Sikker software

Anbefalinger for egenudviklet klinisk software

på danske stråleterapiafdelinger

Marts 2023

Indholdsfortegnelse

Introduktion

Målgruppe

Dette dokument henvender sig dels til ledere af stråleterapiafdelinger og dels til fagpersoner, der producerer software til brug på stråleterapiafdelinger.

Formål

Formålet med dette dokument er at angive en række anbefalinger til håndtering, dokumentation, kvalitetssikring og deling af egenudviklet software, der anvendes klinisk på danske stråleterapiafdelinger.

Dokumentet kommer ikke med rigide krav; men derimod med generelle anbefalinger. Det endelige ansvar for håndtering af egenudviklet software til klinisk brug ligger altid hos de individuelle afdelingsledelser.

Dokumentet er forsøgt udfærdiget i generelle termer, som kan anvendes til alle typer af software, uanset det bagvedliggende programmeringssprog eller den platform, som softwaren udvikles eller afvikles på. Ydermere er dokumentet forsøgt skrevet i et sprog, der kan forstås af personer, der ikke selv har erfaringer med softwareudvikling. Her er især tænkt på ansvarshavende ledere af kliniske afdelinger, hvor der udvikles software. En ordliste kan findes i Appendiks A.

Baggrund for arbejdet

Dette dokument udspringer af arbejde med de såkaldte ESAPI-scripts, der er egenudviklet programkode, som kan interagere med det kommercielle dosisplanlægningssystem Eclipse fra Varian Medical Systems. ESAPI-scripts, der er udbredt på 6 af de 8 danske stråleterapier, gør det muligt at automatisere en række af de arbejdsopgaver, man har i forbindelse med både forskning og klinisk arbejde med strålebehandlingsplaner. Tilsvarende scripting-muligheder findes på de øvrige stråleterapier.

I 2022 blev der startet et nationalt netværk af fysikere, der interesserer sig for ESAPI-scripting. Ofte arbejdes der på at løse præcis de samme opgaver, og hvis én afdeling udvikler et script, der løser en given opgave, vil det være oplagt at dele dette script med andre afdelinger.

Dette efterlader dog en række spørgsmål om, hvordan man gør dette i praksis; både brugsmæssigt og indenfor lovens rammer. Hvem har det endelige ansvar for brug af softwaren? Hvordan håndterer man, hvis nogen opdager en fejl? Hvordan sikrer man, at alle brugere modtager relevante opdateringer?

Det er desuden naturligt, at jobskifte eller længere fravær (sygdom, barsel m.m.) i en klinik kan føre til, at de etablerede software-eksperter pludseligt ikke længere kan varetage programmeringsopgaver. I disse tilfælde bør den eksisterende kode være så nem som muligt at forstå for andre fagpersoner, hvorfor man bør have retningslinjer for, hvordan brugen af softwaren dokumenteres.

På ESAPI-netværkets første møde nedsattes følgende arbejdsgruppe til at udfærdige dette dokument:

- Laura Kaplan (Næstved Sygehus)
- Kasper Lind Laursen (Aalborg Universitetshospital)
- Lars Præstegaard (Aarhus Universitetshospital)
- Thomas Ravkilde (Aarhus Universitetshospital)
- Klaus Seiersen (Dansk Center for Partikelterapi)

Jura

Oversigt over gældende lovgivning

I forhold til brug af software kommer man ikke udenom de juridiske forpligtelser, der gælder i sundhedsvæsenet og på forskningsinstitutioner. Kort fortalt er man forpligtet til:

- at beskytte personhenførbare oplysninger (Persondataforordningen, GDPR),
- at leve op til lovgivning og retningslinjer ift. udvikling og drift af klinisk benyttet software (MDR-forordningen),
- at beskytte og indberette eventuelle opfindelser (Lov om opfindelser ved offentlige forskningsinstitutioner).

Dette beskrives i større detalje i det følgende.

Persondataforordningen (GDPR)

EUs "*General Data Protection Regulation*" (GDPR)[1] - på dansk Persondataforordningen - er et konstant opmærksomhedspunkt i sundhedsvæsenet. Når det gælder lokalt udviklet software, er der primært to ting, man skal have for øje:

1. Software må ikke give tredjepart adgang til personhenførbare data. Eksempelvis
    a. i kildekode (f.eks. test cases),
    b. i dokumentation eller
    c. præsenteret via ikke-beskyttet hjemmeside eller lign. service.
2. Det skal være muligt at se, hvem der har tilgået et vilkårligt individs persondata. Dvs. der skal være logning af al adgang til personhenførbare oplysninger.

I Danmark håndhæves Persondataforordningen (GDPR) af Datatilsynet, og ethvert brud på persondatasikkerheden skal altid indberettes hertil.

MDR-forordningen

EUs "*Medical Device Regulation*" (MDR)[2] trådte i kraft i 2021 som en præcisering og opstramning af tidligere retningslinjer og krav til medicinsk udstyr. I MDR-forordningen opfattes software til klinisk brug som medicinsk udstyr og betegnes Medical Device Software (MDSW). I Danmark håndhæves MDR af Lægemiddelstyrelsen.

I det følgende præsenteres nogle få særlige opmærksomhedspunkter:

Hvornår er software et medicinsk udstyr?

Der skelnes mellem forskellige overordnede typer:

[1] Se GDPR-forordningens fulde tekst her: https://eur-lex.europa.eu/legal-content/DA/TXT/?uri=CELEX%3A32016R0679&qid=1667900240606.

[2] Se MDR-forordningens fulde tekst her: https://www.medical-device-regulation.eu/download-mdr.

- Software der i sig selv har et medicinsk formål.
- Software der styrer eller påvirker brugen af medicinsk udstyr, men som i sig selv ikke har et medicinsk formål.
- En kombination af de 2 ovenstående.

Software defineres i MDR-forordningen som et sæt af instruktioner, der behandler input data og skaber output data. Det betyder at selv et regneark også kan være MDSW afhængigt af formålet. Ligeledes kan en lille ændring i software gøre det til et medical device. For eksempel vil en sortering af data som kan bruges til beslutningsstøtte til behandling gøre softwaren til MDSW.

Der henvises til "*VEJ nr 9366 af 21/05/2021: Vejledning til fabrikanter om software og apps for medicinsk udstyr og medicinsk udstyr til in vitro-diagnostik*"[3], MDR-forordningen og MDCG infographic[4] for yderligere information.

Brug af software som medicinsk udstyr

Software må kun benyttes som medicinsk udstyr, hvis

- det har opnået CE-mærkning via eksternt audit fra et bemyndiget organ ("notified body"),
- det indgår i en godkendt klinisk forskningsprotokol via en Investigational Device Exemption (IDE) eller
- det lever op til kravene for internt udviklet medicinsk udstyr beskrevet i MDR Artikel 5.5.

MDR Artikel 5.5 beskriver en undtagelsesregel for internt udviklet medicinsk udstyr, hvor CE mærkning ikke er påkrævet, såfremt en række krav er opfyldt:

> *a) Udstyret er ikke overført til en anden retlig enhed (i Danmark tolkes Regionerne som retlige enheder)*
> *b) Fremstilling og anvendelse er grundigt beskrevet i et passende kvalitetssikringssystem (Quality Management System; QMS)*
> *c) Der er en nedskrevet begrundelse for, at behovet ikke kan opfyldes af kommercielt udstyr*
> *d) Efter anmodning skal oplysninger om anvendelse kunne udleveres*
> *e) Der skal offentliggøres en erklæring om brug af internt udviklet medicinsk udstyr*
> *f) Der er udarbejdet passende dokumentation*
> *g) Fremstillingen af softwaren er i overensstemmelse med dokumentationen*
> *h) Der er beskrevet en procedure for erkendelse og foretagelse af nødvendige korrigerende handlinger*

Det bør her nævnes, at et QMS f.eks. indebærer et system for versionskontrol (se Versionskontrol), men ikke er begrænset til dette. Der henvises til MDR Artikel 5.5 for en fyldestgørende beskrivelse af kravene, samt MDR Annex I for beskrivelse af de generelle krav til sikkerhed og ydeevne af medicinsk udstyr.

Der henvises desuden igen til "*VEJ nr 9366 af 21/05/2021: Vejledning til fabrikanter om software og apps for medicinsk udstyr og medicinsk udstyr til in vitro-diagnostik*", og endvidere til:

- "*ISO 13485 Medical devices - Quality management systems*"

[3] Se vejledningens fulde tekst her: https://www.retsinformation.dk/eli/retsinfo/2021/9366.

[4] EUs Medical Device Coordination Group (MDCG) har lavet en grafik, der kan hjælpe til at afklare, om din software er et Medical Device. Se: https://health.ec.europa.eu/system/files/2021-03/md_mdcg_2021_mdsw_en_0.pdf.

- *"ISO 82304 Health Software"*
- *"IEC 62304 Medical device software"*
- *"ISO 14971 Medical devices - Application of risk management to medical devices"*.
- *"IEC 61217:2011 - Radiotherapy equipment - Coordinates, movements and scales"*

Vær særligt opmærksom på at din region kan have egne retningslinjer for hvordan MDR-forordningen skal håndteres.

Anbefaling

Det anbefales, at alle afdelinger orienterer sig i regionsspecifikke retningslinjer angående håndtering af MDR-forordningen.

Klassificering af medicinsk udstyr

Medicinsk udstyr klassificeres som klasse I, IIa, IIb eller III. Krav til fremstilling, dokumentation, test mv. skærpes med højere klassificering.

