T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

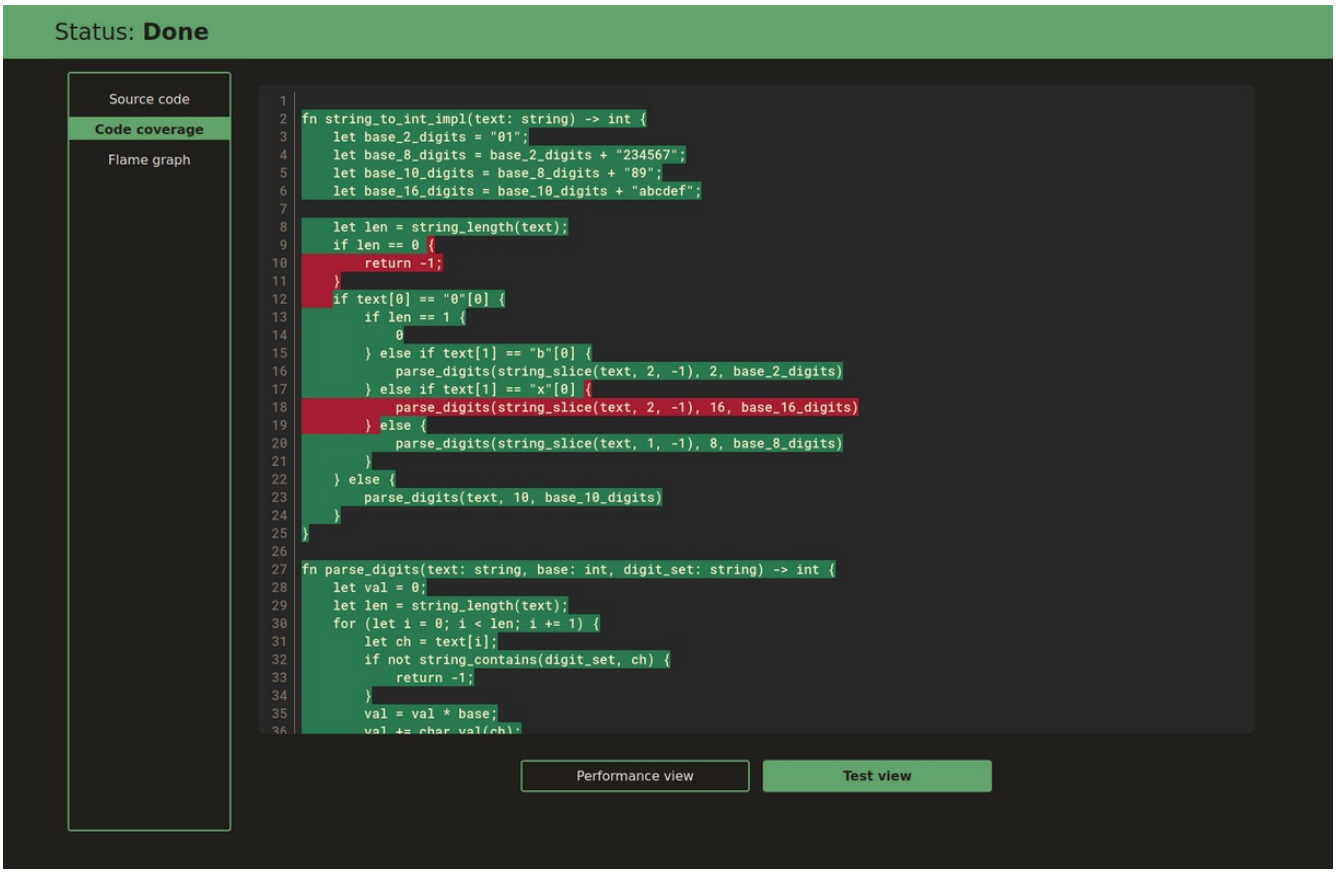
Opgavetitel	Slige – Et værktøj til udvikle software af høj kvalitet
Fag	Teknikfag – Digital design og udvikling
Emnevalg	*Ikke relevant*
Gruppemedlemmer	Theis Pieter Hollebeek Simon From Jakobsen Mikkel Troels Kongsted
Uddannelsessted	Viborg Tekniske Gymnasium – Mercantec
Dato for aflevering	20/12 2024

