at udføre til en intern metode:

```
1
2
3
4
5
6
7
```

Index2.ashx.cs

I linje 3 kaldes det eksterne system via ESB og i linje 5 sendes resultatet retur til klienten.

Input fra klienten automappes direkte til den type som det eksterne system skal kaldes med via ESB'en:

```
1
2
3
4
5
6
7
8
9
10

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

ClientServiceLib.cs

Der benyttes reflection (linje 5, 10 og 12) til at automappe input parametre fra POLSAG-klienten direkte til den datatype, der skal sendes til det eksterne system via ESB'en. Det betyder, at hvis den datatype ændres, så skal ændringen også implementeres alle de steder i klienten, hvor den benyttes.

Tilsvarende sendes svar fra eksternt system direkte videre til klienten:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

Også dette medfører, at såfremt format/type af data, der modtages fra eksternt system via ESB bliver ændret, så skal ændringen også implementeres alle de steder i klienten, hvor data benyttes.

ClientServices-implementeringen fungerer derfor reelt kun som en simpel proxy for kald fra klienten til og fra ESB. Funktionaliteten i ClientServices er således på nuværende tidspunkt begrænset til at samle/centralisere kald til eksterne systemer, så kaldet ikke sker direkte fra klienten. Der er ingen funktionalitet der afkobler klienten fra ESB (og derigennem de eksterne systemer).

Endvidere gøres opmærksom på, at såfremt der ikke sker nogen transformering/afkobling af data fra eksterne systemer på ESB'en, så har ESB-løsningen ingen effekt ifht. at skærme POLSAG server og klient fra ændringer i eksterne systemer. Dette vil således betyde, at ændringer i eksterne systemer altid vil kræve ændringer til POLSAG server og klient (såvel som ESB).

Der er ikke nogen eksplicit sikkerhedsmodel for web service-grænsesnittene, og der er ikke nogen implementering af checks af data-integritet og konfidentialitet. Der anvendes ikke SSL.

Der er ikke nogen eksplicit model for håndtering af replay detection. Det er op til den enkelte udvikler at sikre mod data-korruption som følge af duplikat-beskeder (ASMX-teknologien har ikke indbygget replay detection i modsætning til WCF[9]), og den nuværende implementering beskytter ikke mod gentagelse af beskeder. Implementeringen benytter http-facilitet til at sætte 'nocache', hvilket alene er en anvisning til en http-klient samt mellemliggende http-caches, og ikke en håndtering af replay detection. Det er fuldt ud teknisk muligt at gensende beskeder, og dette vil ikke blive opdaget eller håndteret af POLSAG.

Det har været nødvendigt at bygge et tool til at generere web service-klienter ud fra WSDL[10]-beskrivelser, fordi middlewarekomponenten (IBM ESB) har interoperabilitets-problemer set fra .NET for så vidt angår overførsel af datoer. ESB'en kan ikke garantere en tabsfri konvertering af UTC-datoformatet fra .NET og der er derfor blevet aftalt et custom data-format til udveksling mellem POLSAG og IBM ESB. Ovennævnte tool håndterer denne konvertering.

Derudover bemærkes det at der er en indbygget begrænsning af længden for brugernavne. Disse kan højst bestå af 8 tegn. Rigspolitiets nuværende AD-brugere har formatet 'wrx12345' hvor 12345 er et

[9] http://en.wikipedia.org/wiki/Windows_Communication_Foundation
[10] http://en.wikipedia.org/wiki/Web_Services_Description_Language

medarbejdernummer. Batchjobs afvikles med en SJSERVICEAGENTUSER, som er længere end 8 tegn. RP har accepteret at denne account kalder videre som SJSERVIC.

Det skal bemærkes at den mainframe-baserede ESB formentlig skal udskiftes i POLSAGs levetid, dels fordi RP ønsker er mere moderne produkt, og dels fordi driftsprisen baseret på MIPS angiveligt vil være overordentlig høj, når den nuværende fastprisordning udløber[11].

Fra et performancesynspunkt har vi ikke kunne finde noget information, som indikerer at de eksterne systemer er vurderet med i POLSAG dvs.

- Forventede svartider fra eksterne systemer (som Index2) er ikke dokumenteret.
- Der er ikke foretaget nogen analyse af POLSAGS forventede forbrug af eksterne systemer. Det tætteste vi er kommet på informationer omkring dette er, at John Thomassen fra CSC ikke regner med at POLSAG vil generere noget synderligt load. Dette er formentligt korrekt, hvis man kan antage at der ikke er en udvikler i POLSAG, der har lavet fejl og kalder de eksterne services for meget.

Det blev bemærket at POLSAG indbefatter en custom implementering af FTP-protokollen (som anvendes fra `public class POLSAG_ANM_AGENT`).

Konklusion: POLSAG har valgt en implementering, som ikke etablerer et fælles kommunikationsmodul mellem POLSAG og ESB'en. Service interfaces på ESB'en reelt eksponeret direkte til Javascript-klienten via et late-bound API, som anvender HTML-formatterede data som input og JSON-formaterede data som output. Hvis der i fremtiden er behov for at ændre kommunikationen med ESB'en vil opdateringerne være vanskeligere end hvis POLSAG havde en fælles kommunikations modul.

I den nuværende løsning sker der ingen transformering af data, der sendes fra klienten til ESB og fra ESB til klienten, hvilket betyder, at en ændring i den måde POLSAG kommunikerer med ESB også kræver en ændringer helt ude i POLSAG-klienten. Hvis der var etableret et kommunikationsmodul i POLSAG, der afkobler kommunikationen med ESB fra resten af POLSAG systemet, så ville sådanne ændringer ikke være nødvendige.

I forhold til data-integritet og konfidentialitet kunne man med fordel anvende SSL transport-sikkerhed for at sikre at data sendt mellem browsere og IIS samt mellem IIS og IBM ESB for at sikre mod at data kan undergå ændringer eller overvågning af uautoriserede personer/programmer. Selvom Rigspolitiets interne netværk vurderes som lukkede og sikre er der jo ingen grund til ikke at beskytte løsningen mod angreb af denne type.

Der er ikke noget i systemet som forhindrer identitets-forvirring overfor backend systemer, når/hvis RP overgår til en navnestandard i Active Directory, som indeholder mere end 8 tegn. Der er en indbygget 'silent failure'-funktionalitet fordi brugernavnet trunkeres, hvilket senere kan risikere at skabe ikke-sporbare logs.

[11] Fortalt af Brian Nielsen RP, under interview

Arkitekturmæssig vurdering af integration til eksterne systemer

Som nævnt ovenfor bliver al kommunikation med eksterne systemer sendt gennem ESB'en, der forestår håndteringen af format- og protokol-formatering.

Der er såvidt vi har forstået ikke udformet noget egentlig integrationsarkitektur eller strategi i POLSAG. Dette har affødt at al kommunikation med eksterne systemer foregår direkte (dog naturligvis via ESB'en) mellem de to systemer (point to point integration) på service grænsesnit, der er specielt udviklet til dette behov. Uanset hvilket systemlandskab man opererer i kan denne type integration ikke anbefales, idet den resulterer i hård kobling mellem systemerne og potentielt n*(n-1) kompleksitet mellem systemerne.

Som eksempel på den hårde kobling mellem systemerne kan nævnes integrationen med bødesystemet. Forretningsbehovet er at bødesystemet skal notificeres når en POLSAG sag skifter kreds. Dette kald til bødesystemet er implementeret i syv funktioner i POLSAG, som alle vil skulle vedligeholdes. Derudover skal alle udviklerne være bekendt med denne forretningsregel, såfremt man implementerer anden funktionalitet i POLSAG, der ændrer kreds på en sag.

Efter vores opfattelse vil det korrekte integrationsmønster baseret på forretningshændelser bestå af følgende flow:

1. Forretningsfunktion i POLSAG ændrer kreds på en sag og rejser forretningshændelse i POLSAG.
2. POLSAG integrationskomponent som abonnerer på den pågældende hændelse sender en forretningshændelse til ESB'en.
3. Bøde som abonnerer på forretningshændelsen på ESB'en modtager hændelsen og behandler den.

Dette integrationsmønster ville sikre afkobling mellem de forskellige komponenter og muliggøre andre abonnenter uden at der opstår behov for ændringer i det eksisterende kode.

Eksterne integrationspunkter

I følgende gennemgås i detaljer en række udvalgte eksterne integrationspunkter i POLSAG.

Central Print

Er implementeret i F0881CentralPrint.cs under ScanJour.CentralPrint. Implementeringen består af en fil på 487 linjer (220 kodelinjer som opgjort af Visual Studio 2010 Code Metrics).

Funktionen CentralPrint er implementeret med henblik på at mange instanser skal kunne arbejde samtidig, af hensyn til skalering. Implementeringen er imidlertid ikke bygget over almindelige design patterns for ordnet styring af flere trådes adgang til delte ressourcer. De steder hvor der kræves en trådsikker implementering benyttes et simpelt try-fail mønster, hvor operationen wrappes i en try-catch kontstruktion og evt. exceptions ignoreres, f.eks.:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
```

....

F0881CentralPrint.cs

Som kommentaren i koden også beskriver, så oprettes i linje 32 en lock-fil og efterfølgende tråde der forsøger at oprette samme log-fil vil få en exception fra filsystemet.

Denne tilgang til trådsikring af koden er problematisk, fordi det betyder at det ikke i applikationslogikken er muligt at skelne mellem reelle fejl fra filsystemet og fejl der i dettetilfælde skyldes at flere tråde forsøger at oprette den samme fil. Derudover medfører det også at flere andre tråde kan forsøge at behandle inputfilen, selv om det er unødvendigt hvis den allerede bliver behandlet af en tråd.

Endviderer betyder anvendelsen af en lock-fil også, at behandlingen af input filer, der er oprettet lock-filer for, ikke genoptages i tilfælde af at en CenralPrint agent stoppes uventet. Når først der er oprettet en lock-fil, er der ikke nogen agenter der efterfølgende kan tage fat i input filen.

Logning i CentralPrint sker ved at der skrives direkte til en tekstfil med .Net indbyggede metoder til at tilgå en fil:

```
1
2
3
4
5

6

7
8
9
10
11
12
13
```

F0881CentralPrint.cs

Dette er ikke hensigtsmæsigt, fordi ændrerede behov for logning medfører at der skal laves relativt set store ændringer, og deraf følgende omkostninger til udrulning mv. Der findes mange robuste frameworks til håndtering af logging, der muliggør omkonfigurering af logging (f.eks. til database, logserver, mv.) uden at der kræves ændringer til kildekoden. Dette ville også gøre det muligt at konsolidere logdata fra flere agent-instanser i en enkelt fil.

Endeligt er implementeringen af CentralPrint kendetegnet ved en række af de problemer der generelt ses i POLSAG kodebasen:

- Mange fail-fast[12] violations som følge af meget begrænset parametervalidering. Input og return parametre valideres således ikke.
- Begrænset eller ingen fejlhåndtering. Metoder bruger "blanket wrapping" til at fange exceptions, der blot logges uden at der sker yderligere håndtering. Fejlhåndtering er derfor afhængig af at der regelmæssigt kigges i loggen.

[12] http://en.wikipedia.org/wiki/Fail_Fast

- Mange magic strings overalt i koden, hvilket gør det unødigt komplekst at vedligeholde, og forøger risikoen for fejl ved senere ændringer. Derudover medfører simple tastefejl, at logikken ikke fungerer. Disse "magic strings" benyttes til at referere feltnavne, sammenligne med forretningsdata etc.
- Der findes ikke nogen unit tests til automatiseret og løbende aftestning. Der er defineret en testmetode hørende til klassen DocumentScan, men test-metoden indeholder ikke nogen Assert-kald og tester kun happy-path scenariet for funktionaliteten i DocumenScan og kun med et statisk datasæt.

Konklusion: Tilgangen til trådsikring i CentralPrint er ikke effektiv og der er stor risiko for implementeringen ikke fungerer som forventet. Endvidere anvendes der mange problematiske teknikker i CentralPrint (blanket wrapping[13], magic strings, mv.) der gør implementeringen svær at fejlfinde og vedligeholde. Endeligt er der meget begrænset unit test dækning af løsningen, så al aftestning skal ske manuelt.

Sagsreoler

Sagsreoler er implementeret som en Silverlight-klientapplikation, der kommunikerer med POLSAG via en dedikeret WCF service.

Problemet med store og komplekse metoder ses også i implementeringen af WCF servicen til sagsreoler. Metoderne i implementeringen er lange og med mange nestede niveauer, hvilket gør koden svær at læse og vedligeholde.

Der er mange metoder på webservicen, der er navngivet så de starter med et lille bogstav (f.eks. getReolInfo). Dette er ikke som sådan en fejl, men det gør kildekoden unødig svær at læse, da modtageren forventer at metodenavne starter med et stort bogstav. Endvidere er det imod almindelige god praksis for udvikling af C#-kode, da standarden her er, at alle metodenavne starter med et stort bogstav.

Der findes ikke nogen automatiserede tests (unittests) af koden til sagsreol servicen, så den eneste måde at sikre funktionaliteten, er ved manuel aftestning.

Sagsreol-klienten er opbygget efter et klassisk code behind-mønster, hvor al logik placeres i såkaldte code behind-filer. Dette er uhensigtsmæssigt, da dette designmønster medfører meget store og komplekse code behind-filer. Best practice for implementering af grafiske brugergrænseflader i WPF/Silverlight er at benytte MVVM-arkitekturen (eller eventuelt den mere klassiske MVC-arkitektur).

Konklusion: Implementeringen af sagsreoler er baseret på et forældet mønster til udvikling af brugergrænseflader. Erfaringer viser at løsninger, der er baseret på store code behind-filer, ender med at blive meget uoverskuelige og svære at vedligeholde.

Scanning

Scanning er implementeret via DocumentScan.DocumentScan i solution POLServices. Implementeringen består af en fil på 185 linjer (70 kodelinjer som opgjort af Visual Studio 2010 Code Metrics).

[13] Indkapsling af kode i en try-blok, hvor der i den efterfølgende catch-blok ikke foretages nogen handling. Dette fungerer som en "silent ignore" på alle exceptions i try-blokken, og medfører at vigtige fejl ikke opdages.

Implementeringen af DocumentScan er kendetegnet ved en række af de problemer der generelt ses i POLSAG kodebasen:

- Mange fail-fast violations som følge af meget begrænset parametervalidering. Input og return parametre valideres ikke.
- Begrænset eller ingen fejlhåndtering. Metoder bruger "blanket wrapping" til at fange exceptions, der blot logges uden at der sker yderligere håndtering. Fejlhåndtering er derfor afhængig af at der regelmæssigt kigges i loggen.
- Mange magic strings overalt i koden, hvilket gør det unødigt komplekst at vedligeholde, og forøger risikoen for fejl ved senere ændringer. Derudover medfører simple tastefejl, at logikken ikke fungerer.
- Der findes ikke nogen unit tests til automatiseret og løbende aftestning. Der er defineret en testmetode hørende til klassen DocumentScan, men test-metoden indeholder ikke nogen Assert-kald og tester kun happy-path scenariet for funktionaliteten i DocumenScan og kun med et statisk datasæt.
-

Office integration

Der er udabejdet et antal Word-skabeloner, der installeres på alle klienter og aktiveres fra klienten via Javascript kald til en standard Captia komponent ("sjDocIntegration"). Integrationen med POLSAG er implementeret via VSTO[14]-komponenter, der kan inkluderes i skabelonerne.Skabelonerne henter data fra Captia via SOMASP analogt til Javascript-klienten. Der er ikke nogen overordnet strategi, som sikrer at man får minimeret antallet af kald til Captia fra disse skabeloner, da hver enkelt komponent er selvstændig og foretager et servicekald. Der er dog udviklet en speciel skabelon til døgnrapporten af denne årsag.