Med MDR er der indført en særlig regel for software (MDR Annex VIII Regel 11), som siger:

> *Software, der er beregnet til at tilvejebringe oplysninger, der anvendes til at træffe beslutninger med diagnostiske eller terapeutiske formål, er klassificeret som klasse IIa, medmindre sådanne beslutninger har en virkning, der kan forårsage:*
> *— dødsfald eller en uoprettelig forringelse af en persons sundhedstilstand, idet det i så fald henhører under klasse III, eller*
> *— en alvorlig forringelse af en persons sundhedstilstand eller et kirurgisk indgreb, idet det i så fald er klassificeret som klasse IIb.*
> *Software, der er beregnet til at monitorere fysiologiske processer, er klassificeret som klasse IIa, medmindre det er beregnet til at monitorere vitale fysiologiske parametre, hvor variationer af disse parametre har en sådan karakter, at de kan udgøre en umiddelbar fare for patienten, idet det i så fald er klassificeret som klasse IIb.*
> *Alt andet software er klassificeret som klasse I.*

Se også *"VEJ nr 9366 af 21/05/2021": Vejledning til fabrikanter om software og apps for medicinsk udstyr og medicinsk udstyr til in vitro-diagnostik*".

Lov om opfindelser

En ofte overset forpligtelse er pligten til at beskytte og indberette eventuelle opfindelser til sin arbejdsgiver. *"Lov om opfindelser ved offentlige forskningsinstitutioner"*[5] beskriver denne pligt, hvis man arbejder ved en vidensinstitution. Det gælder forskning, men også hvis man har fået en innovativ idé i det daglige kliniske arbejde.

Der kan være institutionsafhængige regler for rettigheder til opfindelser. Rettighederne ligger som udgangspunkt hos arbejdspladsen, og softwaren må derfor ikke nødvendigvis deles uden videre - heller ikke til inspiration eller uddannelsesformål jf. afsnittet om Deling af egenudviklet software.

[5] Se lovens fulde tekst her: https://www.retsinformation.dk/eli/lta/2009/210.

Der findes som regel lokale retningslinjer for indberettelse af mulige opfindelser. Universiteterne har typisk et Technology Transfer Office (TTO), som man skal kontakte. Hospitaler kan både have egne enheder eller henvise til det tilknyttede universitets TTO.

Generelt opfordres man til at kontakte sit lokale TTO, hvis man er i tvivl om, hvorvidt en idé har opfinderhøjde.

Deling af egenudviklet software

Såfremt noget kildekode ikke har opfinderhøjde (se afsnittet om Lov om opfindelser), må det deles med andre. Hvis den egenudviklede software er et medicinsk udstyr (se afsnittet om MDR-forordningen), må det dog ikke benyttes af en modtagende afdeling i en anden region til andet end inspiration.

Med ordet "inspiration" menes at "sådan kan opgaven f.eks. løses", hvilket kan relatere sig til softwarens funktionalitet, udseende af brugerinterface, programmeringsløsninger, input/output-formater og andet. Populært sagt må man skrive sin egen fortolkning af løsninger observeret i anden kildekode. Afgrænsede kodestumper ("snippets") kan deles som inspiration uden videre, så længe de overholder GDPR og "Lov om opfindelser ved offentlige forskningsinstitutioner", på linje med at man har fået et svar på et online software-forum som Stack Overflow, ESAPI SubReddit eller lignende.

Ved deling af software bør man tilknytte en ansvarsfraskrivelse, hvilket også giver mulighed for at dele kildekode som Open Source Software. Hvis der opdages fejl i softwaren efter deling, er man derfor ikke forpligtet til at informere modtagerafdelingerne, men det anbefales som almen god skik at gøre det. Bemærk at for egenudviklet MDSW til internt brug er man ifølge MDR Artikel 5.5 forpligtet til at foretage intern registrering af fejl.

Anbefaling

Det anbefales, at al deling af software tilknyttes en klar og tydelig ansvarsfraskrivelse, f.eks. en MIT-licens.

Et eksempel på en sådan ansvarsfraskrivelse kan være den udbredte MIT-licens, der ikke er restriktiv:

Eksempel på softwarelicens ("MIT-licensen"):

Copyright <YEAR> <COPYRIGHT HOLDER>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR

PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Deling internt i en dansk region

Både kompileret kode og kildekode må deles, modtages og benyttes, så længe det opfylder kravene i MDR Artikel 5.5 punkt b-h (punkt a er opfyldt). GDPR og "Lov om opfindelser ved offentlige forskningsinstitutioner" skal overholdes (ingen personoplysninger eller nyhedsskadelse).

Deling indenfor Danmark, men mellem regioner

Både kompileret kode og kildekode må deles til inspiration, så længe GDPR og "Lov om opfindelser ved offentlige forskningsinstitutioner" overholdes (ingen personoplysninger eller nyhedsskadelse). Husk ansvarsfraskrivelse. Kontakt evt. lokalt TTO eller lignende inden deling, så sagen kan vurderes juridisk.

Modtaget kildekode må benyttes til inspiration til egen løsning, som selvstændigt skal opfylde MDR Artikel 5.5, hvis der er tale om MDSW.

Deling udenfor Danmark, men indenfor EU

Som Deling indenfor Danmark, men mellem regioner. Dog bør der være opmærksomhed på evt. selvstændig lovgivning i det modtagende land.

Deling mellem Danmark og et land udenfor EU

Som Deling udenfor Danmark, men indenfor EU. Dog bør der være *særlig opmærksomhed* på gældende lovgivning i det modtagende land.

Risikovurdering

Metode til risikovurdering

Når man arbejder med klinisk software, er det afgørende, at man forud for ibrugtagning foretager en risikovurdering, hvor man identificerer mulige negative hændelser (risici) ved brug af softwaren samt vurderer omfanget af den potentielle skade af disse risici.

Risikovurderingen handler altså om at vurdere sandsynligheden for at en given hændelse forekommer, samt hvad konsekvensen er, *hvis* den forekommer.

Identificering af risici

En risikovurdering starter med, at man identificerer risici ved brug af en given software. Denne proces kan for eksempel foregå ved et brainstorming-møde med andre softwareudviklere og/eller kommende brugere af softwaren. Som minimum bør der dog altid være mindst én anden person end udvikleren, der vurderer mulige hændelser.

Vurdering af de enkelte risici

Herefter risikovurderes de enkelte hændelser. En enkel og udbredt metode til at lave denne vurdering går ud på at score sandsynligheden for, at en hændelse forekommer, samt at score alvorligheden af hændelsen:

Eksempel på risikovurdering af en potentiel hændelse:

Sandsynligheden for at en hændelse forekommer scores på en skala fra 1-5, hvor 5 er "mest sandsynlig". Tilsvarende scores konsekvensen af hændelsen fra 1–5, hvor 5 er "mest alvorlig".

Herefter udregnes et risikotal som: ***Risikotal = Sandsynlighed x Konsekvens***

Risikovurderingen, f.eks. kvantificeret ud fra ovenstående risikotal, kan nu anvendes til at sortere de forskellige risici og vurdere, hvor omfattende arbejde der bør ligge i kvalitetssikring af softwaren samt overvejelser om, hvilke nødprocedurer der bør være etableret og eventuelt afprøvet før ibrugtagning.

Der vil til enhver tid være tale om en vurdering, og man bør således overveje at foretage vurderingen i en gruppe med andre softwareudviklere og/eller kommende brugere af softwaren. Som minimum bør der dog altid være mindst én anden person end udvikleren, der vurderer risikotal.

Anbefaling

Det anbefales at afdelingen indfører arbejdsgange til systematisk at *identificere* og *risikovurdere* mulige negative hændelser forud for ibrugtagning af software. Denne procedure bør som minimum involvere én person, der ikke selv har udviklet softwaren. For potentielle negative hændelser med høj risiko, bør man

vurdere mulige handleplaner til at reducere sandsynligheden for forekomst og/eller konsekvensen af hændelsen.

Det er vigtigt at være klar over, at en risikovurdering for den samlede softwarefunktionalitet dels bygger på den grundlæggende arkitektur og dels på funktionaliteten af de enkelte delelementer af koden. Det vil således give mening at risikovurdere dele af softwaren separat. Den del af softwaren, der blot bruges til at lave et grafisk layout kan måske ikke i sig selv forårsage behandlingsfejl, hvorimod de dele af koden, der læser og foretager beregninger på bestemte data, kan have høj risiko for at forårsage en fejl.

Ved kvalitetssikring af software bør man således identificere de mest kritiske kodedele og være særlig omhyggelig med at checke disse dele (og således mindske sandsynligheden for en hændelse). Metoder til kvalitetssikring af software findes i afsnittet Værktøjer og metoder til sikring af høj kvalitet.

Anbefaling

Det anbefales, at man foretager risikoanalyse dels på den samlede brug af softwaren, og dels på softwarens delfunktionaliteter. Dette vil tydeliggøre, hvor i kildekoden det er særlig vigtigt at kvalitetssikre funktionaliteten og dermed minimere sandsynligheden for en potentiel negativ hændelse.

Se Appendiks 2: Eksempel på model for risikovurdering for et uddybende eksempel på risikovurdering og hvordan man kan vælge at reagere på de individuelle risikotal.