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Intentionelt blank

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Slige – Et værktøj til at udvikle software af høj kvalitet



T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Intentionelt blank

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Indholdsfortegnelse

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Indledning

Jo længere i digitaliseringen vi kommer, desto større behov for software, desto flere ressourcer bruges på udvikling af software. Der er mange måder at minimere ressourceforbrug i udviklingen af et stykke software. En måde er at reducere omfanget af projektet. En anden måde er at optimere udviklingsprocessen, så mængden af overhead, altså ressourcer udover det nødvendige, som bruges reduceres. I det, flere ressourcer bliver brugt på softwareudvikling, jo større bliver behovet for at optimere udviklingsprocessen. Vi vil i denne rapport udarbejde metoder, der reducerer ressourceforbruget uden at reducere omfanget.

Initierende problem

Hvordan reducerer man ressourceforbruget uden at reducere omfanget?

Problemanalyse

I første omgang skal vi dykke dybere ned i ressourceforbrug og omkostninger i udviklingenprocessen af software. For softwareudviklingsprojekter, såvel som projektstyring generelt, gælder det, at projektet er begrænset af tre indbyrdes afhængige variable: tid, omfang og omkostninger. Man kan i et hvilket som helst projekt reducere omkostningerne af projektet, i hvilket tilfælde er man så nødt til at udvide udviklingstiden eller reducere omfanget af projektet. Dette er ligeledes gældende for de to andre variable.

Dog findes der måder at reducere tiden, reducere omkostningerne eller øge omfanget uden at gå på kompromis med de to andre variable. Dette hedder processoptimering. Her er målet at gøre de specifikke processer i udviklingen mere effektiv, altså reducere mængden af overhead. Overhead er de ressourcer, dvs. omkostninger og tid, som bruges på aktiviteter, som ikke er direkte relateret til produktets udvikling. Dette kan være alt fra løn til rengøringsmedarbejdere til den ekstra tid en medarbejder bruger på at åbne deres computer. Disse ressourcer er ikke teoretisk nødvendige, men spenderes istedet på grund af specifikke omstændigheder. Ved at reducere mængden af overhead, kan man altså komme tættere på det teoretisk nødvendige behov for tid og omkostninger for et givet omfang af et projekt.

Vi benytter begreberne udviklingsomkostninger, omkostninger i udviklingsprocessen og bare omkostninger. Med disse mener vi det samme. Udviklingsomkostningerne er ikke bare de omkostninger, der nødvendigvis bruges under udviklingen af et produkt. Dette er i en sammenligning, med at betragte en udviklingsfase og en driftfase som havende seperate omkostninger associeret. Istedet betrager vi udviklingsomkostningerne, som bestående af alle omkostninger der er opstået, som konsekvens af valg og aktiviteter i udviklingsprocessen. Dette vil sige, at omkostninger i en hypotetisk driftfase, som er opstået som konsekvens af et valg eller en handling i udviklingsprocessen, er altså

inddraget i udviklingsomkostningerne. Dette har vi valgt, for at kunne betragte alle omkostninger som en enhed, istedet for at skulle skelne mellem omkostninger i driftfasen og omkostninger i udviklingsprocessen, begge som følge af valg og handlinger i udviklingsprocessen.

For at kunne behandle tid, omkostninger, omfang, osv. for softwareprojekter, skal vi specificere de præcise begreber, arbejder med. Omkostningerne er mængden af ressourcer, dvs. prisen for udviklingen af et givet produkt. Omkostningerne består primært af løn til udviklere, men består også af omkostninger for diverse værktøjer, kontorleje, osv. Fordi omkostningerne af lønnen til udviklerne er en funktion af tiden, kan tiden betragtes som indgående i omkostninger, på den måde at vi ikke betragter tiden seperat fra omkostningerne, men istedet betragter øget tid som hævede omkostninger. Dette giver også mening, i tilfælde hvor man ansættere flere udviklere for at reducere tiden. Sammenlignet med en udvikler og et udviklingsår, kan man istedet ansætte to udviklere i et halvt udviklingsår, overhead ude af beregningen. Begrebet omkostninger, vil derfor nu dække over både omkostninger og tid.