Der er p.t. dialog omkring overlevering af opgaven med udvikling af skabeloner fra POLSAG-leverandøren til Rigspolitiet. Der er dog ikke taget nogen endelig beslutning herom endnu.

Al aftestning af skabelonerne er manuel.

Office-skabeloner

Office skabeloner er implementeret som en særskilt solution, der kun omfatter Office-skabeloner. Implementeringen består af en objekt model og logik til at hente data hhv. opdatere data i POLSAG via SOMASP interfacet.

Implementeringen af Office-skabeloner omfatter ca. 8.000 kodelinjer (som rapporteret af Visual Studio 2010). Kvaliteten af kildekoden i implementeringen vurderes at være bedre, såvel kvalitativt og kvantitativt, end resten af POLSAG-kodebasen. Der er 28 metoder (ud af 1506) med en cyklomatisk kompleksitet over 15[15], og heraf 13 metoder med en cyklomatisk kompleksitet over 20. Tallene omfatter ikke test-kode.

Derudover er der implementeret en objektmodel og arkitektur der retter sig mod det problemdomæne, som løsningen arbejder med.

---

[14] http://en.wikipedia.org/wiki/Visual_Studio_Tools_for_Office

[15] En cyklomatisk kompleksitet på 10-15 giver den laveste fejlrate. Se afsnittet Cyklomatisk Kompleksitet.

Der er observeret følgende væsentlige problemer i implementeringen af Office-skabeloner:

- Skabelonerne kan kalde POLSAG rigtig mange gange. Der sker kald til POLSAG hver gang en skabelon gemmes.
- Der benyttes mange "magic strings" til at binde forskellige begrebs-modeller sammen internt i skabelonerne.
- Der er ingen automatiseret test, der sikrer at ændringer et sted ikke introducerer fejl andre steder.
- Transaktionsstyring i Office-skabeloner forudsætter en bestemt opbygning af skabelonerne uden at kontrollere det, hvilket betyder, at en ændret opbygning kan medføre at transaktionsstyringen ikke er tilstrækkelig.

Office-skabeloner er opbygget så data i en skabelon gemmes i POLSAG samtidig med at skabelonen gemmes af brugeren. Dette gøres ved at lytte på eventen "Globals.ThisAddIn.Application.DocumentBeforeSave" og så opdatere data i POLSAG i eventhandleren. Der sker flg.:

1. "PolsagVSTOAddIn.EventHandlers.BeforeSave" er bundet op på DocumentBeforeSave eventen.
2. BeforeSave metoden kalder "PolsagVSTOAddIn.Cls.Utils .SjContentContrel.SaveBackToPolsag".
3. SaveBackToPolsag kalder "PolsagVSTOAddIn.Cls.Utils .SjServer.Update" der opdaterer data i POLSAG via SOMASP.

Måden hvorpå data opdateres i POLSAG sikrer ikke transaktionsstyring, hvilket medfører at der kan opstå inkonsistens i data i POLSAG i forbindelse med opdatering. Den væsentligste del af problemet ligger i denne metode:

```
1
2
3

4
5
6
7
8
9
10
11
12

13
14
15
16

17
18
19
```

SjContentControl.cs

Implementeringen gennemløber alle content controls og indholdet af hver content control bliver gemt tilbage i POLSAG. Hvis der opstår en fejl under dette gennemløb, f.eks. hvis klienten mister netværksforbindelse, så bliver de første opdateringer ikke rullet tlibage, hvilket vil efterlade data i POLSAG i en inkonsistent tilstand.

Denne implementering kan også medføre at der genereres mange kald mod POLSAG. Der genereres et kald for hver content control der skal opdateres, og disse kald genereres hver gang brugeren gemmer skabelonen. Med den nuværende opbygning af Office-skabelonerner genereres kun et kald ved opdatering.

Implementeringen af Office-skabeloner benytter tekstværdier til at koble skabelonfelter med felter i Captia. Der findes ikke noget samlet schema med definitioner og der er ikke implementeret nogen type-model til at referere feltnavnene, hvorfor udviklerne er nødt til at skrive alle feltnavne direkte ind i kildekoden som tekster. Denne tilgang giver følgende problemer:

- Det bliver unødigt besværligt at omdøbe et felt.
- Udviklerne får ingen tooling support, hvilket medfører stor risiko for at feltnavne bliver stavet forkert.
- Der er ingen compile-time tjeks for forkerte feltnavne. Den eneste måde sådanne fejl kan opdages, er ved at teste skabelonerne.
- Det gøres unødigt besværligt for nye udviklere at lære feltdefinitioner. Med en type-stærk løsning kan nye udviklere via almindelig auto-complete funktion "opdage" hvilke feltnavne der findes, hvilket er en meget stor hjælp.

Som en absolut minimumsløsning, bør alle faste feltnavne samles i et eller flere schema objekter der udstiller felterne som "public const string"-typer, da man derved vil kunne undgå man alle ovenstående problemer.

Flere steder benyttes store switch-blokke med mange case-muligheder. Dette gør koden væsentligt mindre overskuelig og sammenblander forskellige formål i den samme metode, hvilket er et uhensigtsmæssigt design. F.eks. indeholder metoden

"PolsagVSTOAddIn. PolsagDocument.ActiveDocument_ContentControlOnExit_forActionPane" en switch-blok på 226 linjer med 17 cases. Mange cases i denne metode anvendes tilsyneladende ikke og kan udelades, og een case indeholder en nested switch-blok med yderligere fire cases. Overskuelighed, læsbarhed og mulighed for vedligehold kunne forbedres betydeligt, hvis der i stedet for switch-blokken, blev benyttet en lazy-load teknik til at indlæse en handler til den pågældende case.

Endeligt er der ikke nogen unit tests eller anden form for automatiseret tests, der kan afvikles løbende, for at verificere funktionalitet og virkemåde af de enkelte dele i Office-skabelonerne såvel som helheden. Manglende tests medfører en stor risiko for at ændringer og udvidelser introducerer fejl andre steder i løsningen.

Office-skabelonerne indeholder ikke nogen form for trace logging, der kan aktiveres i forbindelse med fejlsøgning, hvilket gør det meget svært at bestemme hvor fejlen stammer fra, idet en fejl i en Office-skabelon kan skyldes en fejl i skabelonen, en fejl sendt fra POLSAG til skabelonen eller en ændring i POLSAG, der ikke er implementeret i Office-skabelonerne.

Konlusion: Kvaliteten i kodebasen til Office-skabeloner vurderes at være acceptabel. Måden som Office skabeloner opdaterer POLSAG sikrer ikke transaktionsstyring, og fejl der opstår mens en skabelon opdaterer POLSAG kan medføre at data i POLSAG bliver inkonsistente. Der er ingen automatiserede tests, der sikrer mod regressioner ifbm nyudvikling og vedligehold, ligesom der ej heller er indbygget trace logging.

Døgnrapporten

Døgnrapporten er implementeret som en særskilt skabelon, der indeholder logik til at hente data fra Captia via SOMASP interfacet.

Hele døgnrapporten er implementeret som ren procedural kode uden nogen form for objektorienteret arkitektur.

Implementeringen omfatter 1.696 kodelinjer (som rapporteret af Visual Studio 2010 Code Metrics), hvoraf de 1.385 linjer er applikationslogik. Den resterende del er koden er UI og tilhørende autogeneret kode.

Hele applikationslogikken er fordelt på kun 3 klasser (DrRapport, ThisDocument og Doegnrapport), og heraf udgør een klasse (ThisDocument) alene 1140 linjer eller ca. 82% af applikationslogikken. I denne klasse udgør een metode (ThisDocument.byggIdenterIndhold) 415 kodelinjer eller ca. 30% af den samlede applikationslogik. Den cyklomatiske kompleksitet for denne metode er 164 (bør ligge på 10-15) og maintainability indeks 0, hvilket er dokumenteret i bilaget "Code Metrics – DrRapport.xlsx".

Det skal her bemærkes, at antal kodelinjer som rapporteret af Visual Studio 2010 er væsentligt lavere end det antal der rent faktisk vises på skærmen, hvilket skyldes at Visual Studio normaliserer formateringen af koden i forbindelse med generering af statistik. Således fylder føomtalte metode ca. 1.000 linjer, når den vises i Visual Studio (hele applikationslogikken udgør knap 3.000 linjer).

Der findes ikke nogen unit tests til Døgnrapport-implementeringen. Strukturen for kodebasen til døgnrapport-implementeringen medfører at det kun er muligt at lave overordnede tests af koden, og automatiseret unit test af delelementer i logikken er i praksis ikke muligt.

Strukturen i døgnrapporten medfører samtidig at:

- Det er nødvendigt at der er installeret Microsoft Office på den computer, hvor døgnrapporten skal udvikles/vedligeholdes.
- Det er nødvendigt, at der er adgang til en kørende installation af POLSAG for at kunne vedligeholde døgnrapport-implementeringen.

Der eksisterer en meget hård kobling mellem døgnrapport og Microsoft Word henholdsvis POLSAG, der medfører at begge systemer skal være tilgængelige for at kunne vedligeholde døgnrapport-implementeringen.

Selve implementeringen af døgnrapporten strider imod en række almindelige principper for softwareudvikling (specielt single responsibility principle og dependency inversion principle, se afsnit om kodekvalitet) og de hårde bindinger medfører at systemet bliver meget lidt robust overfor fejl. Almindelig praksis for softwareudvikling er, at delsystemer adskilles, så det er muligt at arbejde på een del uden at være afhængig af de øvrige dele. I stedet for at kræve Microsoft Office og POLSAG systemerne er

tilgængelige bør det være muligt at benytte tilsvarende "stubs" ifbm. vedligehold af døgnrapporten. Det har samtidig den anden positive effekt at f.eks. kørselstiden på unit tests ikke bliver afhængig af svartiden for POLSAG.

Døgnrapport-implementeringen vurderes at give utilstrækkelig information til brugeren ved fejl. F.eks.:

```
1
2
3
4
5
6
7
8
9
10
11
```

ThisDocument.cs

Bemærk også at fejlen heller ikke logges til senere fejlfinding.

Linje 11 er en blanding af dansk og engelsk, og indeholder ikke nogen beskrivelse af, hvad den eventuelle fejl måtte skyldes eller hvad konsekvensen er for brugeren. Døgnrapporten anvendes direkte af slutbrugere, og det er derfor meget vigtigt at brugerne ved hvordan de skal forholde sig i tilfælde af fejl.

Endeligt medfører den måde som døgnrapporten kalder POLSAG, at der potentielt kan genereres et meget stort antal kald til POLSAG, når døgnrapporten dannes. Risikoen består i den måde døgnrapporten kalder SOM API'et (f.eks. ThisDocument.hentDataFraTemplate), som sker i et loop henover de koder der skal hentes data for, hvor der kun hentes data for 5 gerningskoder per kald. Testdata indlejret i kildekoden til døgnrapporten indeholder op til 700 gerningskoder hvilket genererer 28 kald mod POLSAG.

Der er i forbindelse med performance review'et observeret væsentlige performanceproblemer med Døgnrapport-implementeringen, hvor Døgnrapporten opleves som ustabil, hvilket primært er kommet til udtryk ved en eller flere af følgende problemer:

- Døgnrapporten er meget lang tid om at indlæse.
- Der kommer fejl ved indlæsning, når POLSAG systemet er under load.
- Døgnrapporten lukker uventet under indlæsning.

En gennemgang af den logik i døgnrapport implementeringen, der opbygger selve Word dokumentet afdækker flere væsentlige potentielle performanceproblemer i koden. F.eks.:

```
1
2

3
4
5
6
7
```

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

ThisDocument.cs

I linje 10 sættes array Pk_taken til en størrelse på 9999 elementer, og den efterfølgende kode bevirker at der altid itereres gennem den fulde længde af det array i linje 25.

Endvidere er iterationen i linje 25 tredje niveau i en triple-nested for-konstruktion (dette er i sig selv meget uhensigtsmæssigt), hvilket medfører at dannelsen af en døgnrapport meget let kan udarte sig til mange tusinde iterationer af denne triple-nestede for-konstruktion. Dette medfører, at det tager lang tid at opbygge døgnrapporten, og brugeren således vil opleve at det tager meget lang tid at indlæse den.

Derudover ses eksempler på at når POLSAG systemet er belastet, så fejler døgnrapporten med fejl som flg.:

Fejl som denne er kun observeret ved afvikling af døgnrapporten, når POLSAG systemet er under belastning. Afvikles den samme døgnrapport når systemet ikke er belastet opleves denne type fejl ikke.

Dette indikerer at der er en risiko for at der eksisterer fejl i det underliggende Captia system, som kun kommer til udtryk når systemet belastes. Da almindelig udviklingsarbejde og manuel test normalt ikke udføres mens systemet belastes, er der risiko for at fejl som denne kun opdages på et produktionssystem

Der er også implementeret en Excel skabelon (DRExcelSkabelon) i relation til Døgnrapporten. Kildekoden til denne skabelon blev ikke udleveret sammen med den øvrige kildekode, hvorfor denne skabelon ikke er blevet gennemgået.

Overordnede konklusioner omkring Office-skabeloner

Konklusion: Der er risiko for at skabelonerne vil trække mange server-side ressourcer, helt analogt til Javascript-klienten, fordi der ikke er en fælles model for hvordan man får minimeret kommunikations-overheadet.

Der bør være særligt fokus på dette område, hvis planen om at Rigspolitiet selv skal stå for den fremadrettede udvikling af disse skabeloner bliver virkeliggjort. Dokumentskabeloner bør udvikles under skyldig hensyntagen til hvor mange server-side ressourcer de lægger beslag på, hvilket kan være svært at overskue selv for udviklerne af skabelon-komponenterne.

Der bør indbygges automatiserede tests af de skabelonerne. Set i lyset af at disse muligvis skal udvikles af Rigspolitiet bør der bygges et test framework som også Rigspolitiet fremadrettet kan anvende til deres egenudviklede skabeloner.

Døgnrapporten bør omskrives efter almindelige principper for softwareudvikling, så det sikres at den kan testes og vedligeholdes. Med dens nuværende struktur er det fejlsøgning meget kompliceret og test af døgnrapporten kræver et fuldt kørende POLSAG system, hvilket i praksis binder denne opgave til udvikling og vedligehold af POLSAG systemet.

Endvidere bør der i udviklingsarbejdet indarbejdes en struktureret tilgang til aftestning af funktionalitet og performance af døgnrapport-implementeringen, så det sikres at den virker som forventet med realistiske datasæt. Test af performance bør som minimum køres på et realistisk datasæt fra f.eks. København.