Risikovurdering af klinisk software

Ved risikovurdering af klinisk software bør man blandt andet inkludere følgende overvejelser:

Sandsynlighed

- Hvor kompleks er koden? Jo mere kompleks koden er, des større er sandsynligheden for fejl.
    - Er det en kort og overskuelig kode, eller er der mange linjer i koden?
    - Er der tale om simple, standardiserede kommandoer, eller bygger koden på omfattende algoritmer med mange beregninger?
    - Er koden let at forstå for andre, der skal checke den? Er den velkommenteret og -dokumenteret (jf. afsnittet Programmeringsstil (code of practice))?
- Bygger softwaren på kode skrevet af andre, hvor funktionaliteten måske ikke er fuldt forstået (dette betegnes af udviklere som "*software of unknown pedigree*", SOUP)?
- Har softwaren kun læseadgang eller både læse/skrive-adgang til patientdata? Hvis koden også har skriveadgang, bliver det samtidigt muligt at korrumpere patientdata.
- Er softwaren følsom overfor opgraderinger? Er det muligt, at softwaren holder op med at virke, hvis andre systemer opdateres?
- Testniveau: Hvor omfattende er afdelingens praksis omkring afprøvning af softwaren før ibrugtagning - er kritiske funktioner i koden testet grundigt ifht. alle tænkelige input/output scenarier?
- …

Konsekvens

- Hvad er der fare for?
    1. Fejlbehandling af én eller flere patienter?
    2. Midlertidigt stop eller nedsat kapacitet af patientbehandlinger?
    3. Korruption af data?
    4. Unødvendigt ressourceforbrug?
- Foretages der kliniske beslutninger på baggrund af output fra software (ofte MDSW) eller er der kun tale om software, der gør nogle arbejdsgange nemmere/hurtigere (ikke nødvendigvis MDSW)?
- Har softwaren skrive-adgang til patientdata, således at det er muligt at korrumpere patientdata? Hvad er konsekvensen af dette? Er der systemer, der kontrollerer for korrupte data, så fejlen opdages hurtigt? Kan patientdata reddes via backupsystemer?
- Har softwaren adgang til én patient eller flere patienter? Hvis softwaren har skriveadgang til mange patienter, kan konsekvenserne være ganske omfattende.
- Kan softwaren belaste det kliniske system, således at andre arbejdsgange påvirkes?
- …

Eksempel på model for overordnet risiko for software

Herunder vises et eksempel på en simpel model, der har været anvendt på en dansk stråleterapiafdeling:

Sikkerhedsrisiko	
Lav	*Kun mulighed for at åbne en enkelt patient / ingen skrive-adgang: Den største risiko er at softwaren har et ressourcekrævende loop der belaster performance af det kliniske system.*
Lav-medium	*Mulighed for at åbne flere patienter / ingen skrive-adgang. Den største risiko er at softwaren har et ressourcekrævende loop (evt. et loop med mange patienter) der belaster performance af det kliniske system.*
Medium	*Kun mulighed for at åbne en enkelt patient / skrive-adgang. Den største risiko er at softwaren sletter eller ødelægger data for en enkelt klinisk patient.*
Høj	*Mulighed for at åbne en flere patienter / skrive-adgang: Den største risiko er, at softwaren ødelægger data for mange kliniske patienter.*

Bemærk at modellen i sin beskrevne form fokuserer på ødelæggelse af data, men ikke kliniske beslutninger baseret på softwaren.

Bemærk også at denne model har været anvendt for den samlede software. Jævnfør vores tidligere anbefalinger bør dette suppleres med risikovurdering af kritiske underdele af softwaren.

Eksempler på identificerede risici for klinisk software

Konsekvensen af en negativ hændelse i forbindelse med brug af software på en hospitalsafdeling kan variere ganske betydeligt i omfang. Her er tre eksempler:

- **Eksempel 1:** Nogle typer scripts er designet til at gøre en bestemt arbejdsgang en lille smule lettere, for eksempel ved at spare nogle klik på musen. Hvis scriptet fejler, vil man straks opdage, at arbejdsgangen ikke er udført af scriptet. I dette tilfælde vil den største konsekvens ved en fejl derfor

blot være, at man skal gøre "som man plejer" med nogle ekstra klik og et ubetydeligt højere tidsforbrug.

Her er der ikke tale om en målbar konsekvens, og der vil ikke være en påvirkning af afdelingens drift eller en patients sikkerhed.

- **Eksempel 2:** Visse typer af scripts kan aktivt manipulere en database med patientdata. Her kan konsekvensen være, at der utilsigtet introduceres fejl i en patients behandlingsdata, hvilket igen potentielt kan medføre en fejl i behandlingen af denne patient. Det er også muligt, at man korrumperer hele databasen, og således ødelægger mange patienters data på samme tid. Der kan dermed være tale om en ganske alvorlig fejl, som kan forårsage alvorlig skade på mange patienter.

  I forhold til ødelæggelse af patientdata, afhænger konsekvensen dog af lokale datasikkerhedsforhold: Hvis fejlen opdages hurtigt på grund af afdelingens kvalitetssikringsrutiner, og hvis databasen nemt kan tilbageføres via en backupløsning, behøver hændelsen ikke at have nogen nævneværdig konsekvens for patienter[6]. Hvis fejlen derimod overses i længere tid, og/eller hvis det er nødvendigt med en ressourcekrævende indsats for at genskabe databasen - måske endda med pausering af driften nogle dage - vil der være tale om ganske stor konsekvens.

- **Eksempel 3:** Det er ved at blive udbredt i Danmark, at software trækker oplysninger ud af en patientdatabase, hvorefter software foretager en beregning, der har konsekvenser for denne patients fremtidige behandling. Et eksempel på dette kan være dét script, der i dag anvendes klinisk på afdelinger til at trække doser ud af to strålebehandlingsplaner for at vurdere, om en patient skal modtage foton- eller protonstrålebehandling. Her er den potentielle konsekvens af en fejl i softwaren, at en eller flere patienter modtager en forkert type behandling.

  Konsekvensen kan altså være egentlige behandlingsfejl, der ikke kan gøres om. Samtidig afhænger alvoren af en fejl af om der er tale om en fejl for én patient eller en systematisk fejl for mange patienter over en længere periode.

[6] I praksis vil en tilbageførsel af en backup dog ofte være en ganske omstændelig proces med potentiel nedetid til følge.

Værktøjer og metoder til sikring af høj kvalitet

Programmeringsstil (code of practice)

Programmeringsstil vil altid variere mellem personer. Det kan dog være en fordel at stræbe mod en vis grad af lighed, da det vil lette samarbejdet omkring et projekt og facilitere vedligeholdelse og deling af kode. Desuden er det et krav beskrevet i MDR-forordningen, at ethvert selvstændigt stykke software anvendt som medicinsk udstyr indeholder en beskrivelse af kodestil (se afsnittet om MDR-forordningen).

Samtidig vil en etableret "code of practice" være til stor gavn for nybegyndere indenfor programmering, som gerne vil beskæftige sig mere med softwareudvikling.

I den bredeste forstand bør en sådan code-of-practice gå ud på, at man altid skriver sin kode sådan, at den kan læses og forstås af en klinisk fysiker med basale programmeringskendskaber. Der er en række faktorer, der er vigtige at overveje, for at gøre et stykke kode nemmere at læse for andre:

- Navngivning af kodeelementer
- Kommentering
- Strukturering indenfor den enkelte fil (inkl. kompleksitet af koden)
- Strukturering af alle filerne i et større projekt

Disse punkter diskuteres i større detalje i Appendiks 3: Eksempler på god programmeringsstil.

I det følgende gives nogle korte, generelle anbefalinger. Det er vores hensigt at give programmører et godt udgangspunkt for at gøre deres kode mere overskuelig og letlæselig. For en dybere gennemgang af god programmeringsskik henviser vi til relevante kurser og faglitteratur.

Anbefalinger

Det anbefales, at man på afdelingen aftaler rammer for programmmeringsstilen, så der sikres et ensartet format af kode på tværs af udviklere.

Navngivning af kodeelementer: Brug konsistent og ensartet navngivning. Navne bør afspejle elementernes hensigt og funktion. Navne bør ikke være unødvendigt lange.

Kommentering: Brug kommentarer til at forklare kodestykker, hvis funktion og hensigt ikke er umiddelbart indlysende. Kommentér hellere lidt for meget end lidt for lidt. Undgå lokalt slang og indforståede betegnelser.

Strukturering indenfor en fil: Følg sprog-specifikke konventioner og vær konsistent. Brug tabulatorer, mellemrum og linjeskift. Undgå unødvendig kompleks syntax. Undgå for lange klasser/metoder/osv.

Strukturering af et projekt: Undgå for lange filer. Navngiv filer hensigtsmæssigt. Lav evt. et klassediagram eller anden oversigt for store programmer. Følg evt. populære designprincipper (SOLID, MVVM og lignende).

Versionskontrol

Alle idriftsatte versioner af software bør være nemt tilgængelige, inkl. den underliggende kildekode. Dette kræver dels en veldefineret og nem adgang til filerne og dels en fornuftig versionskontrol. Versionskontrol er en forudsætning for at kunne leve op til de juridiske krav ifm. lokalt udviklet software i klinisk brug (se afsnittet Brug af software som medicinsk udstyr, Artikel 5.5, brug af kvalitetssikringssystem (QMS)).

Versionskontrol omhandler to ting: Versionering og en stringent kontrol af samme.

Versionering

Versionen af et stykke software har til formål at beskrive og nøjagtigt definere en specifik udgave af softwaren, som er specifik for en bestemt kildekode og kompilering. Versioneringshistorikken fortæller om softwarens udvikling og nuværende tilstand.