Omfanget af et softwareprojekt, som vi har brugt begrebet i modellen overnfor, dækker over produktets omfang, såvel som for kvaliteten af produktet. Produktets omfang er mængden af utility, som er bygget ind i produktet. I bredde træk for projektstyring gælder det generelt, at produktets omfang har et inverst proportionalitetsforhold til produktets kvalitet, hvis omkostninger er konstante. Det vil sige, at hvis man eksempelvis laver et produkt med stort omfang, vil det have lav kvalitet sammenlignet med et produkt med lille omfang, hvis udviklingsomkostningerne for de to produkter er ens. I softwareudvikling er denne antagelse dog ikke altid gældende. Dette er fordi, kvaliteten af software ofte påvirker mængden af overhead. Man kan altså komme ud i situationer, hvor et projekt bestående af udviklingen af et produkt med software af lav kvalitetet vil have højere omkostninger, en projekt med samme omfang, bestående af software af højere kvalitet. Grundet kvalitetens komplekse forhold til omkostningerne, vælger vi at afkoble produktets kvalitet og omfang, ved at omdefinere begrebet af projektets omfang, til kun at indebærer produktets omfang.

Kvaliteten af et softwareprodukt er et bredt begreb, vi er nødt til at pakke ud. Der er to kategorier af kvalitet. Det ene er kvalitet fra brugeres perspektiv. Det andet er kvalitet fra en faglig udviklingsmæssig betragtning, hvilket i sig selv er et kompliceret begreb, der fortjener at blive uddybet.

Hvor godt et givet produkt er fra en kundes perskektiv, vurderes ud fra en betragtning om mængden af værdi skabt for kunden (MVSK). Jo større produktets omfang, desto større værdi skabt for kunden. Produktets omfang her dækker over mængden af relevante omfang. Man kan altså skelne mellem et produkts absolute omfang, som i et softwareprodukt vil være alle features, og et produkts relevante omfang, som er det subset af features i produktet, som er relevant for den specifikke kunde. Det er relevant i en udviklingsprocess at skelne mellem absolutte og relevante omfang, men behandlingen af dette ligger udenfor rapportens ambitioner. Udover omfanget, påvirker kvaliteten af produktet også MVSK. Kvaliteten påvirker MVSK på den måde, at en lavere kvalitet vil begrænse omfangets bidrag til MVSK. Altså vil en given feature bidrage mindre til mængden af værdi skabt for kunden jo lavere dens kvalitet er. Omfanget (O) og kvaliteten (K) for et givet produkt, vil derfor påvirker mængden af værdi skabt for kunden på følgende måde: MVSK = O * K. Hvor godt et produkt er, altså mængden af

værdi skabt for kunden afhænger altså dels af produktets omfang og dels af kvaliteten, hvor kvalitet er en faktor, som begrænser omfangsets værdi.

Kvaliteten af et produkt fra kundes perspektiv, betragtes ud fra en kundes negative erfaringer med produktet. Jo flere flere negative erfaringer, desto lavere kvalitet. I et softwareprodukt kan disse negative erfanger variere bredt, eksempelvis uventet adfærd af produktet, mangel på forventet adfærd, ofteforkommende systemnedbrud, langsom interaktion, grim brugerflade, osv. Nogle kvalitetssager er tekniske problemer, eksempelvis en feature, der ikke er implementeret efter forholdene. Andre kvalitetssager er design-relaterede problemer, eksempelvis en feature, som er designet og tilrettelagt ufordelagtigt for kunden, måske endda uden kundens behov i beregningen. Kvalitetssager er ofte en kombination heraf, hvor dele spiller ind i hinanden, og hvor løsningen er et sammenspil mellem implementering og design. Forståelsen for kvalitet fra kundens perspektiv og forskellige kvalitetssager heraf er kritisk i udviklingsprocessen i et softwareprojekt, men behandling af dette ligger uden for rapportens ambitioner.

Kvalitet fra en faglig udviklingsmæssig betragtning er som sagt et kompliceret begreb. Generelt kan man sige, denne kategori dækker over elementer, som kan eller vil have en negativ indflydelse på udviklingsprocessen, dvs. øgende effekt på projektets omkostninger over tid. Man kan skelne mellem kvalitetssager der har en konkret negativ indflydelse og kvalitetssager, som kan have en negativ indflydelse. En kvalitetssag, som gør udviklingsprocessen langsommere, har en konkret negativ indflydelse på omkostningerne, da der skal bruges mere tid, på at udvikle samme produkt. Hvor en kvalitetssag, der øger sandsynligheden for systemnedbrud, kun potientielt har en negativ indflydelse, da systemnedbrudet ikke nødvendigvis indtræffer. Sager med potientelt negativ indflydelse, dvs. kvalitetssager med risiko for øget omkostninger bidrager derfor til projektets risiko for øgede omkostninger. Man kan, og vi vil, betragte risiko for øgede omkostninger som også værende en omkostning i sig selv. Derved skelner vi ikke mellem kvalitetssager der kan og kvalitetssager der har en negativ indflydelse i kvalitet fra en udviklingsmæssig betragtning.