Kodekvalitet

Der er blevet foretaget en gennemgang og vurdering af kodebasen, hvor det primære fokus har ligget vedligeholdbarhedsegenskaberne og hvordan POLSAG vil være at udvide og ændre i tiden fremover. De anvendte kriterier for vurderingen har været

- Læsbarhed
- Overskuelighed
- Anvendelse af objekt-orienterede best practices (undtagen for PL-SQL koden)
- Automatiseret test coverage

Der har for de tre objekt-orienterede programmeringsplatforme i høj grad været fokuseret på anvendelsen af best practice-principperne:

S.O.L.I.D [SOLID acronym: http://en.wikipedia.org/wiki/Solid_(object-oriented_design)]

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

DRY [DRY acronym: http://en.wikipedia.org/wiki/Don%27t_repeat_yourself]

- Don't Repeat Yourself

POLA [POLA acronym: http://en.wikipedia.org/wiki/Principle_of_least_astonishment]

- Principle Of Least Astonishment

C#

Gennemgangen af kodekvalitet for C#-kode har taget udgangspunkt i solutions til POLServices, Døgnrapporten, PolsagWordSkabelonAddIn og Sagsreoler klienten. Der er anvendt C# kode i andre solutions, men de solutions har ikke været omfattet af gennemgangen. Udenfor POLServices solution er mængden af C#-kode er meget begrænset og anvendelsen af C# er ofte enten sekundær eller også har solution ikke nogen indvirkning på kernefunktionaliteten (som f.eks. når det bliver benyttet til et installationsprogram).

Gennemgangen af C# koden har givet anledning til følgende:

- Kildekoden indeholder mange uhensigtsmæssigheder der kunne være forebygget ved pro-aktivt at anvende værktøjer til statisk kodeanalyse i forbindelse med udviklingsprocessen.
- Kildekoden bærer præg af manglende kvalitetssikring og afslutning.
- Der anvendes mange "magic strings" for at binde systemet sammen
- Kildekoden er af meget lav kvalitet som målt af gængse målemetoder og værktøjer til statisk kodeanalyse

- Input validering anvendes kun i begrænset omfang

Nøgletal

Vi forstår fra vores interviews at der ikke er anvendt CodeAnalysis eller periodiske statiske analyser af kodebasen i udviklingsforløbet.

FxCop 1.35 rapporterer knap 5000 issues. Nogle af disse skyldes at FxCop ikke kan detektere late-binding kald. Men der er stadig mange konkrete anbefalinger fra FxCop, som udviklere vil kunne drage stor nytte af (se Code Analysis-rapport for C#-delen af POLSAG. Se det vedlagte dokument POLServicesCodeAnalysis.FxCop.]).

Der er fx tale om mange violations af navngivningsstandarder og almindelige policies for robust API-design, og temmelig mange deciderede fejl (bl.a. general-exceptions-catching og string-operations-problemer, som ikke håndterer Culture settings).

FxCop har fundet mere end 200 kritiske fejl, og i alt knap 3.000 fejl. Der er 29 kritiske warnings.

Den autogenererede DataAccess-kode er ikke omfattet af FxCop-kørslen.

FxCop detekterer følgende kritiske violations:

- DoNotCatchGeneralExceptionTypes – 130 violations (http://msdn2.microsoft.com/library/ms182137(VS.90).aspx)
- DoNotRaiseReservedExceptionTypes – 30 violations (http://msdn2.microsoft.com/library/ms182338(VS.90).aspx)
- CallGCSuppressFinalizeCorrectly – 2 violations (http://msdn2.microsoft.com/library/ms182269(VS.90).aspx)
- TypesThatOwnDisposableFieldsShouldBeDisposable – 1 violation (http://msdn2.microsoft.com/library/ms182172(VS.90).aspx)
- OperationsShouldNotOverflow – 1 violation (http://msdn2.microsoft.com/library/ms182354(VS.90).aspx)
- SpecifyCultureInfo, SpecifyIFormatProvider, SpecifyCultureInfo – ialt 483 violations (http://msdn2.microsoft.com/library/ms182189(VS.90).aspx, http://msdn2.microsoft.com/library/ms182190(VS.90).aspx, http://msdn2.microsoft.com/library/bb386080(VS.90).aspx)

FxCop fandt følgende warnings, som indikerer manglende oprydning af koden:

- RemoveUnusedLocals – 81 warnings (http://msdn2.microsoft.com/library/ms182278(VS.90).aspx)
- ReviewUnusedParameters – 89 warnings (http://msdn2.microsoft.com/library/ms182268(VS.90).aspx)
- AvoidUnusedPrivateFields – 211 warnings (http://msdn2.microsoft.com/library/ms245042(VS.90).aspx)
- AvoidUncalledPrivateCode – 18 warnings (http://msdn2.microsoft.com/library/ms182264(VS.90).aspx)

Manglende afslutninger

Ved første gennemgang af kildekoden i juni måned var der 148 TODO's. Ved gennemgang af kildekoden som udleveret 18-11-2011 var der fortsat 134 TODO's i kildekoden (se dokumentet "Forekomst af TODO i koden.txt").

Ligeledes blev der ved første gennemgang fundet 12 steder, hvor der kastes en ***NotImplementedException*** aktivt fra levende kode. Ved gennemgang af kildekoden udleveret 18-11-2011 er der 14 steder, hvor der kastes en ***NotImplementedException*** *(se dokumentet "Forekomst af NotImplementedException i koden.txt")*.

Mange TODO's ser ud til ikke at være blevet fulgt op. Det er ikke klart om det skyldes manglende oprydning af TODO's eller manglende implementering. Det må på den baggrund konkluderes at der ikke findes nogen udviklingsprocesser, der sikrer en struktureret tilgang til håndtering af de TODO, der skrives ind i kildekoden.

Tilsvarende foretages der 31 kald til System.Diagnostics.Debug fra C#-koden i POLServices solution. Heraf sker 16 kald fra kode, der afvikles runtime af systemet i produktion (se dokumentet "Forekomst af kald til System.Diagnostics.Debug.docx" for en nedbrydning). Disse kald påvirker ikke virkemåde og har kun begrænset indvirkning på performance, men de indikerer at udviklingen er afsluttet inden der er sket tilstrækkelig produktionsmodning af kildekoden. Kaldene er typisk enten blevet anvendt i forbindelse med udvikling og skal slettes, eller bør foretages via en af standardfaciliteterne til tracing og logging i POLSAG.

Der findes mange udkommenterede kodelinjer fordelt over hele kodebasen. Den udkommenterede kodebase tager dels form af hele metoder der er udkommenteret dels af 1-5 linjer inde midt i lange metoder, f.eks.:

- Funktionen "HentAfgorelseOplysninger" i F0170
- Funktionen "GemAfgorelse" i F0115
- Funktionen "BeregnEftersogning" i F0125
- Funktionen "ValidateMandatory" i F0114BusinessProcessHandler
- Funktionen "F0888" i "I0256.js"
- Funktionen "singleSelectF0306" i "CustomCaptia.js"

Der er ikke nogen kommentar eller tilsvarende i forbindelse med den udkommenterede kode, hvilket betyder at det ikke er muligt at vurdere formålet med koden og evt hvorfor den er udkommenteret. Når en ny udvikler overtager udvikling og vedligehold (eller den samme udvikler kommer tilbage til kodelinjerne meget senere) er personen derfor nødt til at bruge tid på at gå tilbage i versionshistorikken og se hvornår koden blev udkommenteret for at forsøge at udlede, hvorfor koden er udkommenteret og evt. hvorfor den ikke blev slettet.

Der ses også eksempler på at fejlhåndtering mangler, og at der i stedet blot er indsat en kommentar herom:

```
1
2

3
4
```

F0038FrakendelseBusinessProcessHandler.cs

I eksemplet testes i linje 3 hhv. 8 for specifikke kriterier, men der sker en "silent ignore" i tilfælde af at kriterierne ikke er oplyst. I stedet er der blot indsat kommentaren "ups".

Ovenstående kodelinjer giver ingen muligheder for at afgøre, hvad problemet består i og hvordan denne tilstand bør udbedres. Ud fra det eksplicit tjek på parametrene i linje 3 hhv. 8 kan det konkluderes, at der er situationer, hvor tjekket ikke er opfyldt (ellers er de to tjeks unødvendige) og tilsvarende at fejlhåndteriengen mangler.

Manglende fejlhåndtering er i sig selv meget problematisk ifht. vedligehold af systemer og fejlfinding i tilfælde af problemer. Og uden kommentarer eller kode til illustration bliver det yderst svært gennemskueligt for dem der skal vedligeholde koden hvad formålet med valideringen er.

Den manglende fejlhåndtering betyder endvidere, at der ikke ses nogen tegn på at noget er fejlet. I værste fald medfører den manglende fejlhåndtering, at eksekveringen fortsætter og først fejler et stykke tid senere, hvilket vil starte fejlsøgningen et helt forkert sted relativt til, hvor problemet opstod.

Statiske tekster

Det bemærkes at selvom applikationen i meget høj grad anvender late binding og magiske strengværdier til at binde det hele sammen er der næsten ingen centraliserede definitioner af disse magiske strenges værdier. Der optræder mange hardkodede kopier af disse, og da udviklingsværktøjet ikke kan tjekke stavemåden, er der stor risiko for at der sker tastefejl eller stavefejl, hvilket ikke kan opdages i forbindelse med kompilering.

Derudover er der omfattende brug af hardkodede værdier i den logik som er implementeret og det vil dermed være ganske risikabelt at ændre værdier i f.eks. stamdata, da der ikke er nogen måde at sikre sammenhæng mellem stamdata og kildekoden.

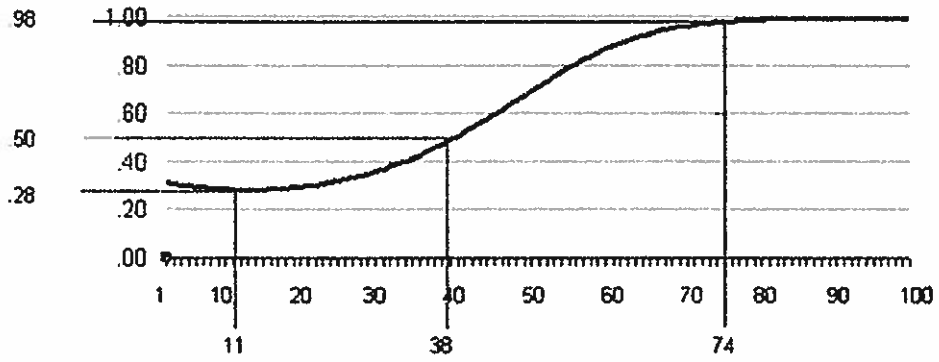
Cyklomatisk Kompleksitet

Den statiske analyse af C#-kodebasen angiver følgende nøgletal: Der er cirka 45.000 linjers kode (auto-genereret Register-model ovenpå datadictionary er ikke medtaget) og cirka 6.300 metoder. Heraf er knap halvdelen (ca 21.500 linjer) koncentreret i 313 metoder.

Med andre ord ligger knap 50% af funktionaliteten samlet i 5% af kodebasen.

Mere end 14.000 (ca. 35%) kodelinjer ligger i metoder som har en cyklomatisk kompleksitet på 15 eller derover. Heraf har halvdelen en cyklomatisk kompleksitet på 30 eller derover. Den højeste fundne cyklomatiske kompleksitet for en metode er 328, hvilket i praksis betyder at man, for at danne sig et overblik over denne metode, skal kunne overskue 328 samtidige mulige uafhængige eksekverings-veje gennem metoden).

For at reducere antallet af fejl mest muligt, bør den cyklomatisk kompleksitet ligge på 10-15, hvilket giver den største sandsynlighed for at begrænse fejlraten. En struktureret gennemgang af kode og fejlhistorik viser, at risikoen for fejl ved en cyklomatisk kompleksitet på 11 kan ligge på 0,28 og den stiger til 0,98 ved en cyklomatisk kompleksitet på 74.



Kilde: http://en.wikipedia.org/wiki/Cyclomatic_complexity#cite_note-enerjy-10

Inputvalidering

Der forekommer i størrelsesordenen 50 eksplicitte checks for null-argumenter, men der er ingen konsekvent gardering mod null-inputs i public metoder for at undgå 'Object reference not set to an instance of an object' runtime exceptions. Eksempler:

- K0033.ValidateControl
- K0432.Validate
- K5042.ValidateControl
- F0589.HentPOLSASSag

Ligeledes ses der heller ikke en konsekvent strategi for håndtering af null-returværdier fra metoder. Eksempelvis vil følgende konstruktion (fra F0589.HentPOLSASSag):

være en kandidat til at terminere webserver-processen, såfremt kommunikationen med det eksterne POLSAS system fejler, eftersom den kaldte `POLSAS.POLSAS.SagHent` er implementeret som følger:

Et andet eksempel på manglende check af null retur-værdi fra metode ses i det vedlagte dokument F0192.cs, metoden `public string findKredsnavn(string input)`.
Det bemærkes at der er lavet en custom implementering til JSON-formattering af svar fra ClientServices til Javascript-klienten. Denne implementering er ikke unit testet, og fungerer via late-binding API'et til .NET Type-systemet. Kommentarerne i implementeringen indikerer at den ikke er helt gennemarbejdet endnu (se 'Custom JSON formatter').

Javascript

Der er gjort følgende observationer i forbindelse med gennemgangen af Javascript-koden:

- Vi forstår fra vores interviews at nogle udviklere benytter JSLint, men at anvendelsen ikke er generel eller konsekvent. Der anvendes ikke andre værktøjer til at sikre kodekvaliteten.
- Javascript-koden overholder ikke almindelig best practice for udvikling i Javascript.
- Der anvendes primært procedural programmeringsparadigme, og der er mange eksempler på meget lange funktioner.
- Kodebasen er ikke tilstrækkelig vedligeholdt. Dvs. der eksisterer en stor mængde kode der enten ikke anvendes eller er udkommenteret.
- Implementeringen er tilsyneladende ikke løbende blevet opdateret med den store nye viden om Javascript og anvendelsen af samme i webapplikationer, der er opstår henover de seneste 4 år.
- Logikken er ikke jævnt fordelt blandt kodefiler. En stor mængde kode er afgrænset til få filer.
- Der er mange funktioner, der udfører stort set det samme arbejde.
- Der er ingen automatiseret test til at sikre mod introduktion af fejl i forbindelse med udvikling og vedligehold.
- Opdateringer fra Javascript sikrer ikke altid effektiv transaktionsstyring. Der er 58 anvendelser af "UpdateFromUrl" end pointet og flere tilfælde, hvor det kaldes to gange fra samme metode.

Der bliver ikke systematisk anvendt værktøjer til at lave statiske analyser af Javascript-kode. Nogle udviklere på POLSAG anvender dog JSLint til at fange de mest almindelige fejl, men anvendelsen er ikke struktureret eller automatiseret.

JSLint er defacto standard for statisk analyse af Javascript-kode og en gennemgang med JSLint af alle Javascript filer i løsningen viser mere end 1.000 afvigelser fra JSLint best practices. Hovedparten er advarsler omkring syntaks, struktur og formaterring, der gør koden unødvendig svær at vedligeholde, f.eks.:

- Property access med "square bracket" notation, hvor der bør anvendes dot-notation.
- Variable der er deklareret, men ikke anvendt.
- Lange linjer der er sværere at læse.
- Ikke-konsekvent anvendelse af source formattering.