Det er umuligt at lave software helt uden fejl. Bl.a. derfor er det vigtigt at have fuldt ud styr på, hvilken kildekode der ligger bag et givent kompileret stykke software. Desuden fortæller en god versioneringspraksis om størrelsesordenen af ændringer i funktionalitet og kompatibilitet med anden software.

Et meget anvendt format for versionering er semantisk versionering (SemVer 2.0.0, https://semver.org), der netop gør det tydeligt, hvor store ændringerne er, samt hvilken kompatibilitet udviklerne forventer med disse ændringer. I semantisk versionering får hver version et versionsnummer af formen MAJOR.MINOR.PATCH. Man øger tallene således:

- MAJOR version, når du foretager inkompatible API-ændringer[7],
- MINOR version, når du tilføjer funktionalitet på en bagudkompatibel måde, og
- PATCH-version, når du laver bagudkompatible rettelser.

Eksempler på versionering:

"Script 1.0.0" er udviklet til at fungere med dosisplanlægningssystemet Eclipse 13.5. I denne version af Eclipse findes en property for objektet PlanSetup med navnet TotalPrescribedDose. Eclipse 15.6 signalerer brud på funktionalitet for aftagende programmer; f.eks. hedder ovenstående property nu TotalDose. Script 1.0.0 benytter TotalPrescribedDose og fungerer derfor ikke længere med Eclipse 15.6 som backend, hvorfor der laves en rettelse i Script 1.0.0. Rettelsen kan laves på to måder:

1) *Rettelsen er bagudkompatibel så Script understøtter både Eclipse 13.5 og 15.6: Ingen ændring i MAJOR. Rettelsen tilføjer ikke ny funktionalitet: Ingen ændring i MINOR. Den nye version bliver således Script 1.0.1.*
2) *Rettelsen er ikke bagudkompatibel så Script nu kun understøtter Eclipse 15.6: MAJOR øges. Den nye version bliver således Script 2.0.0.*

[7] Et Application Programming Interface (API) er et programmatisk interface for et stykke software, som andre programmer kan benytte til at interagere med softwaren.

"ClassLibrary 1.0.0" bliver brugt flittigt i andre programmer og opdateres løbende:

- *Der laves en ny privat metode AwesomeCalculation, som skal understøtte en kommende funktionalitet. For aftagende programmer er der dog ingen ændring at se. Kun PATCH øges og versionen bliver 1.0.1.*
- *Der tilføjes en funktionalitet som de andre programmer kan se og bruge via en offentlig metode NewCoolFeature, som bl.a. benytter AwesomeCalculation. Ellers virker alt som det plejer. MINOR øges, PATCH nulstilles og versionen bliver 1.1.0.*
- *Der rettes en fejl i AwesomeCalculation, men funktionaliteten er ellers den samme. Kun PATCH øges og versionen bliver 1.1.1.*
- *Der ændres i AwesomeCalculation, så denne og NewCoolFeature nu gør noget grundlæggende anderledes, som ikke blot kan kategoriseres som en mindre forbedring. MAJOR øges, MINOR og PATCH nulstilles og versionen bliver 2.0.0.*

Idriftsætninger (releases) med MAJOR nummer nul (0.MINOR.PATCH) betyder at softwaren stadig er under konstant udvikling, og at inkompatible API ændringer kan forekomme på ethvert tidspunkt. Når første stabile release udgives bruges MAJOR nummer et (1.MINOR.PATCH).

Yderligere etiketter til pre-release og build-metadata er tilgængelige som udvidelser til MAJOR.MINOR.PATCH-formatet; f.eks. 1.3.2-beta.

Anbefalinger

Det anbefales at benytte en klar og entydig versionering. Versionsnummer skal fremgå klart i både kildekode og kompileret software.

Det anbefales at benytte semantisk versionering (https://semver.org), så det bliver tydeligt, hvor store ændringer, der er lavet i softwaren, samt hvilken kompatibilitet man kan forvente.

Bemærk at operativsystemet Windows har en særlig for for versionering. Dette er beskrevet i Appendiks D: Versionsetiketter på Windows.

Logbog

En versionslogbog kan have forskellig detaljegrad afhængigt af formålet:

- **Historik**
  Beskriver ændringer for *hver eneste* version af koden. Lav brugbare beskrivelser, f.eks. "Fix text label position off screen when … (close #23)" i stedet for "bugfix". Benyttes et Versionskontrolsystem følger historikken automatisk med.
- **Changelog**
  En beskrivelse af relevante ændringer i koden af både brugsmæssig og teknisk karakter. Kan f.eks. udformes som en kort beskrivelse af den nye version og en sammenfatning af historikken, hvor meget detaljerede beskrivelser forkortes og ændringsbeskrivelser som "Fix typo" eller "Add comment to…" sorteres væk.
- **Release** notes
  Et resumé af de helt store ændringer i en udgivelse af et program siden seneste udgivelse. Skrives i et almindeligt sprog og uden tekniske beskrivelser. Release notes er generelt mere effektive, jo kortere og mere præcise de er.

Vær opmærksom på særlige krav til logbog ifm. MDSW, fx. krav om historik ("traceability").

Versionskontrolsystem

Versionskontrol hjælper udviklere med at spore og administrere ændringer i et softwareprojekts kode. Efterhånden som et softwareprojekt vokser, bliver versionskontrol altafgørende. Versionskontrol kræver logbogsføring af ændringer i kildekoden. Hver logbogsføring skal som minimum indeholde en versioneringsetiket, samt en beskrivelse af hvad der er ændret, hvem der har lavet ændringen og hvornår ændringen og/eller logbogsføringen er foretaget.

Den simpleste løsning

Som absolut minimum bør man opbevare en kopi af kildekoden for hver eneste version samt en form for logbog, der beskriver alle ændringerne. Se afsnittet om Logbog.

Ofte ses en håndholdt kopiering af kildekode til en særskilt folder med versioneringsnavngivning og en kort beskrivelse i en tekstfil. Fordelen ved denne løsning er dens simplicitet, som gør den nem at komme i gang med. Ulemper kan derimod være, at den kræver stor omhyggelighed og pålidelig brug fra udviklerens side for at være troværdig, samt at man ved en menneskelig fejl kan komme til at slette tidligere versioner eller logbogsføringer.

For at imødegå sidstnævnte kan man i stedet bruge en database, hvor man uploader kildekoden sammen med logbogsføringen. Eksempler på sådanne "databaser" kunne være MediaWiki eller Confluence.

Ovenstående håndholdte løsning ses dog primært brugt for enmandsprojekter.

Moderne standard

Så snart flere udviklere samarbejder på samme kode mere eller mindre samtidigt, vil det hurtigt være en fordel at bruge dedikeret software til at styre versionskontrollen. Denne slags software kaldes et Version Control System (VCS) eller et Source Control Management (SCM) system. De to termer bruges ofte til at beskrive det samme, omend nogle tillægger SCM ekstra funktionalitet over VCS.

Eksempler på denne type software kunne være Git, Mercurial, Subversion eller Azure DevOps. Udover at automatisere store dele af versionskontrollen, har mange af disse systemer også ekstra funktionaliteter, som gør det nemmere at udvikle nye funktioner, finde og rette fejl i koden, holde styr på forskellige idriftsætninger og meget andet.

Git - et eksempel på versionskontrolsystem:

Git er nutidens defacto standard[8] indenfor versionskontrol. Git er et meget stærkt værktøj med utrolig mange funktioner. Der findes dokumentation og interaktive tutorials på hjemmesiden https://git-scm.com.

Mange finder det lidt svært i begyndelsen, men man kommer meget langt med følgende funktionaliteter:

1. *Init: Lav en alm. folder til et git repository.*
2. *Clone: Hent en lokal kopi af et eksisterende 'remote' git repositorie.*
3. *Status: Se status af filer i en folder; hvad er ændret lokalt og remote.*
4. *Add: Tilføj ændringer (editerede, nye eller slettede filer) til næste commit ('stage').*

[8] Se f.eks. "Developer Survey Results 2018": https://insights.stackoverflow.com/survey/2018/#work-version-control.

5. *Commit: Tilføj stagede ændringer til git historikken ("logbogen"). Hvert commit indeholder de valgte ændringer, en beskrivelse af hvad ændringerne gør, forfatter, et tidsstempel samt et unikt hash.*
6. *Pull: Hent commits fra remote.*
7. *Push: Skub commits til remote.*

Som udgangspunkt bruger man git via en terminal/kommandoprompt, men der findes også flere tredjepartsprogrammer, som giver brugeren en grafisk brugergrænseflade, f.eks. SourceTree eller TortoiseGit. Visual Studio understøtter også git indlejret.

Ulempen ved ovenstående kan være en stejl læringskurve, som kan afskrække nye og uerfarne udviklere.

Uanset valget af SCM er det god praksis at lave små commits med afgrænsede, veldefinerede formål; såkaldte "atomic commits". Dette gør det væsentligt nemmere at forstå ændringer i koden senere og udnytte øvrig SCM funktionalitet ifm. lokalisering og rettelser af fejl mv.

Opbevaring af kildekode

Uanset valg af versionskontrolsystem skal kildekoden opbevares et sted. I den forbindelse bør det overvejes om man skal bruge et simpelt filsystem, en database i stil med det håndholdte eksempel ovenfor eller en mere komplet SCM platformsløsning med integreret sagshåndtering ("issue tracker") og kommunikationsplatforme mv.

Eksempler på cloud-baserede platforme kunne være GitHub, GitLab eller BitBucket. GitLab tilbyder også mulighed for installation på egen lokal server.