Kvalitet fra kundens perspektiv og kvalitet fra en faglig betragtning er ikke gensidigt eksluderende kategorier. Nogle kvalitetssager om kvalitet fra kundens perspektiv kan også betragtes som kvalitetssager fra en udviklingsmæssig betragtning. Dette gælder specielt for sager med tekniske problemer. Fra en anden vinkel gælder det, at nogle kvalitetssager for kunden, specielt med tekniske problemer, er et produkt af den udviklingsmæssige kvalitet. Dog gælder det også, at nogle udviklingsmæssige kvalitetssager ikke påvirker kundens oplevelse af produktet. Vi vælger at definere kvalitet, som udviklingsmæssig kvalitet. Dette gør vi, for at inkludere alle tekniske kvalitetssager og ekskludere kvalitetssager, der kun har med design.

Altså har vi disse tre begreber om projekter i udvikling af softwareprodukter: omfang, omkostninger og kvalitet. Den naturlige tendens vil være, at reducere omkostningerne, da produktet derfor vil blive billigere at udvikle, øge omfanget, da man derved øger potientialet for værdi skabt for kunden og at øge kvaliteten, da man derved kan komme tættere på potientialet for værdi skabt for kunden.

Vi har etableret, at omfang og omkostninger er gensidigt modstridende, i det vi forsøger at øge omfanget eller reducere omkostningerne, dvs. hvis intet andet ændre sig, hvis så man prøver at reducere

omkostninger, så skal man også reducere omfanget, eller hvis man forsøger at øge omfanget, så stiger omkostningerne også. Dette giver mening, eksempelvis i tilfælde hvor man vil udvikle en ekstra feature, her vil det naturligvis være krævet, at man bruger ekstra ressourcer og tid på udvikling, og derved stiger omkostningerne.

Omfang og kvalitet har ligeledes et simpelt forhold til hinanden. Hvis intet andet ændre sig, så vil et forsøg på at hæve omfang, naturligt få kvaliteten til at falde. Dette er fordi, jo bedre produktetes kvalitet, desto mere tid og ressourcer, skal man bruge på udviklingen. Ligeledes, et forsøg på at øge kvaliteten, uden at ændre andet, vil naturligt få omfanget til at falde, da den ekstra tid og ressourcer brugt på den ekstra kvalitet, så ikke vil blive brugt på det resterende omfang. Dog gælder dette forhold kun, hvis man antager, at forholdet mellem omkostninger og kvalitet fungerer på samme vis. Dog er dette ikke nødvendigvis sandt.

Forholdet mellem omkostninger og kvalitet afviger fra de to førnævnte. Fra en overfladiskt betragtet, ville det give mening, at forholdet var magen til forholdet mellem omfang og omkostninger. Grunden til forholdet mellem omkostninger og kvalitet er, at udvikling af software af lav kvalitet kræver mere tid og ressourcer. Altså det kan koste mere, at udvikle et produkt med samme omfang, hvis produktet har lav kvalitet, sammenlignet med hvis produktet har højere kvalitet.