Derudover ses mange advarsler omkring implementering, der har uforudsigelig opførsel:

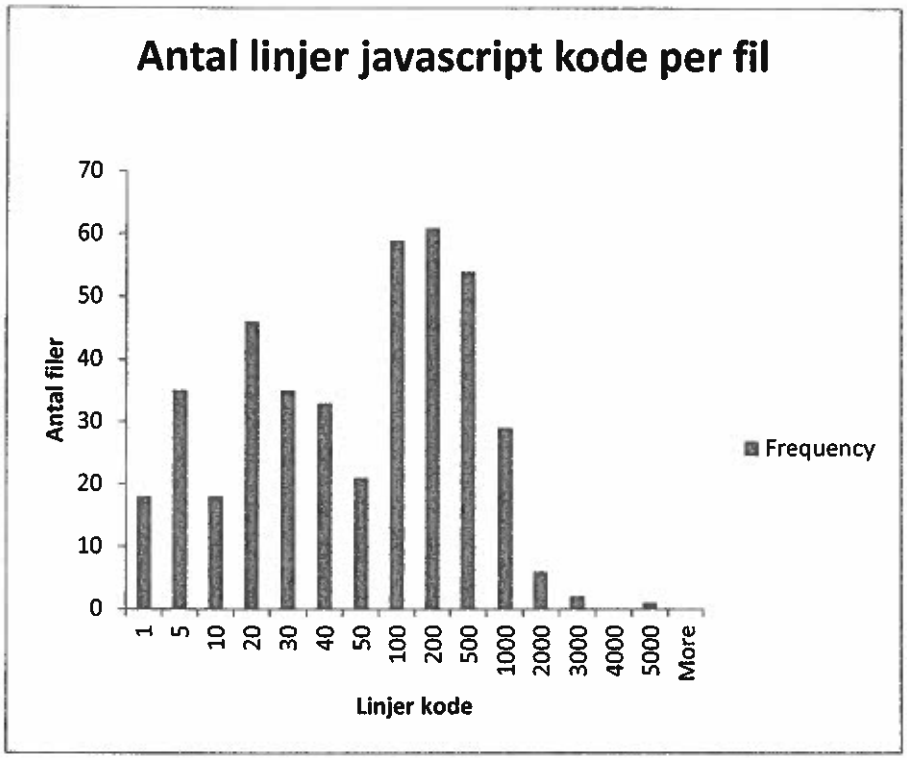
- Der anvendes konsekvent ikke-transitive varianter af ligheds-komporatorer, f.eks. benyttes '==' konsekvent i stedet for '==='.
- Der erklæres implicitte globale variable. Givet der normalt *ikke* anvendes implicitte globale variable i koden tilskriver vi indtil videre dette til kodefejl, hvor man har glemt at skrive 'var' foran deklarationen. Globale variable vurderes normalt som den værste facilitet i Javascript, der bør undgåes, hvor det er muligt. Se bl.a. akt_reg_user.js linje 175, ClassPakke.js linje 114 og 168, crm3_client.js linje 16 og 29, m.fl.
- Anvendelse af eval-funktionen på steder, hvor der findes alternative og mere hensigtsmæssige løsninger. Specielt anvendes 'eval' også i nyere dele af Javascript-koden. 'eval' har historisk været en meget anvendt funktion, men allerede tilbage i 2007/2008 begyndte man at erkende de mange problemer ved anvendelsen. JSLint advarer imod anvendelsen.
- Ingen anvendelse af 'use strict'. 'use strict' er en ny facilitet i ECMAscript version 5 (Javascript), der er fuldt bagudkompatibel med tidligere versioner af Javascript. Det er almindelig best practice at anvende 'use strict' i Javascript også selv om koden ikke forventes at skulle afvikles i en Javascript fortolker, der understøtter det pragma.

Javascript-kode vedligeholdes fra Visual Studio 2008 direkte gennem Team Explorer. Dette giver kun simpel kode highlight og få andre funktioner. Der eksisterer mange andre IDE'er og tools der rummer en væsentligt bedre understøttelse af Javascript. Givet den store mængde Javascript-kildekode så vil det tjene til at højne kvaliteten og produktiviteten at benytte af en eller flere af disse.

Programmeringsparadigmet er næsten udelukkende proceduralt og der er identificeret meget forretningslogik i brugergrænsefladen. Dette er forventeligt givet Captias client/server arkitektur, men strider mod nyere arkitekturprincipper, hvor man af hensyn til vedligeholdbarhed og fleksibilitet tilstræber helt at undgå at placere forretningslogik i præsentationslaget.

Der er cirka 100.000 linjers Javascript-kode, hvoraf cirka 30.000 linjer er SDD-relaterede kommentarer og udkommenteret kode.

Mønsteret med få meget store filer/funktioner fra C#-sporet gentager sig for Javascript-kodens vedkommende, og der observeres en del duplikering af funktioner.



For eksempel afslører en manuel analyse af den client-side funktionalitet, som er lavet for at implementere 'SDD Skærmbillede D0019 Fængslet V0027', hele fire duplikerede funktioner (convertDatoTidtilDato, FejlKoderSomAsp, getPolErrMsg, MergeDynamicMsg, onInitUdforendeAfd) med funktionelt identiske implementeringer, men med subtile forskelle som indikerer at kun nogle af duplikaterne er blevet vedligeholdt i forbindelse med oprydninger. Samme manuelle analyse afslører at der er mindst fem roundtrips til serveren via SOMASP alene for at få initialiseret viewet. Og så er der tilsyneladende en inkonsistens mellem SDD og implementeringen, eftersom SDD kræver at 'Id-nr D0019-017 skal filtreres via F0259', men F0259 findes ikke i, hverken Javascript- eller C#-kodebaserne.

Et godt eksempel på den manglende indkapsling af forretningslogik bag passende abstraktioner kan observeres ved at værdien af feltet "GK_HOVEDKATEGORI" indgår i if-else og switch konstruktioner omtrent 60 steder i Javascript koden.

Javascript-funktionalitet inkluderes i runtime-miljøet ved at placere source-filerne i et specifikt bibliotek under Captia-installationen, som Captia kigger i by default. Den Javascript-funktionalitet som skal anvendes i et view inkluderes i viewets layoutfil (standard Captia style).

Der er ikke support af at Javascript-filer kan inkludere andre Javascript filer, og der benyttes ikke dynamisk load i scripts. Derfor bliver man er nødt til at lægge alle includes i den yderste layoutfil.

Der er ingen support i POLSAG for at få kontrolleret om alle skærmbilleder er konfigureret korrekt i forhold til Javascript-funktionalitet. Det kan kun detekteres manuelt ved at navigere rundt i UI og holde øje med dels browser-rapporterede Javascript-fejl, dels manglende funktionalitet, når der trykkes på knapper mv.

Der er 178 layout-filer i POLSAG. Der er 1380 includes af javascript-filer i disse. Altså i gennemsnit 7 includes per layout.

Der er 418 Javascript-filer ialt. Af disse er der 84 som ikke eksplicit er inkluderet i nogen layoutfil (se 'Javascript-filer som ikke eksplicit er inkluderet i layout-filer' for en komplet liste).

Det bemærkes at der er steder i Javascript-koden, hvor der eksekveres flere på hinanden følgende opdateringer, og dermed potentiel risiko for data-inkonsistens. Eksempler:

- funktionen "rekvir()" i customControl_Sigtelse.js
- funktionen "SletFrihedsberovelseF0092" i D0019_D0021.js
- funktionen "F0886MaalePaavirkningTilknytTilFUH" i D003_F.js
- funktionen "fjernKR_oversendelse" i D0003_D0019_D0021_D0031_F.js
- funktionen "TilfoejSigtelserTilSAS" i customFunctions_sagsstyr.js
- funktionen "sasTilfoejSigtelseTilSAS" i customFunctions_sagsstyr.js
- funktionen "opretSigtelsesVandring" i D0024_M.js
- funktionen "personkategori_am_udskil" i F0081.js
- funktionen "SaveAnmeldelseHandler" i D0005_F.js

som alle enten først laver en opdatering gennem BusinessProcessManager og dernæst kalder SOMSAP's UpdateFromUrl endpoint eller laver to eller flere på hinanden følgende opdateringer igennem BusinessProcessManager. Hvis det er meningen at de data som opdateres ikke behøver at være konsistente er dette selvfølgelig ikke det store problem. I modsat fald skal opdateringerne samles server-side, hvis man skal sikre mod inkonsistens.

De to første eksempler blev rapporteret til ScanJour i forbindelse med det teknisk review i sommeren 2011, hvor Jan Olsen meldte at det er en fejl at der ikke er blevet ryddet ordentligt op i JavaScript-koden efter introduktionen af Business Process Manager. Ved det aktuelle code review gennemført i november 2011 eksisterer de to rapporterede problemer fortsat.

Der er 56 anvendelser af url'en UpdateFromUrl.asp spredt rundt omkring i Javascript-koden. Der har ikke været udført en gennemgang af alle disse, samt deres anvendelse fra andre dele af Javascript-koden for at kontrollere om der findes tilsvarende opdateringsmønstre andre steder i kodebasen.

Det er vores opfattelse, at den begrænsede kvalitet og manglende arkitektur i den store Javascript-kodebase fremadrettet vil gøre det meget svært at vedligeholde og udvikle POLSAG. Javascript-kodebasen bærer tydeligt præg af, at den er ikke implementeret efter almindelig best practice[16], der er ikke anvendt designprincipper for objekt-orienteret programmering og der er ingen unit test til at sikre integriteten og kvaliteten af koden i forbindelse med vedligeholdelse og udbygning.

Javascript kodebasen vurderes heller ikke at være blevet opdateret relativt til den store nye viden omkring Javascript og anvendelsen i web appilkationer der er etableret de seneste 4 år. Vi vurderer i øvrigt at en standard JSON-implementering vil kunne erstatte de proprietære tiltag (JSON formatter), som ScanJour har udviklet og derved tjene til at reducere kompleksiteten.

[16] "Javascript: The Good Parts", Douglas Crawford, 2008

C++

Det er ganske omstændeligt at komme til at kunne kompilere C++-kildekode. For at gøre dette skal man følge en 42-punkts installations-guide.

Såfremt man har brug for at udvikle drivere er der et ekstra dokument med rettelser til 42 punktsguiden samt ekstra opsætning. Der er endvidere en del manuelle skridt forbundet med at skubbe en ny release ind i source control (man skal lægge binaries manuelt ind i TFS).

Der er knap 13.700 linjer C++-kode (knap 8.300 kodelinjer excl. kommentarer).

Koden er, helt som for C#- & JavaScript-delen, procedural og langt mere data-orienteret end objekt-orienteret. Der er ingen indpakning af feltnavne i datamodellen.

Der er 348 unit tests, fordelt på 14 fixtures. Der observeres mange DRY violation i test setup-koden, men anti-pattern AssertionRoulette observeres dog kun få steder. Alene baseret på antallet af unittests relativt til antallet af kodelinjer er C++ kildekoden væsentligt bedre omfattet af automatiserede tests ifht øvrige dele af POLSAG.

PL-SQL

POLSAG består af omkring 35.000 linjers PL-SQL kode. Der er ingen test coverage af denne del af koden.

ScanJour rapporterer at man primært anvender PL-SQL i forbindelse med performanceoptimeringer af views. De anvendes også når SOM's register-model kommer til kort, hvilket f.eks. er tilfældet, når en use case kræver at et register skal hente data fra grand child-tabeller.

Triggers anvendes primært til at håndtere skema-opdateringer og til at holde eventuelle cache-tabeller i sync. Eksempel på en cache-tabel er 'nuværende sagsplacering', der er meget tung at beregne og som ikke var en del af det oprindelige SDD-design.

Under det tekniske review fik vi ikke adgang til nogen dokumentation af alle de efterhånden mange tabeller, triggers og views, som ikke er en del af SOM-modellen. Indtil dette måtte være tilvejebragt er den eneste oplagte vej til at danne sig et overblik er således en kombination af Oracle's dependency analyse tools og data dictionary-definitionen. Det var således ikke umiddelbart muligt at grave dybere på denne kant.

Navngivningstandarden for SQL scripts er ikke dokumenteret.

Da det er muligt at omgå SOM's indbliksstyring fra denne del af kodebasen er det udelukkende op til den enkelte udvikler at sørge for altid at repetere indbliks-filtreringen fra SOM i SQL-baserede views. Skulle man få brug for at tilføje indbliksfiltrering på eksisterende tabeller, enten direkte eller via nedarvning af indblik, skal man således sørge for at gennemgå PL-SQL koden for at sikre at denne ændring slår igennem alle steder, hvor disse tabeller måtte være anvendt. Denne mulighed for at omgå indbliksstyringen i SOM anvendes i øvrigt aktivt til oversendelser til anden kreds.

Da det er muligt at omgå SOM's indbliksstyring fra denne del af kodebasen er det udelukkende op til den enkelte udvikler at sørge for altid at repetere indbliks-filtreringen fra SOM i SQL-baserede views. Skulle man få brug for at tilføje indbliksfiltrering på eksisterende tabeller, enten direkte eller via nedarvning af indblik,

skal man således sørge for at gennemgå PL-SQL koden for at sikre at denne ændring slår igennem alle steder, hvor disse tabeller måtte være anvendt. Denne mulighed anvendes i øvrigt aktivt til oversendelser til anden kreds, hvilket er designet i samarbejde med RP, og giver i dette tilfælde ikke visningsrettigheder til nogen der ikke har det i forvejen.

Konklusion: Der mangler overblik over tabeller triggers og views. Indbliksstyringen i SOM kan ved kodefejl ikke blive aktiveret, hvilket medfører risiko for at fejl i forretningslogikken eller manglende viden hos udviklerne medfører at der implementeres manglende eller utilstrækkelig adgangskontrol. I den nuværende implementering benyttes muligheden for at omgå indbliksstyringen kun aktivt i forbindelse med at overføre sager til en anden kreds, hvilket er aftalt med Rigspolitiet. Leverandøren udtaler, at såfremt al adgang sker igennem SOM, så ville ovenstående risiko ikke komme til udtryk.

Konklusion

Der er ingen konsistent navnestandard for kildekoden. Konsekvensen af dette er at selv rutinerede ressourcer kan have vanskeligt ved at navigere rundt i kodebasen og dermed overskue hvor en given funktionalitet aktiveres/reageres på.

Den nuværende konvention om at anvende fortløbende nummererede navne uden domæne-betydning rapporteres at være lettere at navigere i end en tidligere konvention som forsøgte at anvende domæne-relaterede navne til objekter og metoder. Imidlertid er der ikke foretaget nogen oprydning af den del af kodebasen, som anvender andet end den nyeste navngivning, hvilket medfører at navngivningen ikke er konsistent.

Dette betyder at ramp-up tiden for nye folk alt andet lige vil være længere end hvis der eksempelvis var anvendt forretningsnære navne (se for eksempel 'Domain-Driven Design: Tackling Complexity in the Heart of Software (af Eric Evans)' for en glimrende udredning af, hvordan et klart, utvetydigt og konsekvent sprogbrug bidrager meget positivt til udviklingen af software).

Det er vanskeligt at danne sig et overblik over den samlede kodebase og der er ingen tooling support til at afhjælpe dette. Nye ressourcer skal således sætte sig ind i hundredevis (hvis ikke tusindvis) af ikke-intuitive navne fra SDD'er, hvis de skal være i stand navigere rundt i kodebasen på nogenlunde effektiv vis.