Ligeledes bør det grundigt overvejes, evt. for hvert enkelt udviklingsprojekt, om man kan opbevare kildekoden udenfor afsnit, afdeling og/eller region i henhold til gældende lovgivning (se afsnittet Jura). Særlig opmærksomhed bør tildeles overholdelse af Persondataforordningen/GDPR, således at personhenførbar information ikke ved uheld eller uagtsomhed bliver tilgængelig for tredjepart; f.eks. ved hardcodede test cases i kildekoden.

Anbefalinger

Det anbefales, at al softwareudvikling bruger en eller anden form for versionskontrol. Den sikreste løsning er en lokal installation af en SCM-platform med issue tracker, der sikrer både GDPR-hensyn, flere elementer af MDR og effektiv softwareudvikling.

Automatiseret versionering

Afhængig af omfanget af lokal softwareudvikling kan det give mening at automatisere versioneringen for at lette arbejdsbyrden på sigt. Dette er integreret i nogle SCM systemer og kan tilføjes andre via tredjepartssoftware.

Eksempel på automatisk versioneringsværktøj:

Til .NET projekter kunne et automatisk versioneringsværktøj f.eks. være Nerdbank.GitVersioning (https://github.com/dotnet/Nerdbank.GitVersioning#readme), som er let tilgængelig som en NuGet pakke.

MAJOR og MINOR version mv. styres let i en version.json fil. PATCH nummer, pre-release og build-metadata håndteres automatisk ud fra Git-historikken.

Dokumentation

Et stykke software eller program skal betragtes som et produkt, hvortil der skal følge en overordnet dokumentation. Dokumentet skal fungere som en brugervejledning, med det formål at en ny bruger af programmet relativt hurtigt kan danne sig et overblik over implementering og anvendelse. Såfremt softwaren skal benyttes klinisk skal dokumentationskrav opfylde MDR jf. softwarens klassificering (se MDR-forordningen). Udarbejdelsen af dokumentationen skal ses som en integreret del af den overordnede software udviklingsproces.

Dokumentet skal ikke nødvendigvis gå i stor detalje med f.eks. forklaring af den bagvedliggende kodes centrale logik, idet dette kan forklares vha. kommentarer i selve koden (se afsnittet Programmeringsstil (code of practice)). Dog bør overordnede valg og betragtninger beskrives kort og præcist.

Den overordnede software-arkitektur bør dokumenteres med passende figurer, ifald formidling af disse er vigtig for fremtidig vedligehold og videreudvikling af koden. Hvorvidt denne mere detaljerede beskrivelse er nødvendig vil afhænge af kodens kompleksitet og i hvor høj grad programmet integrerer med andre systemer (f.eks. databaseadgang). Figurerne skal indeholde flows-charts der illustrerer programopbygning og hvilke andre systemer der interageres med.

Komplekse programmer kan man med fordel dokumentere ved hjælp af flowcharts, der beskriver kodens opbygning. Dette hjælper til at skabe overblik over programmets opbygning og afvikling, samt hvilke delelementer af koden, der gør hvad. Denne visuelle dokumentation har flere fordele: Det er nemmere for andre at gennemgå og forstå i forhold til dels kvalitetssikring og dels at overtage koden, og det er nemmere at identificere de kritiske algoritmer i relation til risikovurdering af softwaren (se afsnittet Vurdering af de enkelte risici).

Man skal være opmærksom på, at der findes en international standard for flowcharts[9], hvor de enkelte blokke har bestemte former efter om de er f.eks. beslutningsprocesser, input/output eller en dataoperation. Der findes forskellig software, der gør det nemt at tegne software-flowcharts. En udbredt, gratis udgave kan findes på https://app.diagrams.net (tidligere kendt som draw.io).

Det bør sikres at programkendskab, på et niveau hvor opdateringer og udbedringer af fejl kan udføres, ikke kun hviler på én person. Software-dokumentationen skal ses som et led i at sikre dette. Den enkelte afdeling kan med fordel udforme sin egen dokumentationsskabelon, som udfyldes ved frigivelsen af ny software. Omfanget og detaljegraden af dokumentationen må vurderes ift. programmeringskompetencerne på den enkelte afdeling, med det formål at øge levedygtigheden af programmet, således at føromtalte opdateringer og fejludbedringer kan udføres.

Det skal overvejes om der er regionale retningslinjer, der skal følges ift. MDR.

[9] "*ISO 5807:1985 Information processing*". Et overblik kan f.eks. findes på https://en.wikipedia.org/wiki/Flowchart.

Anbefalinger

Det anbefales, at man gør dokumentation af software til et krav i lokal softwareudvikling. Et nyt program, eller en ny version af programmet må aldrig frigives, før dokumentation er udarbejdet.

Det anbefales, at den enkelte afdeling udarbejder sin egen dokumentationsskabelon således, at dokumentationskrav er overholdt og så softwaren kan serviceres og videreudvikles af andre end kodeforfatter(ne) selv.

Det anbefales, at dokumentationen af komplekse programmer suppleres med flowcharts, der skaber overblik over koden, hvilket både er til gavn i forhold til kvalitetssikring, risikovurdering og overdragelse af koden til andre udviklere.

Appendiks E: Eksempel på dokumentationsskabelon for software indeholder et eksempel på, hvordan en minimumsdokumentation for et stykke software for eksempel kan se ud.

Udvikling, test og godkendelse af software

Udvikling og test af software

Udvikling og test af egenudviklet software til klinisk brug bør foregå på et testsystem. Udføres softwareudvikling på et klinisk system, skal man være opmærksom på forøget risiko for kompromittering af patientdata samt påvirkning af ydeevnen af det kliniske system. På et testsystem kan der udvikles og testes software uden nogen godkendelser.

Et stykke software er først færdigudviklet, når en række betingelser er opfyldt (se boks). Dette inkluderer bl.a. en grundig test af softwaren ved brug af en række værktøjer.

Anbefalinger

Det anbefales, at et stykke software først betragtes om færdigudviklet, når følgende er opfyldt:

- Version 1.0.0 af koden er færdig.
- Der er udarbejdet tilstrækkelig dokumentation (se afsnittet Dokumentation).
- Der er lavet en risikovurdering (se afsnittet Risikovurdering).
- Der er udført en grundig test af software med baggrund i risikovurderingen, for eksempel:
    - Unit tests af den vigtigste funktionalitet.
    - Tests på en række testpatienter, således at alt funktionalitet i koden er afprøvet.
    - Tjek af belastningen på testsystem ved krævende beregninger eller databasekald.

Godkendelse af software

Når software til klinisk brug er færdigudviklet, skal softwaren gennemgå en godkendelsesprocedure. Afdelingen bør have en fastsat arbejdsgang for, hvordan dette foregår. Godkendelsen bør foretages af mindst én programmeringskompetent fagperson i afdelingen, som ikke har deltaget i udviklingen af koden. Dette sikrer både en uafhængig godkendelse, og at der er mindst to fagpersoner med kendskab til softwaren i afdelingen, hvilket medvirker til en mere robust drift af softwaren i tilfælde af jobskifte, sygdom mm.

Godkendelsesproceduren af softwaren skal vurdere om softwaren er egnet og lovlig til brug på det kliniske system, hvilket kræver gennemgang af flere forskellige aspekter af softwaren (se boks).

Anbefalinger

Det anbefales, at afdelingen har en veldefineret arbejdsgang for, hvordan software godkendes til brug, samt hvordan det dokumenteres, at softwaren er godkendt.

Godkendelsesproceduren bør foretages af mindst én programmeringskompetent fagperson i afdelingen, som ikke selv har deltaget i udviklingen af koden. Det anbefales, at afdelingen udpeger ansvarlige personer for godkendelsesproceduren.

Godkendelsen bør inkludere:

- Indledende dialog med udvikleren/udviklere om baggrund, funktion og opbygning af software
- Afklaring af juridiske forhold, inkl. MDR, GDPR mv. (se afsnittet Jura).
- Grundig gennemgang af kode med fokus på kodens læsbarhed (se afsnittet Programmeringsstil (code of practice)), mulighed for at simplificere koden, håndtering af fejltilstande i koden mm. Gennemgangen skal have særlig fokus på dele af koden med kritisk funktionalitet (f.eks. kode der beregner kliniske resultater) og evt. mindre fokus på dele af koden som definerer det grafiske layout.
- Gennemgang af dokumentation (se afsnittet Dokumentation), herunder gerne flowcharts.
- Gennemgang af risikovurdering (se afsnittet Risikovurdering).
- Gennemgang af test af software.
- Stress-test af kode ved brug af forkerte input og ekstreme værdier.

Først efter godkendelse af softwaren, kan softwaren idriftsættes på det kliniske system. Her kører først en testfase, hvor det undersøges om softwaren fungerer som forventet. I nogle systemer er det muligt at lave en betinget godkendelse for en mindre gruppe af testbrugere. Efter en godkendt testfase skal brugerne undervises i brug af softwaren. Først herefter kan softwaren idriftsættes for alle brugere på det kliniske system.

Hvis der opdages en alvorlig fejl i softwaren, eller softwaren ikke længere bruges, skal godkendelsen for softwaren på det kliniske system trækkes tilbage.

Egenudviklet software til klinisk brug bør implementeres med samme omhu som kommerciel software og hardware (f.eks. acceleratorer og dosisplanlægningssystem). Omfanget af implementeringen skal modsvare risikoklassificeringen jf. MDR. Dette kræver en arbejdsindsats, som bør ses som en investering snarere end en udgift. Et grundigt arbejde ifm. første implementering vil uden tvivl styrke og effektivisere udvikling, godkendelse og idriftsættelse af senere versioner.