Dette forhold kommer af, at udviklingsprocessen bliver mere ineffektiv af, at softwaren er lav kvalitet. Der findes mange grunde og konkrete eksempler på, hvorfor lav kvalitets software øger udviklingsomkostningerne. Et sådanne eksempel kunne være, at integration af nye features er sværre, dvs. kræver mere udviklingstid, for lav kvalitets software sammenlignet med højere kvalitet, da højere kvalitets software vil være designet bedre, så mængden af arbejde det kræver at udvide softwaren, vil være mindre. Et andet eksempel er i forbindelse med kodefejl, altså tilfælde hvor softwaren opfører sig på en anderledes måde end forventet. Jo lavere kvalitet softwaren er, jo flere kodefejl, og set fra en anden vinkel, jo lavere softwarens kvalitet, jo flere hændelser vil opstå, hvor softwaren opfører sig anderledes end forventet. Dette har mange ulemper. For det første bidrager det til en ringere oplevelse for kunden. For det andet kræver det mere udviklingstid, for at forebygge uventede fejl. Det vigtigste i forbindelse med udviklingsomkostninger er, at udvidelse af software, dvs. introduktion af nye features, er sværre og tager længere tid. Dette er fordi, i det man introducerer en udvidelse til softwaren, vil man derved bruge den allerede eksisterende software på anderledes måder. Dette kan udlløse uventet opførsel i den allerede eksisterede software, hvis den allerede eksisterende software indeholder fejl, hvis forekomst vil være større i software af lav kvalitet. Det kan altså påvises, at omkostninger og kvalitet ikke nødvendigvis er modsætninger, men istedet også kan være komplementære, dvs. at fra et perspektiv vil det at øge kvaliteten også hæve omkostningerne, og det at sænke omkostningerm, vil også sænke kvaliteten, men fra et andet perspektiv kan det at øge kvaliteten også sænke omkostningerne.

Vi betragter dette forhold mellem omkostninger og kvalitet, som værende mellem omkostninger og kvalitet, sammenlignet med, hvis det var mellem kvalitet og omfang. Dette er fordi enhver indflydelse, dette forhold har på omfang, vil være et kompenserende initiativ, for at begrænse omkostningerne. Altså, hvis man i et projekt oplever stigende omkostninger på grund af lav kvalitet software, og reducerer omfanget, for at fastholde omkostningerne, her vil reduktionen af omfangen være en

sekundær effekt, altså en reaktion til kvalitetens indflydelse på omkostningerne. Derfor betragter vi dette forhold primært, som værende mellem kvalitet og omkostninger, og vi betragter forholdet mellem kvalitet og omfang primært som fælles begrænsende og sekundært ligesom forholdet mellem kvalitet og pris.

For at kunne behandle dette forhold mellem kvalitet og omkostninger, vil vi benytte begrebet overhead. Overhead i en general sammenhæng er de yderlige omkostninger, der ligger i en aktivitet, efter at selve aktivitetens grundlæggende omkostninger er dækket. Konkret i en virksomhedssammenhæng er overhead de yderligere omkostninger, virksomheden skal betale, for at udføre en service, efter at lønnen til arbejderne er betalt. Altså er overhead omkostningerne for at dække eksempelvis kontorleje, strømregninger, rengøring, værktøjer, osv. Vi vælger at benytte begrebet overhead, til at henvise til mængden af ekstra omkostninger, introduceret som konsekvens af lav kvalitet. Altså, jo større yderligere omkostninger på grund af lav kvalitet, desto større overhead. Vores begreb af overhead bygger på forståelse af, at der er en teoretisk perfekt version af softwaren man udvikler, en version med perfekt kvalitet, som fører til ingen yderligere omkostninger, og overhead (OH) er så omkostningerne for en teoretisk perfekt udviklingsprocess (TO) trukket fra de reelle udviklingsomkostninger (RO), altså: OH = RO – TO. Vi tilføjer en klarificerende definition til begrebet kvalitet, som lyder på, at kvaliteten af software afhænger af mængden af overhead i omkostningerne i udviklingen af softwaren, dvs. jo større overhead i udvikllingsomkostninerne, jo lavere er softwarens kvalitet.

Eftersom den umiddelbare tendens i et projekt er at minimere omkostningerne, vil den naturlige tendens være at minimere overhead i udviklingsprocessen. Vi har etableret, at for at minimere overhead i udviklingsprocessen, er man altså nødt til at hæve kvaliteten af softwaren. Yderligere for at minimere omkostninger ved at øge kvaliteten skal man øge kvaliteten af software uden at hæve omkostningerne.

For at kunne besvare dette spørgsmål, er vi i første omgang nødt til, at undersøge begrebet ‘øge kvaliteten af software’. For at forstå implikationen af ‘øge’, er vi nødt til at forstå et koncept af udgangspunkt og mål. For at kunne sammenligne målet med udgangspunktet, er vi nødt til at kunne vurdere kvaliteten af et stykke software. Og for at kunne inddrage dette i en process, vil det også være fordelagtigt, at kunne kvantificere kvaliteten af et stykke software.