Tre af de fire anvendte programmeringsplatforme (C#, C++ og JavaScript[17]) er baseret på objekt-orienterede programmeringsparadigmer. Men langt størsteparten af den gennemsete kode er af procedural karakter, og store mængder funktionalitet er samlet i ganske få 'God-like classes/methods', som er vanskelige at vedligeholde/videreudvikle. Der henvises i øvrigt til 'Kodemetrikker for C#-delen af POLSAG. Se det vedlagte dokument POLServicesCodeMetrics.xlsx' for yderligere detaljer.

Der er ikke i forbindelse med reviewet fundet eksempler på konsekvent anvendelse af veletablerede OO-designprincipper fra f.eks. 'Gang of Four' [Design Patterns: Elements of Reusable Object-Oriented Software (af Gamma, Helm, Johnson, Vlissides)], **S.O.L.I.D.**, **DRY** eller **POLA**.

[17] JavaScript er ikke et klassebaseret sprog, som C++ og C#, men understøtter objektorienterede koncepter som tillader bedre strukturering af kode end hvad der p.t. er situationen i POLSAG.

Det forhold at der ikke er blevet udført en konsekvent oprydning af kodebasen, såvel i forhold til 2009 reviewets anbefalinger, som en del af en fortløbende proces, bidrager til at reducere læsbarheden og overskueligheden af kodebasen ganske væsentligt. Der henvises bl.a. til 'Clean Code: A Handbook of Agile Software Craftsmanship (af Robert C. Martin)' for en udredning af, hvorfor oprydninger (og konsekvent minimering af antallet af kodelinjer i en kodebase) bidrager meget væsentligt til en kodebases vedligeholdbarhed og udvidbarhed.

Test-strategi

Hovedparten af testarbejdet foregår manuelt.

Der er opsat en leverandør-specifik funktionel test suite i QTP der køres hver nat. Det er oplyst at denne omfatter 75% af *funktionaliteten* i POLSAG. Der er ikke nogen evaluering af hvor stor kodedækning der opnås igennem denne, og den funktionelle test-suite tester ikke fejlhåndtering.

Derudover er der nogle få automatiserede tests i C#- og C++-sporene. Disse tests afvikles dog kun manuelt, når udviklerne finder det nødvendigt. Det bemærkes at det ikke var muligt at kompilere C#-test suiten under det tekniske review så vel som det aktuelle code review og at der dermed de facto ikke er nogen test coverage af den del af kodebasen.

Der er ingen automatiserede tests til PL-SQL koden, og Javascript-delen er kun dækket via manuelle tests og QTP testen der kun sikrer at systemet overordnet set fungerer.

Det kan således konstateres at det nuværende test coverage er utilstrækkeligt til at forhindre fejl i forbindelse med koderettelser eller -udvidelser. Mens review'et blev udført on-site observeredes et eksempel på at introduktionen af et nyt HTML element i et view triggede en fejl i den logik, som behandlede opdateringer for et andet view.

Dette betyder samlet sét at enhver ændring eller ny feature i POLSAG kun kan verificeres manuelt ved hjælp af en debugger. Udviklerne rapporterer således at de debugger virkeligt meget, og indikerer at det ville være praktisk umuligt at foretage modifikationer uden en sådan.

Der er samtidig intet som forhindrer at andre code paths end de som trigges manuelt går i stykker i forbindelse med udførelse af ændringer. Og der er ingen automatiserede tools til at bistå med at få trigget alle relevante code paths.

Test-drevet udvikling

Hvis test skal blive en effektiv støtte under udviklingsprocessen kræver det, at systemet udvikles med henblik på effektivt at kunne testes. Unit test har igennem mange år været en del af de gængse practices til at kunne sikre kvaliteteten af kode med kompleks logik. Unit test er i det sidste årti blevet yderligere udbredt (og effektiviseret) med disciplinen test-drevet udvikling (Test Driven Development), som dog reelt blot handler om at anvende almindelig best practice for softwareudvikling (separation of concerns, single responsibility, DRY, modularitet, etc.) til at forbedre muligheden for at teste systemet.

I POLSAG ses mange tilfælde af funktioner der ikke overholder "single responsibility" pattern, hvor funktionen kun varetager een funktion. Antallet og dybden af disse tilfælde vidner om at koden ikke er implementeret med henblik på at kunne testes automatisk.

F.eks. er der følgende metode fra Døgnrapporten:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

DrRapport.ThisDocument

Her ses fire problemer i forhold til navngivningen af metoden og i selve implementeringen:

1. I linje 5 defineres et "magic number". Hvis tallet 5 en dag ikke længere er korrekt, så skal man altså ned i koden og ændre det og rekompilere applikationen.
2. Dette "magic number" stemmer ikke overens med den tilhørende kommentar i linje 5. Iflg. kommentaren må der kaldes med max 25 fileKeys, men det valgte tal medfører at der kaldes med max 5 fileKeys, hvilket i øvrigt resulterer i 5 gange så mange round trips til POLSAG for at hente data.
3. I linje 14 kaldes direkte til POLSAG via SOMASP (metoden DRRapport.HentDataFraPolsagTemplate er en wrapper om SOMASP asmx servicen).
4. I linje 17 sorteres resultat til en bestemt sortering inden det returneres. Ifht at genbruge dette kald er det uhensigtsmæssigt, da det ikke er oplagt af navngivningen at metoden returnerer sorteret, og kalderen ikke selv kan bestemme om data skal sorteres og hvordan (risiko for dobbeltsortering).

Skal metoden "hentDataFraTemplate" unit testes medfører ovenstående at:

1. Der skal findes et komplet kørende POLSAG system for at det bliver muligt at teste en relativt simpel funktion.
2. Det er ikke muligt automatisk at teste effekten af at ændre det magiske tal i linje 5.
3. Metoden "SortedList" skal være implementeret før "hentDataFraTemplate" kan testes.

Hvis metoden "hentDataFraTemplate" var unit testet havde man sandsynligvis også opdaget fejlen i linje 11, der gør at metoden kun virker med det magiske tal 5, fordi også står skrevet direkte på linje 11.

Der ses eksempler på denne manglende "separation of concerns" der gør der meget svært at teste logikken. Nogle af disse er:

- DocumentScan.DocumentScan.IsLatestVersion
- DrRapport.ThisDocument
- ScanJour.CentralPrint.CentralPrintImport.MoveFile

Et andet tegn på at der ikke udvikles med henblik på unit test ses af at der i koden indgår logik til håndtering af "testemode" i forskellige varianter, f.eks. i samme klasse:

```
1
2
3
4
5
6
7
8

9
10
11

12
13

14
15
16
17
18
19
20
21
22
23
24
25
26

27
28
29
30
31
32
33
34
```

DrRapport.ThisDocument

Her ses følgende:

1. I linje 15 og 18 benyttes den særlige testmode til at bestemme opførsel. Parameteren "testmode" er defineret med flg. kommentar i koden:
   "`bool testMode = false;  //@@@ Skal sættes til false før indtjekning`".
2. I linje 22 og 31 er angivet navne på specifikke testsystemer.
3. Linje 23-34 er beregnet på at der manuelt indkommenteres en linje ifbm test.

Dvs den kode der kompileres til Døgnrapporten indeholder også kode til testformål, hvilket er meget uhensigtsmæssigt fordi der er risiko for at kode til testformål bliver kaldt af kørende produktionskode. Derudover er den eneste måde, hvorpå man kan sikre at "testmode" ikke er aktiveret i en produktionsversion, at udvikleren husker at sætte den til false inden koden bliver tjekket ind. Tilsvarende findes der seks udkommenterede kald til "`System.Diagnostics.Debugger.Launch()`" i solution Døgnrapporten og POLServices. Disse kald vil forsøge at starte en debugger, hvis de ikke er udkommenteret.

Der findes ligeledes eksempler på testkode der er indlejret i produktionskoden. Et par eksempler er:

- SagsstyringsReol.SomFactory.GetDsnAndHostName
- POLSAG.Common.Functions.F0852
- POLSAG.DataAccess.Registers.SigtelsesRegister
- POLSAG.Agents.POLSAG_ANM_AGENT.POLSAG_ANM_AGENT (flere tilsvarende)
- POLSAG.Agents.F0727
- Funktionen F0888 i "I0256.js"

I dele af POLSAG-koden ses forsøg på at implementere mindst een unit test per C# klasse. Hvis man f.eks. kigger på kontrol- og funktionsklasserne (Kxxx hhv Fxxx) ses følgende:

Klasse	Antal	Antal unit tests	Heraf deaktiveret	I procent
Fxxx	115	65	12	18,5%
Kxxx	137	111	28	25,2%

De fleste tests er deaktiveret ved at udkommentere koden og indsætte (37 ud af 40) linjen "`Assert.Inconclusive("due to refactoring")`", og det kan derfor ikke udledes om testen ikke længere er relevant (og måske reelt bør slettes?) eller om testen ikke længere er korrekt (f.eks. pga manglende vedligehold). Omkring 20% af kontrol- og funktionsklasserne er på nuværende tidspunkt slet ikke er dækket af nogen unit tests. Ved det tekniske review i foråret/sommeren 2011 kunne unittesten ikke umiddelbart kompilere. Det samme problem eksisterer også for materialet udleveret som del af baseline til det aktuelle code review. Der er derfor defacto ingen unittest dækning af POLSAG.

Konklusion

POLSAG lider i udpræget grad under manglen på automatiserede unittests på kodebasen. De høje tilbageløbsprocenter som er dokumenteret i næste afsnit er efter vores vurdering en klar indikation af at det er forbundet med væsentlige vanskeligheder at vedligeholde den nuværende kodebase.

Alene systemets karakteristika gør at vi vurderer det er nødvendigt at Rigspolitiet foretager komplette systemtests før hver produktionssætning af nye releases for at undgå regressionsfejl. Disse system-tests må nødvendigvis også inkludere scenarier, hvor batch-agenterne har været aktiveret for at kunne verificere at de ændringer som disse foretager i data er korrekte.

Det er tvingende nødvendigt at en væsentlig andel af disse tests er automatiserede, da det ikke er økonomisk og tidsmæssigt holdbart at denne del af testen tager mindst 75[18] manddage.

Vi anser på den baggrund den manglende automatiserede test dækning som et meget væsentligt problem i forhold til udviklings- og vedligeholdelsesprocessen for POLSAG, eftersom udviklerne ikke automatisk og hurtigt bliver gjort opmærksom på at de har introduceret fejl.

Dette forhold sammenholdt med at det er praktisk taget umuligt at lave statiske impact-analyser af ændringer betyder at man ikke umiddelbart kan forvente at se et fald i antallet af tilbageløb, hvis ikke der sker en væsentlige ændringer til det bedre på dette område. Hvis ikke udvikleren som foretager en ændring har næsten komplet overblik over samtlige funktionaliteter og deres indbyrdes implicitte afhængigheder i systemet er der jo så at sige næsten meget få og små muligheder for at det er muligt at modificere den eksisterende kode uden at løbe en høj risiko for at introducere fejl.

[18] Baseret på information fra telefoninterview med Michael Englev, RP

Funktionelt testforløb

Der udføres funktionel test i alle led i leveranceprocessen.

Test hos ScanJour

Johnny Sabinski fra ScanJours interne QA-afdeling rapporterer at de mener at dække omkring 75% af al funktionalitet ved hver test cycle. QTP test suiten indgår i dette. Det meldes fra ScanJour QA at der er tilbageløb fra Rigspolitiet på omkring 20% af alle fejlrettelser, samt på 10-15% af nyudviklingen.

ScanJours minimal-test (som foretages ifm hotfix-releases) tager 10-12 timer at gennemføre. Der er 3-4 personer involveret i denne test.

Vi er ikke i stand til at afgøre hvorvidt Scanjour QA's fornemmelse af testdækningen er korrekt, når vi vurderer op imod at Rigspolitiet kun mener at være i stand til at dække 2/3 af løsningen med en funktionel test, der strækker sig over ca. 80 manddage.

Begge testafdelinger medregner i øvrigt ikke andet end slutbruger-rettet funktionalitet. Batch jobs og integrationer er således ikke medtaget i ovenstående tal.

Der er aldrig foretaget egentlige code coverage målinger af test suiterne, hvorfor den faktiske dækningsgrad relativt til kildekoden er ukendt.

Jan Olsen (de facto dev lead fra ScanJour) rapporterer at mange af de oplevede fejl kan henføres til forskelle i opsætningen af stamdata hos hhv. Rigspolitiet og ScanJour (f.eks. organisationsopsætningen). Denne opfattelse deles tilsyneladende ikke af Rigspolitiet.Der hersker dog enighed om at forskelle i stamdata i de forskellige miljøer gør det svært at vide om der er tale om fejl er i systemet eller i opsætningen.

Test hos CSC

CSC's tester kun de rettelser som er indeholdt i hver release. Der har indenfor de seneste cirka 8 måneder været mellem 30 og 40 releases ifølge Mikkel Nielsen.

Der testes for defects, men ikke for regressionsfejl. Det er ifølge Dorte Krogsgaard ikke altid at releases indeholder de forventede defect-rettelser, og det er heller ikke altid tydeligt at de modtagne rettelser er testet af SJ før de leveres, eftersom rettelserne ikke altid virker korrekt.

Nogle batch-agenter testes først i produktion ifølge Mikkel Nielsen. Lars Bech Rasmussen oplyser at dette handler om den fysiske flytning af data via OPC-schedulerede FTP-jobs.

Mikkel Nielsen rapporterer en tilbageløbsprocent på 20-30%. CSC mener ikke der er tid nok i planen fra Rigspolitiet til at de kan nå at teste ordentligt, og dette vanskeliggøres yderligere af de meget hyppige releases.

Afslutningsvis foretages en testbarhedstest af CSC før en release frigives til Rigspolitiet. Varigheden er cirka en halv dag.

Test hos Rigspolitiet

Efter endt aftestning hos CSC testes nye releases hos Rigspolitiet.

Her foretages en separat testbarheds-test som tager en dag til halvanden. Denne test indeholder et par hundrede test cases.

Herefter starter den egentlige funktionelle acceptance-test. Der har været brugt cirka 110 manddage på denne indtil videre i forbindelse med UDV6 releases. Et fuldt test-forløb tager cirka 75 manddage, og Rigspolitiet mener at testen dækker cirka to tredjedele af kernefunktionaliteten i det typiske daglige politiarbejde.

Sideløbende foretages der regressionstests, såfremt der er overskydende tid i kalenderen. Defect-rettelser gentestes og der rapporteres en tilbageløbsprocent på 20%. Forskelle i de involverede it-miljøer, uklare fejlrapporter og misforståelser på udviklingssiden bidrager alle til tilbageløb.

Ud af de cirka 650 defects som har været rettet i år har der været opdaget følgefejl i et sted mellem 40 og 80 tilfælde, spredt ud over alle fejlkategorier. Cirka 10 fejl er sluppet helt ud til brugerne. Brugernes fejlbeskrivelser er ofte vanskelige at kommunikere videre, hvorfor der forekommer et større niveau af dialog i forbindelse med rettelser af bruger-opdagede fejl.

Der vurderes p.t. ikke at være nogen indikationer for at mængden af fejl er for nedadgående. Der findes stadig lige så mange nye fejl som der bliver rettet. Der ses dog kun meget sjældent kategori 1-fejl på nuværende tidspunkt. Fejlene ligger nu primært i kategori 2 og 3. De mindst betydende kategori 4-fejl logges ikke længere.

Fremadrettet er der ikke en målsætning om at lave fuld regressionstests. Det er planen at fordele testindsatsen med 50% til forretningskritiske testdrejebøger, 25% til eksploratoriske tests og 25% til at teste funktionalitet der ikke har været testet længe.