Appendiks

Appendiks A: Ordliste

- **API**: *Application Programming Interface*; en softwaregrænseflade, der tillader ét stykke software at interagere med andet software.
- **Assembly**: En samling af kompilerede kodefiler. På Windows typisk i en DLL eller EXE fil.
- **C#**: Et programmeringssprog, der udtales "C Sharp". Anvendes f.eks. i ESAPI.
- **Confluence**: Software der anvendes til dokumentstyring. Kan anvendes til at opbevare instrukser og filer, som tilgås via en browserbaseret brugergrænseflade. Et kommercielt alternativ til MediaWiki.
- **DAHANCA**: *Danish Head and Neck Cancer*; den danske multidisciplinære cancer-gruppe for hovedhals-kræft.
- **DCPT**: *Dansk Center for Partikelterapi*; et national stråleterapicenter beliggende på Aarhus Universitetshospital.
- **Eclipse**: Et kommercielt dosisplanlægningssystem fra Varian Medical Systems. Anvendes klinisk på 6 af 8 danske stråleterapier.
- **ESAPI**: *Eclipse Scripting API*; et system der tillader, at man udvikler egenproducerede C#-scripts, der interagerer med dosisplanlægningssystemet Eclipse.
- **GDPR**: *Persondataforordningen*; håndhæves af Datastyrelsen og beskriver lovkrav ifm. adgang til og brug af personhenførbare data.
- **Git**: Et udbredt eksmpel på et softwaresystem til automatisk versionskontrol af kildekode.
- **IDE**: *Integrated Development Environment*; softwareudviklingsmijø til udvikling af ny software. For eksempel Microsoft Visual Studio.
- **IDE**: *Investigational Device Exemption*;
- **MDR**: *Medical Device Regulation*; MDR-forordningen håndhæves af Lægemiddelstyrelsen og beskriver europæiske lovkrav til medicinsk udstyr inkl. software.
- **MDCG**: *Medical Device Coordination Group; en ekspertgruppe under Europa-Kommissionen.*
- **MDSW**: *Medical Device Software*; software brugt som medicinsk udstyr.
- **MediaWiki**: Et stykke gratis, open-source software, der anvendes til dokumentstyring. Kan anvendes til at opbevare instrukser og filer, som tilgås via en browserbaseret brugergrænseflade.
- **MIT-licensen**: En udbredt softwarelicens, der er forholdvis lidt restriktiv i forhold til at lade andre bruge softwaren. Se f.eks. formulering her: https://opensource.org/licenses/MIT.
- **MVVM**: *Model–view–viewmodel*; et mønster for software-arkitektur, der adskiller den grafiske brugerflade fra den bagvedliggende logik.
- **Pinnacle**: Et kommercielt dosisplanlægningssystem fra Philips. Anvendes klinisk på 1 af 8 danske stråleterapier.
- **QA**: *Quality Assurance*; et samlet system af værktøjer og procedurer til at sikre, at planlagte og udførte behandling stemmer overens og dermed undgår fejl.
- **QMS**: *Quality Management System*; et system til kvalitetssikring af medicinsk udstyr.
- **RayStation**: Et kommercielt dosisplanlægningssystem fra RaySearch Laboratories. Anvendes klinisk på 1 af 8 danske stråleterapier.
- **SCM**: *Source Control Management system*; et softwaresystem til automatisk versionskontrol af kildekode.
- **SDK**: Software Development Kit. En pakkeløsning, der indeholder underliggende software-pakker og udviklingsværktøjer. Et eksempel er .Net Framework.

- **Script**: Et stykke software, der anvendes til at interagere med et eksisterende stykke software, hvorved man kan automatisere eller forbedre funktionalitet.
- **Scripting**: Det er programmere et script. Se “Script”.
- **Snippet**: En lille, afgrænset kodestump, der nemt kan genbruges og kopieres ind i en ny sammenhæng.
- **SOLID**: Et akrononym for en række principper, der anvendes til at gøre objekt-orienteret programmering mere forståelig.
- **SOUP**: *Software of unknown/uncertain pedigree/provenance*; software udviklet af andre og/eller efter en ukendt sikkerhedsstandard. Anvender man dette i sin egen kode, betragtes det som særligt risikabelt.
- **SQL**: *Structured Query Language*; et særligt computersprog der anvendes til at lave opslag i databaser.
- **TTO**: *Technology Transfer Office*; en enhed som har til formål at støtte ansatte i forbindelse med teknologioverførsel, dvs. udveksling af viden og opfindelser mellem arbejdspladsen og det resterende samfund, inkl. kommercialisering.
- **VCS**: *Version Control System*; et softwaresystem til automatisk versionskontrol af kildekode.

Appendiks B: Eksempel på model for risikovurdering

Herunder gives et eksempel på én model, der kan anvendes til risikovurdering:

Sandsynligheden for en hændelse og alvoren af en hændelse (konsekvens) scores på følgende skalaer:

Sandsynlighed	Konsekvens
1. Usandsynligt 2. Sjældent 3. Muligt 4. Sandsynligt 5. Ofte	1. Ubetydelig 2. Mindre 3. Moderat 4. Større 5. Katastrofal

Ud fra disse faktorer kan risikoen for den enkelte hændelse vurderes ud fra risikotallet:

Sandsynlighed	**Risikotal = Sandsynlighed x Konsekvens**				
Ofte	5	10	15	20	25
Sandsynligt	4	8	12	16	20
Muligt	3	6	9	12	15
Sjældent	2	4	6	8	10
Usandsynligt	1	2	3	4	5
	Ubetydelig	Mindre	Moderat	Større	Katastrofal
	Konsekvens				

Risikograd: Acceptabel (1-4), Signifikant (5-9), Uacceptabel (10-25)

Når man har kortlagt risici og scoret dem ud fra ovenstående model, har man to handlemuligheder såfremt en hændelse har et for højt risikotal: 1) Man kan forsøge at forebygge, så hændelsen bliver mindre sandsynlig, eller 2) man kan lave en plan for at afhjælpe problemet, så konsekvenserne begrænses.

Hvis risikograden er Acceptabel (risikotal 1-4), behøver man ikke at foretage sig yderligere. Hvis risikograden er Signifikant (5-9), bør man forsøge at reducere risikotallet. Kun hvis dette ikke er teknisk eller praktisk muligt, kan man vælge at acceptere denne risikograd. Hvis risikograden er Uacceptabel (10-25) skal risikotallet altid reduceres, før softwaren tages i brug.

Appendiks C: Eksempler på god programmeringsstil

Eksempler i det følgende er skrevet i C#, men vil være relevante for softwareudvikling i alle andre sprog i et eller andet omfang. Kommentarer angives med '//'.

Navngivning

Navngivning af variable, funktioner osv. er afgørende for hvor nemt et stykke kode er at overskue. Der findes mange forskellige konventioner (som også afhænger af programmeringssproget), og dette dokument har ikke til hensigt at foreslå én bestemt måde at skrive kode på. Det er dog en klar anbefaling, at man holder sig til den samme navngivningskonvention indenfor et stykke software. Hvis hensigten er at dele et stykke kode med en større kreds, kan det være en fordel at holde sig til de sprogspecifikke navngivningsskikke (kilder hertil er opført nederst i afsnittet).

Et par generelle principper kan være værd at overveje, uafhængigt af programmeringssprog og -arkitektur:

- **Navngivningen bør afspejle funktion og hensigt.**
  Brug entydige, beskrivende navneord til variable/konstanter/klasser og udsagnsord til funktioner/metoder.
- **Brug konsistent navngivningsstruktur for objekter af samme type.**
  F.eks. *PascalCase* til alle klasser, *lower_case* (med underscore) til alle variable, *camelCase* til alle argumenter i en metode, etc.
- **Undgå unødvendigt lange navne.**
  Hvis et langt navn er nødvendigt for at sikre entydighed, overvej en letforståelig forkortelse i stedet (som skal defineres i en kommentar).

Eksempler:

	Hensigtsmæssig	**Uhensigtsmæssig**
Variabel / Konstant	heart_mean_dose, heartMeanDose, eller lignende	D, dosis, a
Variabel / Konstant (langt navn)	heart_mean_dose_fx15 // heart mean dose at fraction 15	heart_mean_dose_at_fraction_15
Funktion / Metode	CalculateDose, calculate_dose, eller lignende	calc, func
Klasse	DosisPlan, dosisplan, eller lignende	dp
Exception (i en *catch* blok f.eks.)	doseNotDefined, invalid_dose, eller lignende	exception

Kommentering

Generelt for kommentarer gælder, at kode bør kommenteres i et sådant omfang, at den kan forstås af en anden programmør - og helst ikke mere end det. Det er meget individuelt, hvor detaljeret den enkelte programmør vælger at kommentere sin kode. Det anbefales dog, at overveje følgende principper:

- **Brug kommentarer (kun) dér, hvor koden ikke er selv-forklarende.**

  Følgende kommentar er unødvendig:

```
// a og b lægges sammen
double sum = a + b;
```

  Det kan tit give mening at lave en "help header" til et stykke kode (i C# tilføjes den hurtigt ved at skrive /// ovenover f.eks. en metode):

```
/// <summary>
/// Method to get the voxelwise dose values from the dose object for a given structure
/// and return them as a 3D array.
/// <param name="dose">Dose object</param>
/// <param name="structure">Structure object</param>
/// <returns>3D double array of voxel dose values. Voxels outside the structure are set to 0.</returns>
/// </summary>
```

- **Skriv hele sætninger, som kan forstås af andre (minimér brug af "lokal slang").**

- **Husk at ting, som virker indlysende for dig selv lige nu, ikke nødvendigvis er indlysende for andre (eller dig selv om et år). Kommentér hellere lidt for meget end lidt for lidt.**

For et større program, der indeholder mange filer, kan det være hjælpsomt at skrive en Readme-fil eller tegne klassediagrammer (draw.io er et eksempel på et open-source værktøj til dette formål), der kort opsummerer de enkelte filers funktion og sammenhæng (se også afsnittet Dokumentation).