Hvordan vurderes kvaliteten af software?

Ovenfor har vi etableret, at kvaliteten af software, kan betragtes ud fra mængden af overhead i udviklingsomkostningerne. Dette betyder altså, at man til en vis grad kan vurdere kvaliteten af software ud fra udviklingsomkostningerne. Vi kan selvfølgelig ikke lave en absolut vurdering af softwarens kvalitet ud fra overhead, da vi ikke ved, hvor stor del af omkostningerne overhead udgører. Dog kan vi bruge denne metode, til at sammenligne et projekt med andre projekter, selvfølgelig med projekternes relevans taget i betragtning.

For at undersøge denne metode yderligere, er vi nødt til at undersøge omkostningerne. Vi betragter ikke omkostningerne som en monolitisk mængde, men istedet består de totale udviklingsomkostninger af

forskellige omkostninger. For at øge præcisionen og akuratheden en kvalitativ vurdering af softwaren ud fra omkostningerne, er vi nødt til at forstå hvilke delomkostninger, den omkostning består af.

Hvilke omkostninger er associeret med software?

For det første består omkostningerne af løn til udviklerne, der udvikler softwaren. Dette udgør første kategori af omkostningerne. Dette består af diverse programmører, designerer, software-arkitekter, projektlederer, osv. Disse personers roller, er at bidrage konkret til produktet. Denne omkostning er direkte associeret til projektets omkostninger, omfang og kvalitet. Vi nævner denne kategori først, da det er den meste interessante i sammenhæng med emnet diskuteret. Dog er andre kategorier også værd at nævne. Vi vil komme tilbage til denne kategori.

Den anden kategori af omkostninger er løn til diverse support-personale. Dette kan være managers, receptionister, rengøringspersonale, osv. Personer med disse roller bidrager ikke direkte til udviklingen af produktet, istedet yder de en eller anden service afhængende af deres rolle, som støtter udviklerne, deraf navnet på kategorien. Denne omkostning, istedet for at være direkte associeret med projektets omkostninger, omfang, kvalitet, er istedet indirekte associeret med selve projektets omkostninger.

Tredje kategori af omkostninger, er diverse udgifter til eksempelvis kontorleje, licenser til diverse softwareværktøjer, indkøb og vedligeholdelse af diverse materialer og udstyr, osv. Denne kategori, ligesom førnævnte, er ikke direkte associeret med projektet i sig selv, men er til en vis grad en nødvendighed, ligesom førnævnte, for projektet.

Traditionelt, ville man betragte de to ovenstående kategorier, som den egentlige overhead. Altså, at overhead er de omkostninger, der går til andet end løn til selve dem, der yder servicen generelt, specifik udviklerne af softwaren. Behandling af denne form for overhead er udenfor denne rapports ambitioner, og vi vurderer derfor, at denne definition af overhead, ikke bidrager til rapportens emne. Af denne grund fastholder vi den førnævnte definition af overhead, som den gældende.

Den fjerde kategori er driftsomkostninger. Med en forståelse af, at der er en udviklingsfase, som er seperat fra driftfasen, vil man normalvis betrage driftsomkostninger som seperat fra udviklingsomkostninger. Dels betragter vi ikke driftfasen seperat fra udviklingsprocessen. Dette gør vi ikke, da de to såkaldte faster ofte ikke er seperate, men istedet overlapper markant. Og dels fordi de omkostninger vi betragter, er de omkostninger, vi tidligere har etableret, stammer fra valg og handlinger i udviklingsprocessen. Driftomkostninger inkluderer udgifter til eksempelvis serverhosting og hvad man kalder operations, som er alt teknisk opsætning og vedligeholdelse.

Driftomkostninger inkluderer også udgifter til diverse leje og drift af hardware til hosting og kørsel af software. Dette betegnes som digital infrastruktur eller server-infrastruktur. Server-infrastrukturen skal have kapacitet til at understøtte softwaren, som skal køre på den. Kapaciteten af infrastruktur, som softwaren kræver, er bestemt under udviklingsprocessen.

Den femte kategori er kundesupport. Kundesupport er ydelser, som går ud på, at hjælpe kunden med at bruge produktet eller løse kundens problemer relateret til produktet. Ligesom fjerde kategori for driftomkostninger, er kundesupport i hvis grad afhængigt af valg taget i udviklingsprocessen.