Der eksisterer et sted mellem 2.000 og 4.000 testdrejebøger. Cirka 10% af disse menes at være forældede.

Udfordringer ved nuværende testforløb

Rigspolitiet ser følgende udfordringer ved det nuværende testforløb:

- Configuration management er ikke optimal – Dette gælder ikke mindst for stamdata. Stamdataopsætningen er ikke den samme i test og produktion, hvilket både giver spildtid i testen og en risiko for at testen ikke er dækkende for systemets opførsel i produktion. Disse problematikker er især rodfæstet i den begrænsede understøttelse for change management af stamdata.
- Mere stabilt leverance-forløb – Der er mange rettelser op til skæringsdatoerne, og der låses ikke for feature-udvikling op til releases. Der ønskes en stabiliseringsfase før releases, hvilket imidlertid konflikter med at der er mange kritiske rettelser som skal med hver gang.
- Det har vist sig at forbedre processen, når Rigspolitiet hjælper CSC med deres test, idet det letter videnoverdragelsen på tværs af parterne.

Vurdering af det funktionelle test forløb

Størrelsen af den testbyrde som påhviler Rigspolitiet sammenholdt med observationerne fra kodereview'et giver anledning til bekymring. Bekymringen bunder i at kombinationen af at fejlretning ofte giver anledning til følgefejl og det forhold at en fuldt dækkende test ikke er realistisk er erfaringsmæssigt nærmest dømt til at resultere i, at der er en del fejl, som nødvendigvis først vil blive opdaget når de rammer brugerne i produktionen.

Anbefalinger

Såvidt vi kan vurdere arbejder Rigspolitiet målrettet og professionelt med testopgaven. Det mest oplagte sted at sætte ind over for omfanget af regressionsfejl (og fejl, der ender med at komme helt ud til brugerne) er således at reducere arbejdsbyrden ved den funktionelle test samt øge dækningen af samme.

Rigspolitiet arbejde selv på at udvikle en strategi for udvælgelse af den test-delmængde, som giver den største reelle testdækning. Vores anbefaling er dog at man i højere grad bør fokusere på at reducere testbyrden ved at automatisere så store dele af disse tests som det er økonomisk og praktisk muligt.

Det må desværre forudses at være en meget stor opgave at automatisere den funktionelle test, hvorfor det anbefales først og fremmest at øge testkvaliteten under udviklingen, hvor effektiviteten er langt større og byrden med at holde testene opdateret er mindre.

Det er især være følgende typer funktionalitet der med fordel kan unit testes:

- Batchoperationer
- Kontroller
- At data kan gemmes og hentes igen

Som nævnt andetsteds er etableringen af et passende niveau af unit tests i Scanjours kode en udfordring, idet koden ikke er designet med automatisk test for øje. Givet de omkostninger, der er forbundet med testen hos Rigspolitiet, samt udfordringerne med at vedligeholde koden ser vi ikke noget alternativ til at investere massivt i automatiske tests med det samme.

Stamdata

En stor det af POLSAG's funktionalitet er styret at konfiguration af stamdata, f.eks. ændrer brugergrænsefladen opførsel baseret på stamdata.

Det er Rigspolitiet som ejer og vedligeholder det komplette sæt af stamdata. Hidtil har produktionssætninger baseret sig på enten en komplet load af stamdata eller ingen ændringer til stamdata.

Der er nu efter første produktionssætning ved at blive fastlagt en strategi for, hvordan man skal håndtere change management. Eksisterende stamdata kan i praksis ikke slettes, og vil derfor blive de-aktiveret i stedet.

For at give en indikation om omfanget af POLSAG stamdata så tager det angiveligt 4-5 timer at installere et komplet sæt af stamdata i en 'frisk' database.

Konklusion

Baseret på den høje kompleksitet i POLSAG stamdata vurderes det at udgøre en risiko at der ikke er fastlagt en procedure for håndteringer at stamdata ændringer.

De nuværende værktøjer i POLSAG betyder angiveligt at stamdataændringer en manuel proces, som udføres gennem CAPTIA Configuration manager. Stamdataændringer skal derfor manuelt flyttes gennem de forskellige testmiljøer med den deraf følgende risiko for inkonsistens til følge.

Ydermere gør mængden af stamdata det kombinatorisk umuligt at teste alle kombinationer af opsætning, hvilket kun tjener til at øge risikoen omkring stamdataopdateringer.

Logning

Captia indeholder en standard use-log.

Derudover er der er bygget en POLSAG-specifik såkaldt SOAPLogger, som logger alle kald til og fra ASMX web services i POLSAG. Logningen foregår asynkront (hvilket formodes at ske af hensyn til performance).

Der er ikke specificeret hvilke web service-operationer som skal logges, eller hvilke delmængder af beskederne som skal logges, hvorfor alt angiveligt bliver logget. Hvis logning til databasen fejler skrives der en kort meddelelse i Windows Application Event Log, som imidlertid ikke indeholder request/responses.

POLSAG-applikationen logger rigtigt meget data til databasen. Der blev logget ca. 4 millioner rækker i timen under CSC/Scanjours performancetest.

Det syntes på den baggrund at være relevant at der bliver lagt en strategi for, hvordan det forskellige logs skal håndteres (og hvorvidt det reelt er nødvendigt at logge alt) inden POLSAG går i drift.

Konklusion

Der er risiko for at ikke alle ASMX-adgange til data logges, fordi logningen foregår asynkront. Der er således ingen garantier for at logningen af web service-operationer kan anvendes til at rekonstruere et fuldt hændelsesforløb.

Der eksisterer desuden ingen procedurer for, hvordan der sker backup af Windows Event Loggen, eller nogen beskrivelser af, hvordan man undgår at Windows Event Log'ens almindelige rollover-indstillinger fører til overskrivning af fejlbeskeder.

Udviklingsproces

Der er ikke foretaget en egentligt review af udviklingsprocessen i dette review. For at kunne vurdere mulighederne for videreudvikling og vedligeholdelse af POLSAG har vi til gengæld udført et overordnet review af den tilgængelige udviklerdokumentation.

Overblik

Den metode der er anvendt i POLSAG-projektet er i store træk som følger:

- Scanjours analytikere arbejder sammen med Rigspolitiet omkring skærmbilleder/funktionelle områder i et workshop-lignende forløb. Resultatet bliver dokumenteret i SDD'er. Nogen gange er

workshops dokumenteret som billeder af whiteboards. Disse billeder befinder sig på udviklernes pc'er og kan i nogen grad genfindes dér.
- Scanjours udviklere implementerer funktionaliteten på baggrund af SDD'erne.
- CSC og RP testere udvikler testdrejebøger baseret på SDD'erne.

Vi er af opfattelsen at denne udviklingsproces er noget speciel, idet der såvidt vi forstår rent faktisk bliver udført en funktionel analyse, som imidlertid ikke bliver dokumenteret. Det er derefter op til både udviklere og testere at udlede den funktionelle anvendelse ud fra SDD'erne, som mange af udviklerne selv betegner somsvært forståelige.

Vi vurderer at en væsentlig årsag til at denne proces overhovedet virker beror på at de medarbejdere der deltager i processen har været med i projektet længe og derfor i vid udstrækning er i stand til at udlede den nødvendige information baseret på deres hidtidige erfaring.

Status på dokumentationen

POLSAG har genereret en stor mængde udviklingsdokumentation. Der er følgende nøgletal for dokumentationen:

- 154 SDD'er for skærmbilleder.
- 770 SDD'er for funktioner.
- 207 SDD'er for lister.

Vi har ved stikprøver fundet at der hersker en god overensstemmelse mellem dokumenterne og den implementerede kode baseret på den overordnede struktur og de indbyrdes afhængigheder.

Vi har dog også (som forventet givet projektets omfang og type) fundet mindre inkonsistenser som f.eks. referencer i dokumentationen til funktioner, der ikke findes. Dette betyder at dokumentationen kan anvendes til støtte i forståelsen af kodens organisering, men at udviklerne tillige er nødt til at anvende fritekstsøgning for at etablere fuld forståelse for afhængighederne.

Dokumentationen (og koden) er et resultat af det skærmbillede-fokuserede udviklingsforløb, som udviklingsmodellen dikterer. Dette forhold kombineret med en proces, som har meget høje krav til detaljeringsnivau og godkendelser resulterer i en dokumentation, der først og fremmest rummer "kodenær" information.

Dette er i øvrigt en udfordring som Scanjour er bevidst om, hvorfor man i de seneste iterationer har gjort forskellige forsøg på at gøre specifikationen mindre kode-orienteret. Dette har dog vist sig at give nogen udfordringer i relation til at noget viden ikke er blevet kommunikeret til udviklerne.

Det er vores opfattelse at den foreliggende dokumentation har værdi set fra et kontraktuelt synspunkt, når det skal afgøres om der er udviklet det som er aftalt. Til gengæld har den meget lav værdi for de udviklere, der skal vedligeholde løsningen, da langt størsteparten af indholdet i SDD'erene er informationenr som normalt kan findes ved blot at læse kildekoden.

Set fra et vedligeholdelsesperspektiv er værdien ved dokumentationen således lav, idet tvivlstilfælde under alle omstændigheder vil kræve en større udredning for at afgøre om kildekoden eller SDD'erne har den rigtige tolkning af løsningen.

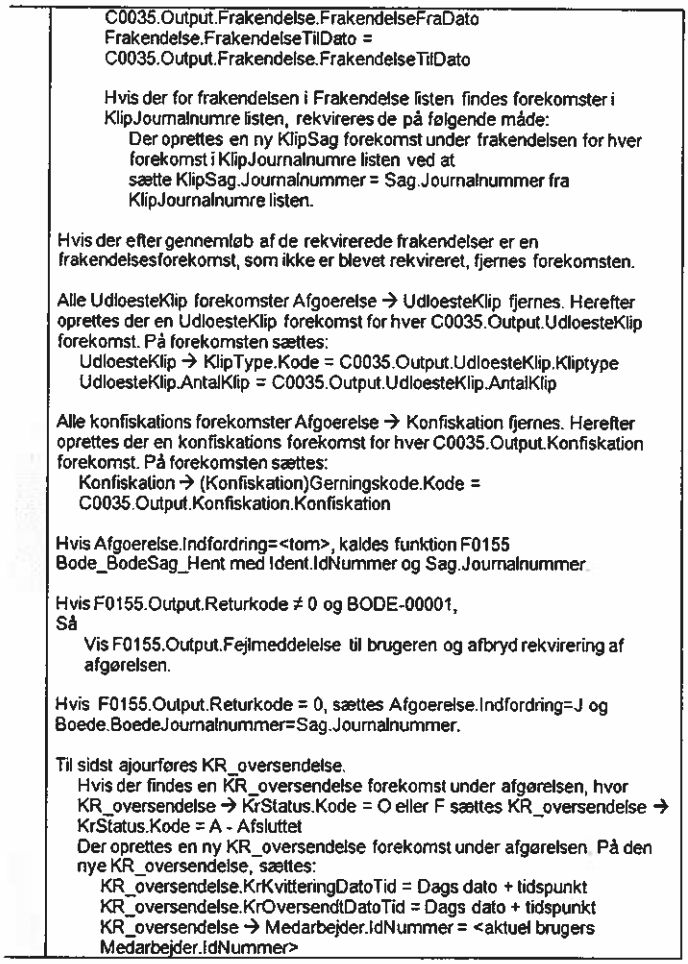
Vores anbefaling til forbedringer på dokumentationsområdet er:

Forbedret designdokumentation

Det er ikke muligt at komme med løsningen på projektets dokumentationsudfordringer i denne sammenhæng, da det forudsætter en indgående analyse af processen mellem Rigspolitiet og Scanjours analytikere og udviklere.

Med det sagt, så har vi dog observeret at det umiddelbart virker til at dokumentationen ville bliver betydeligt forbedret, hvis der blev arbejdet på flere niveauer

- Anvendelsesniveau (use case/scenarios) – Altså hvordan funktionaliteten forventes anvendt. Dette vil relatere sig direkte til de testcases som anvendes til at verificere implementeringen.
- Felt definitioner og regler – De nuværende tabeller er meget komprimerede og henviser typisk til koder eller noget indforstået "hentes/beregnes ...". Det anbefales at justere lidt i tabellerne, så pladsen bliver benyttet bedre og teksten bliver mere selvforklarende. Det drejer sig generelt om små justeringer, som virkelig bedrer overblikket (e.g. K0XYZ Check CPR nummer i stedet for blot K0XYZ)
- Logikken i funktionen – Selve logikken i funktionerne er alt for kompleks til at være indeholdt i en enkelt funktion. I nogen tilfælde får man reelt en fornemmelse af, at de store funktioner blandt andet skyldes at hvis man foretager en opsplitning kræver det endnu flere dokumenter. Et eksempel ses i Figur 2 Eksempel fra SDD. For at reducere kompleksiteten så det bliver muligt at vedligeholde er det nødvendigt at splitte funktionen op i mindre klumper og anvende UML-diagrammeringsværktøjer som sekvens- og aktivitetsdiagrammer for at give et overblik over funktionaliteten.
- Der bruges meget plads på at beskrive hvordan felter hentes og flyttes til andre objekter – Dette kunne med fordel enten flyttes til et Appendiks eller indpakkes i C#/Javascript-klasser som dokumenteres separat, hvis disse ønskes dokumenteret til bunds.



C0035.Output.Frakendelse.FrakendelseFraDato
Frakendelse.FrakendelseTilDato =
C0035.Output.Frakendelse.FrakendelseTilDato

Hvis der for frakendelsen i Frakendelse listen findes forekomster i KlipJournalnumre listen, rekvireres de på følgende måde:
Der oprettes en ny KlipSag forekomst under frakendelsen for hver forekomst i KlipJournalnumre listen ved at sætte KlipSag.Journalnummer = Sag.Journalnummer fra KlipJournalnumre listen.

Hvis der efter gennemløb af de rekvirerede frakendelser er en frakendelsesforekomst, som ikke er blevet rekvireret, fjernes forekomsten.

Alle UdloesteKlip forekomster Afgoerelse → UdloesteKlip fjernes. Herefter oprettes der en UdloesteKlip forekomst for hver C0035.Output.UdloesteKlip forekomst. På forekomsten sættes:
UdloesteKlip → KlipType.Kode = C0035.Output.UdloesteKlip.Kliptype
UdloesteKlip.AntalKlip = C0035.Output.UdloesteKlip.AntalKlip

Alle konfiskations forekomster Afgoerelse → Konfiskation fjernes. Herefter oprettes der en konfiskations forekomst for hver C0035.Output.Konfiskation forekomst. På forekomsten sættes:
Konfiskation → (Konfiskation)Gerningskode.Kode =
C0035.Output.Konfiskation.Konfiskation

Hvis Afgoerelse.Indfordring=<tom>, kaldes funktion F0155 Bode_BodeSag_Hent med Ident.IdNummer og Sag.Journalnummer.

Hvis F0155.Output.Returkode ≠ 0 og BODE-00001,
Så
Vis F0155.Output.Fejlmeddelelse til brugeren og afbryd rekvirering af afgørelsen.

Hvis F0155.Output.Returkode = 0, sættes Afgoerelse.Indfordring=J og Boede.BoedeJournalnummer=Sag.Journalnummer.