Opbygning af den enkelte fil

For hvert programmeringssprog findes der konventioner til opbygningen af kode-filer. Disse bør så vidt muligt overholdes for at holde koden overskuelig. Et simpelt eksempel på en sådan konvention er, at de brugte softwarepakker anføres øverst i filen i en C# klassedefinition:

```
using System.Windows
using VMS.TPS.Common.Model.API
using VMS.TPS.Common.Model.Types

// Her starter resten af programmet
```

Konventioner kan variere afhængigt af programmeringssprog og -stil. Nogle generelle anbefalinger kan dog være værd at overveje:

- **Skriv ikke mere end én deklaration per linje**
- **Indentér linje 2 og derefter, hvis en lang deklaration strækker sig over flere linjer**

- **Vær påpasselig med brugen af implicitte typer i stærkt-typede sprog ("var" i C# f.eks.)**
- **Brug ikke unødvendig komplex syntax (bare fordi det *kan* skrives på én linje, betyder det ikke at det *bør*)**
- **Strukturér en fil således at delelementernes rækkefølge giver mening**
  Eks.: C# klasse

```
public class MinKlasse
{
   // Felter står øverst
   public double A;
   public double B;

   // Constructor derefter
   public MinKlasse(double a, double b)
   {
      A = a;
      B = b;
   }

   // Metoder
   public double metode1()
   {
      // en metode
   }

   public void metode2()
   {
      // en anden metode
   }
}
```

Mange udviklingsmiljøer (integrated development environments - IDEs) indeholder værktøjer til visuel strukturering af kode. Her kan bl.a. nævnes #region funktionaliteten i VisualStudio eller %% i MATLAB. Disse værktøjer tillader at sammenklappe koden i forskellige afsnit for at gøre filen nemmere at overskue.

Noget så simpelt som brugen af tabs og mellemrum kan gøre en verden til forskel for overskueligheden i et stykke kode. Generelt anbefales det at sætte mellemrum efter kommaer og semikolon, og både før og efter matematiske operatorer (c = a + b). Indentering anbefales til at fremhæve kodens understrukturer (klasser, metoder, løkker, etc.). Tabulatoren anbefales indstillet til at være fire mellemrum. Ligeledes anbefales én tom linje imellem to delelementer i koden. Eksemplet ovenfor anskueliggør disse principper.

Links til mere detaljerede kode-konventioner i forskellige programmeringssprog er anført sidst i afsnittet.

Opbygning af hele programmet

Når det gælder opbygningen af et større programmeringsprojekt, er det igen vigtigt at gøre det så overskueligt som muligt. Ligesom med opbygningen af de enkelte filer er der ingen faste regler, men det anbefales at overveje følgende punkter:

Længden af de enkelte filer: Hvis en fil begynder at blive meget lang og indeholde mange forskellige funktionaliteter, bør man overveje at splitte den i flere mindre filer og evt. klasser jf. SOLID princip beskrevet nedenfor.

Navngivning af filer: De enkelte filer bør navngives, så det er nemt at forstå filens funktion ud fra navnet.

Der er publiceret et antal forskellige softwarerammer (frameworks) til software designprincipper. Det er ikke hensigten med dette dokument at beskrive alle disse i detaljer, og læseren henvises til anden relevant litteratur for dybdegående diskussioner af de forskellige designmønstre. To populære eksempler beskrives kort i det følgende.

SOLID
Solid design modellen henvender sig til objekt-orienteret programmering (OOP). Den består af fem principper:

S: Single Responsibility Principle - En klasse bør kun have ansvaret for én bestemt del af programmets funktionalitet.

O: Open/closed Principle - En klasses funktionalitet bør kunne udvides *uden at ændre koden til selve klassen*. I praksis kan dette gøres ved brug af interfaces og abstrakte klasser.

L: Liskov Substitution Principle - Objekter y af en klasse Y, som er en sub-klasse af klasse X, bør kunne erstatte objekter x af klassen X. Dvs. objekter y skal som minimum implementere de samme egenskaber og metoder som objekter x.

I: Interface Segregation Principle - En klasse, som implementerer et interface *I*, bør ikke være tvunget til at implementere alle metoder til rådighed i *I,* hvis der ikke er brug for dem. I stedet for et stort interface *I* med for mange metoder bør der i stedet laves flere mindre adskilte interfaces.

D: Dependency Inversion Principle - En klasse bør ikke afhænge af en anden konkret klasse, men i stedet af en abstraktion (dvs. et interface).

Brug af SOLID designprincipperne kan være en hjælp til at gøre ens kode mere overskuelig og robust ved ændringer. Læseren henvises iøvrigt til relevante eksterne ressourcer for en mere detaljeret gennemgang. Se for eksempel:
Introduction to SOLID Design Principles | by Sangeeth Arulraj | Nerd For Tech | Medium
Introduction To SOLID Principles (c-sharpcorner.com)

MVVM (Model - View - ViewModel)
MVVM modellen er en specifik måde at opbygge programmer med en brugeroverflade på. Tanken er at adskille de dele af koden, der styrer brugeroverfladen (View og ViewModel), fra de dele, der foretager beregninger (Model).

View er det, der styrer brugeroverfladen. Det kan f.eks. være et XAML vindue, der specificerer, at der skal være et tekstfelt og en "Enter" knap på en grøn baggrund.
ViewModel styrer brugerfladens funktionalitet. Det vil være her, man definerer en funktion, der læser tekstfeltets indhold ved tryk af "Enter" knappen og sender det videre til den egentlige kode.
Model indeholder så den egentlige funktionalitet. Det kan være en funktion, der gennemser en patients struktursæt og giver en melding, hvis en strukturs navn stemmer overens med det, der blev skrevet i

tekstfeltet. Det er muligt at lade Modellen returnere en "True" boolean værdi til ViewModellen, som så vil få View til at vise denne melding.

Brug af MVVM modellen gør koden nemmere at vedligeholde og nemmere at samarbejde omkring i en større gruppe. Læseren henvises iøvrigt til relevante eksterne ressourcer for en mere detaljeret gennemgang. F.eks.:
https://www.tutorialspoint.com/mvvm/mvvm_introduction.htm
https://www.c-sharpcorner.com/UploadFile/ptmujeeb/wpf-mvvm-pattern-a-simple-tutorial-for-absolute-beginners/

Mange aspekter af god programmeringsskik har været emner for virtuelle foredrag i gruppen af danske ESAPI brugere. Disse kan ved forespørgsel findes i følgende Google-Docs dokument: ESAPI Netværksgruppe - Google Tabellen.

Eksempler på guidelines og konventioner

C#: https://mitcommlab.mit.edu/broad/commkit/coding-and-comment-style/
C# Coding Guidelines And Best Practices v1.0 (c-sharpcorner.com) (C# intro level.)
https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions
https://github.com/dotnet/runtime/blob/main/docs/coding-guidelines/coding-style.md

Python: **PEP 8 – Style Guide for Python Code | peps.python.org**
Code Style — The Hitchhiker's Guide to Python (python-guide.org)

MATLAB: **MATLAB Style Guidelines 2.0 - File Exchange - MATLAB Central (mathworks.com)**

Appendiks D: Versionsetiketter på Windows

Vær opmærksom på, at på operativsystemet Microsoft Windows defineres versionen af et "assembly"[10], med følgende deletiketter: major, minor, build og revision; hvilket *ikke* er i overensstemmelse med SemVer (se afsnittet om Versionering). Microsoft er dog selv gået væk fra denne opbygning af version i nyere projekter. Der findes desuden tre versionsetiketter for hver DLL- eller EXE-fil:

- AssemblyVersion:
  Specificerer versionen af det kompilerede *assembly*. Det er her andre programmer, som benytter dette *assembly* kigger. Hvis dette nummer ændrer sig, skal de aftagende programmer opdatere deres referencer for at kunne kompilere. Opdater derfor kun hvis versionen ikke er bagudkompatibel. Benyttes SemVer, betyder det at kun MAJOR opdateres, dvs. version 2.1.3 har AssemblyVersion 2.
  AssemblyVersion er påkrævet.