Til sidst ajourføres KR_oversendelse.
Hvis der findes en KR_oversendelse forekomst under afgørelsen, hvor KR_oversendelse → KrStatus.Kode = O eller F sættes KR_oversendelse → KrStatus.Kode = A - Afsluttet
Der oprettes en ny KR_oversendelse forekomst under afgørelsen. På den nye KR_oversendelse, sættes:
KR_oversendelse.KrKvitteringDatoTid = Dags dato + tidspunkt
KR_oversendelse.KrOversendtDatoTid = Dags dato + tidspunkt
KR_oversendelse → Medarbejder.IdNummer = <aktuel brugers Medarbejder.IdNummer>

Figur 2 Eksempel fra SDD

Etablering af vedligeholdelsesdokumentation

Givet situationen anbefales det ligeledes at der udarbejde dokumentation, som kan støtte den fremtidige vedligeholdelsesproces.

Denne dokumentation skal bygge på den eksisterende kodebase og ikke være en konkurrerende beskrivelse af det samme. Vedligeholdelsesdokumentationen skal også indbefatte information om sammenhængen i systemet.

Det er en stor opgave at vende dokumentationsfokus fra instruktioner om, hvordan koden skal implementeres til at give information om, hvordan systemet forventes at opføre sig. Selvom der ikke umiddelbart sker en definition af vedligeholdelsesdokumentationen her har vi dog følgende forslag:

- Det skal overvejes om Rigspolitiets brugermanual kan bruges som hjælp til at navigere i kodebasen, hvis den annoteres med numre på skærmbilleder og funktioner.
- Det er vigtigt hurtigst muligt at få nedfældet den viden, der befinder sig i hovedet på Scanjours medarbejdere. Denne analyse kan ses som første runde af den sådan aktivitet.
- Det bedste ville være, hvis man kunne organisere kodebasen på en måde, der gør det muligt at følge afhængighederne mellem modulerne direkte.

Risikovurdering af udviklingsprocessen

Vi har identificeret en del risici i relation til udviklingsprocessen.

Manglende overblik over den udviklede løsning

POLSAG-systemlandskabet er komplekst. Der er mange teknologier og programmeringsplatforme i spil, der er ikke nogen lagdeling i arkitekturen, der er ikke nogen domæne-model til at definere de centrale abstraktioner i Rigspolitiets forretningsmodel og man skal krydslæse mange SDD-dokumenter og forskellige kildetekster for at finde ud af dels, hvordan en given feature hænger sammen, dels hvor mange steder den er implementeret.

En af de største risici ved en sådan situation er at løsningen udvikler sig til at blive mere og mere låst, fordi ingen kan overskue konsekvenserne af rettelser. Vi er allerede stødt på følgende eksempler på dette

- Det vurderes for en for stor risiko at opdatere Captia platformen til version 4.5, selvom det angiveligt også ville medføre en række tekniske fordele.
- CSC forventer at opgraderinger kun kan foregå som "big-bang" opgraderinger hvor alle dele af systemet skal opgraderes samtitidig i forbindelse med ændringer
- Kode kopieres, hvilket fører til dublering af stort set samme kode, fordi man ikke kender de andre anvendelser samt ikke kan overskue hvordan det testes.

Omkostningstungt specifikationsforløb

Den nuværende udviklingsproces (dvs. selve processen såvel som SDD'erne), som finder anvendelse ved specificering af ny funktionalitet eller ændringer til det eksisterende, vurderes at komme til at påvirke POLSAG fremadrettet på flere punkter.

- Udviklingen af POLSAG tegner til at blive meget langstrakt og alt andet lige gøre business casen svagere.
- Brugerne vil opleve at der går meget lang tid før der kan reageres på deres feedback.
- Afhængigheden af leverandørens nøgleressoucer vil kun blive større henover tid.

Ingen vedligeholdelsesegnet dokumentation

Denne risiko vokser med tiden. Vi har flere gange under review'et forespurgt på information, som har vist sig at være forsvundet med medarbejdere, der har forladt Scanjour.

Det vurderes på nuværende tidspunkt at være forbundet med væsentlige økonomiske og praktiske problemer, hvis POLSAG skal vedligeholdes og videreudvikles uden de medarbejdere hos Scanjour og CSC, som har fulgt projektet gennem de sidste par år.

Udviklingseffektivitet

For at underbygge vores påstande om at det er vanskeligt og kostbart at udvikle videre på POLSAG har vi taget udgangspunkt i to eksempler fra SDD5/Polsas efterslæb.

Dyretransport

Formålet med dyretransport er beskrevet som følger "Der skal kunne registreres klip for dyretransportforseelser efter de samme principper som for klip ved færdselsforseelser. Dog må de to typer forseelser ikke blandes sammen."

For at implementere denne funktionalitet er der opbygget en pakke der indeholder følgende ændringer

- Fem skærmbilleder.
- 24 funktioner (hvor de 23 allerede eksisterer).
- Har resulteret i produktionen af i alt 46 opdaterede SDD-dokumenter.

Dette eksempel påviser flere udfordringer med POLSAG

- At logik og elementer er spredt over løsningen.
- At processen med at vedligeholde SDD'er ikke fungerer særlig godt med ændringer. De SDD'er vi har kigget på ved stikprøve har godt nok en ændringshistorik der nævner Dyretransport. Men det er ikke til at se, hvor meget der er ændret. Så i praksis bliver det meget svært at have mere end en ændring kørende pr. SDD.
- Det er overordentlig svært at dokumentere og vedligeholde de nødvendige funktionelle tests (både til godkendelsen af ændringen og de fremadrettede regression tests), da der ikke findes en samlet funktionel beskrivelse.

Det har ikke umiddelbart været muligt at fremskaffe informationer om estimater og det faktiske ressourceforbrug for denne opgave hos ScanJour.

Idræt

En noget mindre ændring omkring "Idrætsbegivenheder" som har følgende formål

"At kunne meddele en person generel karantæne (Forbud) mod at opholde sig ved bestemte idrætsbegivenheder indenfor en bestemt afstand. Evt. forbud gældende i bestemt tidsrum. Der kan tillige gives påbud om den sikkerhedsmæssige indretning. Efterkommes dette påbud ikke kan Politiet meddele forbud mod, at idrætsbegivenheden afholdes. Løsningen falder ind i det der er designet, derfor er det en udvidelse til det designede."

Dette eksempel viser et par aspekter ved udfordringerne ved projektet:

- Ved 1-til-1 konverteringen har man valgt at "bunke" idrætsfunktionalitet, selvom ovenstående i virkeligheden udgør to vidt forskellige processer.
- Politiets arbejde er meget processtyret og ovenstående mangler en masse detaljer og afhængigheder, som formentlig har været genstand for debat i workshops med Politiet, men aldrig er blevet dokumenteret. SDD'en indeholder kun de kodenære konklusioner, hvor der nedenfor er

vist et simpelt uddrag

> Der oprettes en ny kontrol til afgørelsesbilledet for at udføre den nævnte kontrol.
>
> Når IND har fået meddelt forbuddet ændres koden for "Afgørelsestype" til 40. Dette er allerede håndteret i kontrol 1383 for afgørelse.
>
> I "Frakendelse" sættes koden "FK" og længden på forbuddet angives med en "Fra dato". FrakendelseType Laengdekontrol skal være = 3 for den nye kode, hvilket betyder, at frakendelse fradato skal udfyldes.
>
> I anmærkningsfeltet skrives den tilhørende tekst, eksempelvis:"Må ikke tage ophold på og ved arealer, hvor der afholdes fodboldlandskampe". Der skal ikke foretages nogen kontrol af om feltet udfyldes.
>
> Afgørelse skal manuel indberettes til KR. Dette håndteres af funktionen F0143 BeregnAfgoerelseKR. Funktionen skal ikke rettes.
>
> Ved overtrædelse af forbuddet oprettes en ny sag med gerningskode 84377 – Karantæne ved idrætsbegivenheder, overtrædelse af forbud.
>
> På afgørelsesudskriften skal udløbsdatoen for påbuddet være anført, som i Polsas. Det antages, at udløbsdatoen er lig med Frakendelse tildato, hvorfor afgørelsesudskriften ikke skal rettes.
>
> Aktualitetsmarkeringer skal vises i relevante skærmbilleder. Hvis der er ændringer til aktualitetsbilledet i POLSAG forudsættes det løst ifm. ændringer til Aktualiteter i ændringsanmodningen Æ098.

- Det er ikke muligt for udenforstående ved at læse SDD'en at kunne verificere at reglerne faktisk dækker det nævnte formål. Der er f.eks. ingen referencer til påbud om sikkerhedsmæssige indretninger, hvilket formentlig skyldes at man opfatter det som en selvfølge at alle læsere ved at det allerede var implementeret i POLSAG.

Dette er en såkaldt lille ændring som kun dækker over 5 SDD'er. Her oplever man dog igen, hvor meget viden, der står "mellem linjerne". I ændringen nævnes "Ændring til frakendelsestype, SDD5 - datamodelændringer" som ikke indgår i baseline-dokumentationen, hvilket gør det meget svært at vide, hvad der er ændret. Det er formodentlig relateret til den med gult overstregede tekst.

Det har ikke umiddelbart været muligt at fremskaffe informationer om estimater og det faktiske ressourceforbrug for denne opgave hos ScanJour.

Udviklingsmiljøet

I et store og komplekst udviklingsprojekt som POLSAG er det vigtigt at fokusere på udviklingsmiljøet og den produktivitet det bidrager med.

Der er p.t. mange manuelle processer involveret i at udvikle funktionalitet i POLSAG:

- Captia-platformen tilskynder til at placere forretningslogikken i JavaScript.
- Captia-platformens proprietære syntaks for, hvordan man laver views og data binding i klienten er ikke understøttet af et tilsvarende værktøjssæt til at arbejde med den proprietære syntaks.
- Den meget lave test coverage og det manglende fokus på dels vigtigheden af, dels produktivitets-forøgelsen man får fra dækkende unit test suiter, betyder at udviklingen i dag som udgangspunkt foregår ved hjælp af debuggeren og at funktionalitet som udgangspunkt verificeres ved hjælp af brugergrænsefladen.

- De omfattende POLSAG-specifikke udvidelser, som er lavet ved siden af standard Captia, udvider antallet af steder, hvor der er implementeret forretningslogik, hvilket øger antallet af kodelinjer og teknologier, man skal overskue for at ændre/udvide funktionaliteter.
- Den omfattende brug af late binding og u-typede data input/output formater betyder mange af de fejl, som ellers kunne være blevet fanget af compileren, i stedet vil manifestere sig som runtime fejl, når systemet er i brug.
- Den omfattende brug af magiske strenge som fundament i logikken og dataudveklings-kontrakterne gør det nødvendigt at falde tilbage på fil-baserede søgninger i hele kodebasen, når man f.eks. skal vurdere hvilket navn man kan tillade sig at anvende til et nyt felt på et view.
- Det manglende udviklingsmiljø til Javascript-kodebasen betyder, at der ikke er nogen automatisk integration til Team Foundation Serveren, hvilket igen medfører at mange små opgaver som i vores optik burde blive udført automatisk p.t. udføres som manuelle arbejdsprocesser.
- Den lidt kryptiske navngivningskonvention og de meget skærmbillede/data-fokuserede SDD-dokumenter betyder at man er nødt til manuelt at krydsnavigere mellem kodebaserne og SDD-dokumentationen hver gang man skal arbejde med kodebasen

Det samlede indtryk af udviklingsmiljøet står i stærk kontrast til de sidste mange års trend indenfor integrerede udviklingsmiljøer, hvor der netop har været fokus på at samle så meget funktionalitet som muligt i automatiserbare processer og hvor navigation i kildekode og anvendelse af frameworks er blevet udstyret med stadig bedre værktøjsunderstøttelse.

En anden faktor som bidrager væsentligt til at forlænge/fordyre udviklingsprocessen er den ret lave grad af automatisk test coverage: Desto hurtigere en udvikler kan få svar på om dennes rettelser har ødelagt anden funktionalitet i systemet, desto hurtigere kan fejlen blive rettet. Dermed vil man dels undgå at andre skal bruge tid på at opdage, undersøge og rapportere fejlen, dels kan man minimere den tid som udvikleren benytter på at iterere sig frem til den korrekte løsning.

De udviklere vi har talt med betegnede selv sproget i SDD-beskrivelserne som grænsende til 'kryptografi'. De rapporterer som sådan at de bruger meget tid på at afkode betydningen af SDD'erne, samt at det i nogle tilfælde er nødvendigt at opspore tidligere ressourcer på projektet for at få besvaret deres tvivlsspørgsmål omkring detaljerne i en SDD.

Referencer

1. Kodemetrikker for C#-delen af POLSAG. Se det vedlagte dokument POLServicesCodeMetrics.xlsx.
2. Code Analysis-rapport for C#-delen af POLSAG. Se det vedlagte dokument POLServicesCodeAnalysis.FxCop.
3. Design Patterns: Elements of Reusable Object-Oriented Software (af Gamma, Helm, Johnson, Vlissides).
4. SOLID acronym: http://en.wikipedia.org/wiki/Solid_(object-oriented_design)
5. DRY acronym: http://en.wikipedia.org/wiki/Don%27t_repeat_yourself
6. POLA acronym: http://en.wikipedia.org/wiki/Principle_of_least_astonishment
7. Clean Code: A Handbook of Agile Software Craftsmanship (af Robert C. Martin)
8. Domain-Driven Design: Tackling Complexity in the Heart of Software (af Eric Evans)

Appendix A

Javascript-filer som ikke eksplicit er inkluderet i layout-filer

A0004.js

address_etage_domain.js

afgoerelse_afgoerende_domain.js

afgoerelse_dom.moedendeanklager_domain.js

anholdelse_ansvarlig_kreds_domain.js

anholdelse_ext.anbringelses_myndighed_domain.js

anholdelse_ext.indbragt_til_domain.js

boede_client.js

contact_myndighed.kode_domain.js

contact_party.name_ref_key_domain.js

customCaptia.js

customControl_overfoer.js

customControl_sagsdisposition.js

customFunctions_sigtsag.js

customMenus_genstand.js

customReport.js

customServerSjlink.js

custom_subscription.js

D0045_F.js

D0045_M.js

D0077_F.js

D0091_C.js

D0091_F.js

D0091_M.js

D0145_F.js

D0154_F.js

D0187_M.js

D0200_F.js

D0208_F.js

D0218_F.js

dcs_client.js

eftersoegning_ext.udfoerende_myndighed_domain.js

F0072.js

F0536.js

F0539.js

F0659_VisDatafoelgeseddel.js

faengsling_ext.anbringelses_myndighed_domain.js

file_action_ou_domain.js

file_officer_domain.js

forholdsafgoerelse_afgjortfor_domain.js

genstandskategorier_elabval_domain.js

genstand_koeretoej.nationalitettekst_domain.js

hierarchicalMenu.js

kontantdelsum.js

L0003.js

L0034_F.js

L0035_F.js

L0040_F.js

L0063_F.js

L0067_L0068.js

L0069_F.js

L0084_F.js

L0111_L0112_L0113_F.js

L0138_F.js

L0139_F.js

L0146_F.js

L0147_F.js

L0148_F.js

L0163_F.js

L0164_F.js

L0166_F.js

l0167_l0176_f.js

L0169_L0170_L0171_F.js

L0179_F.js

L0181_F.js

L0187_M.js

L0215.js

L0236_F.js

L0238_F.js

L0245_F.js

L0253_L0269_L0270_F.js

l0256.js

l0263_l0264_f.js

l0266_M.js

l0268_M.js

Lxxx_Afg_UdlKlip_F.js

patruljeresource_radioid_domain.js

personkategori_ident.nationalitet_landenavn_domain.js

record_action_ou_domain.js

record_dokument.journalnummer_domain.js

record_file_key_domain.js

S0007_D0124_F.js

S0018_M.js

S0023_F.js

S0025_F.js

S0026_M.js

sagsdisposition_patruljeenhed_key_domain.js

sagsdisposition_sagsdispositiontype_domain.js

signalement_beklaedning.sig_beklaedningfarve_domain.js

sigtelse_ext.gerningskode_domain.js

sigtelse_ext.udfoerende_myndighed_domain.js

soege_domains_foedested_domain.js

soege_domains_nationalitet_kode_domain.js

Appendix C Spørgsmål omkring POLSAG ClientServices

Dette appendix indeholder uddybende spørgsmål til ScanJour som opfølgning på afklaringsmødet 22/6. Besvarelsen af spørgsmålene er udført af Dennis Lauritzen (DPE), som ligeledes har indsat en række yderligere kommentarer. Al Dennis' tekst er angivet i kursiv.

Q: Kan det bekræftes at C++-drivers ikke implementerer logik som er specifik for web applikationen?

DPE: Ja det kan bekræftes. Drivers har kun adgang til data der gemmes i felterne på et register. Der implementeres forretningslogik der sikrer at data indeholder de korrekte data (både et minimum set, samt dataafhængigheder som defineret af RP). Jeg kan finde kommentarer der indeholder D0XXX – men dette er taget direkte fra SDD dokumenterne, og gør det derfor nemmere at finde SDD dokumentationen fra koden og omvendt.

Q: Kan det bekræftes at PL-SQL laget ikke implementerer logik som er specifik for web applikationen?

DPE: Jeg går ud fra at I ser bort fra PL-SQL der blot indlæser data (stamdata, fejlbeskeder, navne på skærmbilledernes labels).

Jeg har fundet 1 trigger som desværre ikke aflæser typen af genstand på andet end layout navnet. Dette er triggeren "SJTK_HIST_GENSTANDE_ROW"... denne har dog en default (if... elseif... else...) som blot sætter genstanden til en anden genstand. Det er dog muligt for en hvilken som helst applikation at udfylde layout med det nødvendige – selvom dette er ikke er så gennemskueligt i vedligeholdelsesøjemed, så dette vil kræve en kommentar (eller at vi retter op på triggeren – hvilket vil være det naturligt rigtige at gøre ☺)

Jeg har fundet i oprettelsen af nødidenter, at der i en kommentar nævnes "D0014" i triggeren "noedident_forslag_before_row", men dette er blot et felt til statistik, og kan ligeså godt udfyldes med teksten "iPhoneSmartyApp" såvel som teksten "D0014".

Jeg har ikke kunnet finde andet?

Q; Kan det bekræftes at nyudvikling skal anvende SOMASP til datalæsning og enten SOMASP eller ClientHandler.ashx til enkelt-register-opdateringer samt ClientHandler.ashx til fler-register opdateringer?

DPE: Alle skærmbilleder henter data via SOMASP, så det ville der ikke være noget mærkeligt i.

ClientHandler.ashx vil være en mulighed for de udstillede registre der er her, men der er stadig mulighed for at benytte de eksisterende BPM facader, eller at lave nye BPM facader mere tilegnet klienten eller klienttypen.

Q: Er det korrekt forstået at Javascript regerer på forretningskonfiguration i stamdata modulet.

DPE: Ja det er korrekt – det er jo kodet ud fra stamdatatyper og de tilhørende stamdataattributter og relationer hvordan de skal reagere. Det samme sker på serversiden, ligesom beskrevet i SDD'erne?

Q: Er det korrekt forstået (baseret på de vedhæftede eksempler) at BPM er tæt koblet til skærmbillederne og at den kode ikke kan genbruges.

DPE: Det er korrekt at BPM i første omgang er udviklet med henblik på at understøtte POLSAG web-

applikationen, men som jeg forsøgte at forklare i går, er der netop lavet en XML abstraktion så det er muligt for andre applikationer at tilgå de samme facader (som BPM jo egentlig er/skal være).

Q: Er det korrekt forstået at der er faktisk er forretningslogik(baseret på vedhæftede) i BPM koden

DPE: Det er korrekt at der er faktisk forretningslogik i BPM laget – dette skal selvfølgelig refaktoreres, da det ikke følger guidelines. Dvs. at BPM laget udelukkende skal bestå af facader der kalder ned i funktioner der kan genbruges på tværs af applikationen og andre eksterne systemer... ligesom vi gør med f.eks. webservices og agenter.

Vi har ledt efter egentlige arkitektur guidelines der sikrer den opsplitning som Bo præsenterede i går men har ikke kunne finde dette. Kan I pege os i den rigtige retning?

DPE: Omkring BPM, så fremgår det af vejledningen til Business Process Handlers API'et. Dokumentet ligger på DVD'en under "\POLSAG Guidelines and documentation").

Omkring integrationen til eksterne systemer er det beskrevet i "Arkitekturbeskrivelse af POLSAG integration til randsystemer V0005", der ligger på DVD'en i mappen "\Overordnet POLSAG arkitektur\Integrationer", samt ren tilføjende beskrivelse i dokumentet "Generelle kravsbesvarelser for POLSAG løsningsarkitekt V0005", der ligger på DVD'en i mappen "\Overordnet POLSAG arkitektur".

Bare for at vi er sikre på at vi forstår hvordan ændringer implementeres. Kan I ikke på kort bullet form angive hvordan følgende udvidelser implementeres, men angivelse af hvad der er direkte genbrug fra POLSAG.

- Sagsoprettelse uden for POLSAG, e.g. fra smart device

DPE: Kald Sag.Anmeldelse.Opret Webservice (den findes jo allerede?)

- Indgivelse af anmeldelse fra en Borger

DPE: Kald Sag.Anmeldelse.Opret Webservice (den findes jo allerede?)

- Outsourcing af delopgaver e.g. alarmcentralen.

DPE: Der vil formentlig være tale om en anderledes arbejdsgang og som vi sagde i går, så vil det formentlig give mening at implementere nye facader der dog stadig kalder ned i eksisterende funktioner – med mindre RP mener at der skal laves nye funktioner der understøtter forretningsgangen på en anden måde – dog uden at bryde med dataintegriteten for RP's begrebsmodel.

- Fuld digitalisering af kommunikationen med domstolene (dvs. de processer hvor der i dag modtages papir materiale modtages fuldt digitalt, inklusiv kendelser, retsmøder etc)

DPE: Her ville jeg genbruge datafølgeløsningens funktioner, udvide facaden (eller lave en ny facade hvis det ønskes at begge versioner skal kunne sameksistere) med et ekstra funktionskald der tager XML'en som input, og der udvider indholdet i XML'en med det ønskede, hive de ekstra dokumenter ud og ændre i distributionen af disse (det er utopi at tro at man kan sende flere 100 MB på e-mail med den eksisterende infrastruktur ☺)... distributionen skal selvfølgelig aftales sammen med RP og CSC.

Nedenfor følger an analyse af hvordan kode-basen i client services er struktureret. Hvis der er nogle misforståelser i analysen bedes I også kommentere disse.

Analyse

ClientServices består teknisk set af en udstilling af følgende funktionalitet i et browser-konsumerbart format:

- Udstilling af de eksterne systemer over HTTP, så disse kan kaldes fra webapplikationens Javascript-libraries
- Udstilling af Business Process Handler-funktionalitet[19] over HTTP (BPHandler.ashx)
- Udstilling af fler-register Create/Update/Delete operationer over HTTP (ClientHandler.ashx)

Ingen af de ovennævnte interfaces er typestærke.

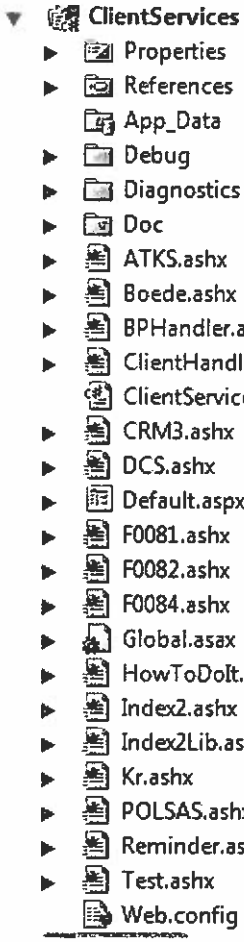
Alle er bygget til at fungere på en HTTP-klient som post'er en HTML FORM med felter, som skal have navne der passer med de skærmbilleder som er i POLSAG-webapplikationen.

DPE: Som jeg forsøgte at forklare i går er det er korrekt at der fra klienten mange steder blot sender HTML FORM objektet – dvs. det "serialiseres" til blot at være en XML repræsentation af HTML FORM objektet. Denne XML skal selvfølgelig overholde en struktur og det er korrekt at navnene er dem der passer i skærmbillederne, men det er en ret simpel struktur (<Element id="navn på felt">Tekstbaseret dataindhold</Element>) og XML er netop den valgte abstraktion for at gøre det muligt for nye klienter at tilgå samme funktionalitet. I Jeres konklusion var det udtrykt at dette ikke er egnet til digitalisering ved f.eks. mobile applikationer, men da det blot er XML er det absolut ikke uegnet til at bruge for f.eks. mobile applikationer. Jeg vil medgive at hvis det skal udvikles af eksterne parter vil det kræve noget specifik dokumentation, der nemt kan autogenereres ud fra skærmbillederne – netop fordi skærmbillederne også blot er XML.

Udstilling af de eksterne systemer over HTTP

ClientServices indeholder p.t. følgende HTTP-protokol endpoints (BPHandler og ClientHandler er behandlet i de efterfølgende afsnit):

[19] Business Process Handlers er det som "populært" benævnes Business Process Manager (BPM).



Udstilling af Business Process Handler-funktionalitet over HTTP (BPHandler.ashx)

Implementeringerne af Business Process Handlers anvender primært det typestærke billede af register-modellen til at manipulere data.

Der er 531 steder i Business Process Handler frameworket, hvor man er direkte afhængig af at HTTP POST *(DPE: Se tidligere kommentar omkring "serialiseringen til XML format)* fra klienten respekterer POLSAG webapplikationens HTML-struktur. Dette er fordelt på ca. 110 Business Process Handler-implementeringer (ud af i alt ca 130 business process handlers). Disse tal er fremkommet ved at undersøge, hvor der anvendes en XPathNavigator *(DPE: Som netop blot er et XML API fra Microsoft til at navigere i XML?)* til at trække data ud af den indkommende HTML FORM *(DPE: Nu XML)* og anvende de ekstraherede data. De eksisterende Business Process Handlers kan af denne grund ikke forventes genanvendt til andre klienter end POLSAG web applikationen.

DPE: Her er jeg ikke enig jfr. tidligere kommentar omkring "serialiseringen" til XML format og muligheden for at autogenerere dokumentation.

Den generelle tilgang til BPM er at man som klient kalder en facade (ikke typestærk), der kalder videre ned i funktioner (typestærke). Med denne opdeling har man også muligheden for nemt at lave flere facader (også

typestærke til klienter der understøtter dette), men genbruge hele den underliggende funktionalitet, blot man følger signaturen til funktionerne.

Og som vi sagde i går, så vil der være eksempler på at det ikke er udført korrekt og stadig skal refaktoreres visse steder (oprydning).

Udstilling af fler-register Create/Update/Delete operationer over HTTP (ClientHandler.ashx)

Implementeringen af ClientHandler.ashx anvender standard-Captias streng-baserede API til at manipulere data, samt en POLSAG-specifik formatterings-syntax for, hvordan man skal addressere de registre og felter i disse som skal manipuleres.

Når man anvender ClientHandler.ashx API'et er der ingen anvendelse af de server-side C#-valideringer, som man får med gennem Business Process Manager interfacet *(DPE: Der er de C# valideringer der er implementeret på de udstillede klasser igennem dette API).* C++-driver valideringer kører stadig fordi databasen tilgås via standard SOM API, men man skal implementere al forretnings-proces- og data-validering client side, hvis ikke funktionaliteten i C++-driveren er dækkende for den konkrete use case *(Eller benytte BPM som den meste funktionalitet netop blev implementeret i som reaktion på egen anderkendelse og dette berkræftet i Triforks review).*

Data skal stadig hentes via SOMASP, eftersom ClientHandler.ashx ikke udstiller Read-access til SOM registre. Det vil sige at nye klienter på POLSAG de facto er bundet til en browser (fordi dataadgang er over HTTP) *(DPE: Det er da ikke nødvendigt at være browser for at kommunikere over HTTP? Langt hovedparten af webserviceløsninger benytter da SOAP over HTTP? Jeg er godt klar over at det ikke er SOAP der benyttes her, men HTTP er jo blot en transportprotokol som benyttes af flere overliggende standard dataprotokoller – XML (som vi benytter, om end lidt proprietært i formatet) kan også transporteres via HTTP.)* og man skal direkte anvende standard Captias SOMASP ESDH-rettede API til udviklingen *(Som nævnt tidligere vil dette være naturligt – alle skærmbilleder benytter SOMASP til at hente data til især renderingen – endvidere giver det muligheden for at angive tekstbaserede skabeloner (enten simple skabeloner deklareret i URL'en som en streng, hvilket er yderst fleksibelt for klienten, eller både simple og komplicerede skabeloner gemt som en tekst-/HTML-/XML fil, hvilket selvfølgelig er bundet til det der er implementeret i filen). Skabeloner igennem SOMASP benytter mergehandler til at hente data, som vi snakkede om den aller første dag for reviewet.*

POLSAG bidrager således med en teknisk løsning på opdatering af flere registre samtidig, samt de C++-drivers som knytter sig til de enkelte registre *(alle registre der kommer i spil vil aktivere drivers i alle tilgåelser af registrene).*

Der er 13 funktioner i Common Funktions-bibliotek, hvor HTML *(XML?)*-strukturen af input data er i sving. *(DPE: Jeg har ikke kigget nærmere på dette, men det lyder som om at der også her er behov for lidt oprydning!)*

Eksempel fra en Business Process Handler

Nedenfor er oplistet en del af den ca 1250 linjer lange Business Process Handler F0015Omjournaliser_BusinessProcessHandler implementering. *(DPE: Jeg er enig i at dette er et grelt eksempel der trænger til refaktorering – både som opdeling i flere funktioner (1250 linjer er bare for mange), samt adskillelse mellem BPM facade (ikke typestærk) og funktion (typestærk). Så ja der er stadig behov for oprydning, men dette eksempel er heller ikke i tråd med hverken god kodeskik eller guidelines. Det*

lagde vi heller ikke skjul på i går at der stadig er behov for).

Implementeringen er helt afhængig af at HTML input-strukturen/feltnavne/attributnavne er identisk med POLSAG webapplikationens skærmbilledet/skærmbillederne, som kalder denne Business Process Handler. *(DPE: Jeg håber det er nok at referere til det jeg har skrevet tidligere omkring "serialisering" til XML og kommentarer omkring dette).*

Bemærk i øvrigt at uddraget herunder kun udgør en mindre del af den omtalte Business Process Handler.