- AssemblyFileVersion:
  Bruges som versionsnummer for filversionen. AssemblyFileVersion behøver ikke være den samme som AssemblyVersion. Bruges til at adskille programmer genereret af forskellig kode eller kompileringer ("*builds*"). Denne version øges som minimum for hver offentliggørele af koden. Benyttes SemVer har version 2.1.3 typisk AssemblyFileVersion 2.1.3.7894, hvor 7894 er et unikt *build* nummer, som ofte genereres automatisk[11].
- AssemblyInformationalVersion:

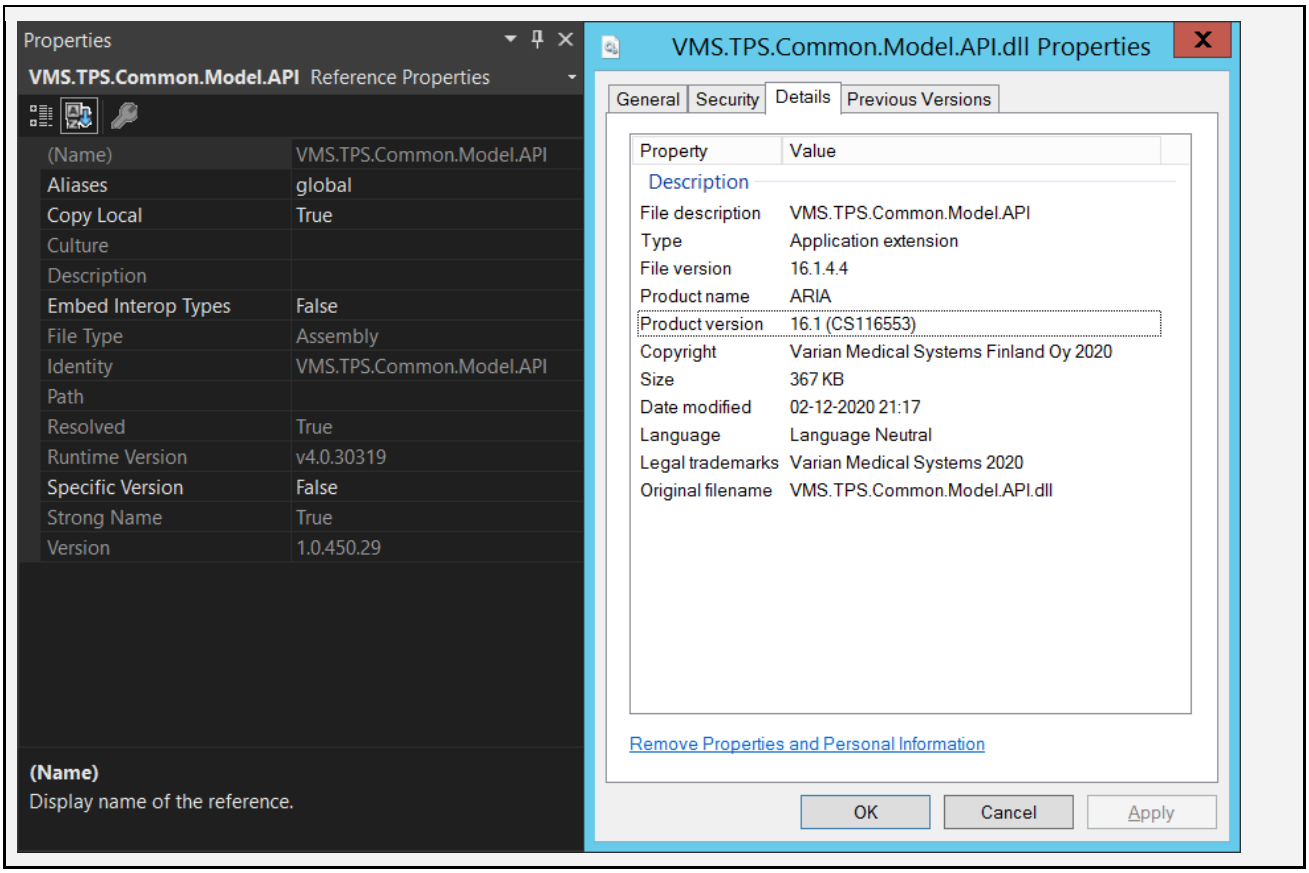
  Definerer yderligere versionsinformation for et *assembly*, og benyttes oftest til "produktversionen" af programmet. Dette er den version et firma ville bruge på deres hjemmeside, eller når de talte med kunder. Denne version kan være alt muligt, man kunne bruge til at beskrive sit produkt, f.eks. "2.1.3-beta", "2.1 Release Candidate", "v2 foXy Physics" osv.

ESAPI som eksempel på versionering:

En udgave af Varian ESAPI, VMS.TPS.Common.Model.API.dll, har følgende versionnumre:

- *AssemblyVersion: 1.0.450.29*
  ESAPI er "hårdt navngivet", dvs. man altid skal opdatere referencer ved nye udgivelser af ESAPI.
- *AssemblyFileVersion: 16.1.4.4*
- *AssemblyInformationalVersion: 16.1 (CS116553)*

[10] Et *assembly* er en samling af kompilerede kodefiler. På Windows typisk i en DLL eller EXE fil.
[11] I Visual Studio kan dette gøres med et wildcard '*': [assembly: AssemblyFileVersion("2.1.3.*")]. Se også Automatiseret versionering.


Properties
VMS.TPS.Common.Model.API Reference Properties
(Name) VMS.TPS.Common.Model.API
Aliases global
Copy Local True
Culture
Description
Embed Interop Types False
File Type Assembly
Identity VMS.TPS.Common.Model.API
Path
Resolved True
Runtime Version v4.0.30319
Specific Version False
Strong Name True
Version 1.0.450.29
(Name)
Display name of the reference.
VMS.TPS.Common.Model.API.dll Properties
General Security Details Previous Versions
Property Value
Description
File description VMS.TPS.Common.Model.API
Type Application extension
File version 16.1.4.4
Product name ARIA
Product version 16.1 (CS116553)
Copyright Varian Medical Systems Finland Oy 2020
Size 367 KB
Date modified 02-12-2020 21:17
Language Language Neutral
Legal trademarks Varian Medical Systems 2020
Original filename VMS.TPS.Common.Model.API.dll
Remove Properties and Personal Information
OK
Cancel
Apply

Appendiks E: Eksempel på dokumentationsskabelon for software

Her vises et eksempel på minimumskrav til dokumentation af et stykke klinisk software. Eksemplet tager udgangspunkt i et Eclipse script som automatisk danner strukturer på et struktursæt.

Programnavn	*CreateStructures_v1.0.0.dll*
Version i produktion	*1.1.1* (MAJOR.MINOR.PATCH-formatet)
Type	ESAPI plugin script (.dll)
Navne på udviklere	Peter Petersen
Institution og kontakt	AAUH, e-mail: pp_kontakt@rn.dk, tlf.: 12213443
Godkendelse (Navn(e) og dato)	Anders Andersen, 01-02-2022
Funktion/formål med program	Programmet anvendes til at danne strukturer på et struktursæt hørende til en CT-scanning.
Klinisk berettigelse	Tidsbesparelsen ved anvendelse af programmet sammenlignet med manuelle operationer i *Contouring* er ca. 10 min for en typisk HH patient. Anvendelsen af programmet sikrer desuden at strukturnavngivningen matcher den tilsvarende optimization objective template.
Brugergruppe	Dosisplanlæggere
Input	Excelark eller tekstfil placeret i mappen: (“\\zzzz\EclipseScripts\InputFiler\”). Input skal instruere scriptet i hvilke strukturer der skal skabes og hvordan.
Output	Der skrives til ARIA databasen, i form af dannelsen af nye eller modificering af eksisterende strukturer.
Installation af program	Der kræves ingen særlige installationer. Når scriptet er approved er det klar til anvendelse i “External Beam Planning”.
Brugervejledning	Beskrivelse af programanvendelsen findes i standard inputfilen (“\\zzzz\EclipseScripts\InputFiler\Standard.xlsx”) og instruks i afdelingen kvalitetsikringssystem,
Risikovurdering	Risikotal = 4. Skriver til databasen, og danner basis for dosisplanlægningen. Se risikovurderingsdokumentet *CreateStructures_risikovurdering_01012022.pdf* placeret i “\\xx\Egen_udviklet_software\Risikovurdering”.
Kvalitetssikring/Test	• Uafhængig kode gennemlæsning af Karl Karlsen. • Unit test dækning på 80%. • Test af alle operationstyper vha. inputfilen (“\\zzzz\EclipseScripts\InputFiler\Test\test_af_alle_operationstyper.txt”) på den anonymiserede pt Id: “StructScript_test”.

	• Se dokumentet “CreateStructures_test.pdf” der beskriver de udførte tests.
Programmeringssprog	C#
Placering af kildekode	“\\xx\x\kildekode”
Placering af færdigkompileret program	“\\xx\x\Debug”
SDK (Software Development Kit)	.NET version 4.6.1 (skal være installeret lokalt)
IDE (Integrated Development Environment)	Visual Studio
Kodeopbygning	• Programmet består af 3 hoveddele: • WPF opsætning (xx.cs) • Class/klasse der indeholder metoder til indlæsning af input data (yy.cs) • Class/klasse der indeholder metoder til ændringer på det valgte struktursæt (zz.cs)
Kendte fejl og begrænsninger	• Kan ikke håndtere boolean operationer mellem strukturer med forskellig opløsning. • Brugeren af programmet skal have adgang til til mappen “XXX” på serveren “YYY”.
Periodiske opdateringer	Opgradering af ARIA: Den tilhørende nye “Eclipse Scripting API Reference Guide” informerer i starten af dokument omkring ændringer ifht. seneste ARIA version. Typisk skal man være opmærksom på hvilken version af .NET frameworket der skal anvendes.
Nødprocedurer	Hvis programmet fejler og fejlen ikke umiddelbart kan udbedres kan instruks for manuel udførelse anvendes (se: \\xx\instrukser\\)
Versionshistorik	Tidligere versioner af programmet kan findes her “\\xx\x\yyy\TidligereVersioner\”. Logbogen “CreateStructures_log” beskriver historikken i ændringerne.
Grafisk fremstilling af interaktion med omgivelsernne (f.eks. databaseadgange mv.)	Kan være simpel tegning, bare den er overskuelig
Evt. grafisk fremstilling af arkitektur/design i softwaren	Kan være simpel tegning, bare den er overskuelig