Den sjette og sidste kategori vi vil behandle, er omkostninger associeret med potientielle omkostninger. Potientielle omkostninger er omkostninger, der ikke nødvendigvis er skal dækkes. Dette kan eksempelvis være erstatning, der skal betales til kunder, ved systemnedbrud eller datalæk. Det kan også være ekstre løn og overarbejde, til at rette akkutte opståede problemer. Og det kan også være tabt værdi, i form af mistede kunder. Omkostninger, der går til at afbøde potientielle omkostninger, oftest i form af forsikring, dækker denne kategori også over. Dette gør den, fordi disse omkostninger er en sekunder der handling, ud fra de potientielle omkostninger. Vi betragter alle potientielle omkostninger, som en konsekvens af valg og handler i udviklingsprocessen. Derfor inddrager vi potientielle omkostninger som udviklingsomkostninger, selvom potientielle omkostninger ofte også står på, efter udviklingen af produktet er afsluttet.

Som sagt betragter vi løn til udviklerne som den væsentligste omkostning generelt. Grunden er, at alle andre omkostninger i større eller mindre grad har et forhold til denne omkostning. Hvis man betragter kategorierne som puljer, så ville mange puljer kunne op- og nedsjusteret, med en kompenserende justering af denne kategoris pulje. Specifikt for denne rapport, vælger vi at fokusere på driftomkostninger og potientielle omkostninger udover udviklingsomkostninger. Det vil fremstå nedenfor, hvorfor vi vælger disse to kategorier ud.

Hvordan påvirker kvaliteten af software omkostningerne?

For at kunne behandle og betragte kvalitet og omkostninger, er vi nødt til at undersøge, hvordan kvalitet påvirker omkostningerne for software. Ovenfor har vi beskrevende hvilke kategorier af omkostninger, der er forbundet med softwareudvilking. Hertil vil vi nu også beskrive forskellige aspekter af softwareudvikling, som kan påvirke omkostninger på grund af kvalitet.

En måde, hvorpå kvalitet kan påvirke omkostningerne, er ift. drift og krav til serverinfrasstruktur. Software af lavere kvalitet i dette aspekt, vil have større krav til serverinfrastruktur. Dette kunne eksempelvis være på grund af afviklingsprofilen af softwaren, hvor software af lavere kvalitet enten skal køre i længere tid på samme serverer eller køre på flere servere, for at opnå det samme. Dette hæver driftomkostningerne, da der kræves mere serverinfrastruktur, som skal dækkes af omkostningerne.

Et andet eksempel på hævede driftomkostninger er software, som benytter eksotisk hardware og usædvanlige platformer. Eksotiske hardwarekrav og usædvanlige platforme hæver driftomkostningerne, da driften er mere specialiseret. Dette betyder, der skal udføres mere arbejde,f for at hoste et stykke software, sammenlignet med, hvis det kørte på standard-hardware på moderne standard-platforme. Kvalitetsaspektet i dette består af, at software i udgangspunkt kan ændres og adapteres til dets behov, altså det er i udgangspunkt muligt at adaptere gammel software, til at kører på nyere hardware, men software af lavere kvalitet er sværre at ændre. Dette er, fordi software af lavere kvalitet vil være

sammensat mindre fordelagtigt, så det kræver mere arbejde at ændre. Det skal siges, at den mindre fordelagtige sammensætning ikke er på grund af lav kvalitet, men istedet er den ufordelagtige sammensætningen, i denne sammenhæng, det bidragende element til lavere kvalitet i første omgang.

En anden og ret væsentlig effekt kvalitet kan have på driftomkostninger, er i forhold til softwarens ressourceforbrug. Software af lav kvalitet, vil kræve flere ressourcer til at udføre samme opgave. Oftest vil dette ressourceforbrug betegnes i katorier, som processorforbug, hukommelsesforbug, pladsforbrug, osv. I udgangspunkt er disse katogier, incl. Fordelingen afhængigt af hvilken slags software det er og hvilke formål det skal tjene. Men udover det grundlæggende nødvendige ressourceforbrug, vil noget software bruger flere ressourcer end nødvendigt. Lavere kvalitet software vil have større unødvendigt ressourceforbrug. Hvor meget øget ressourceforbrug bidrager til driftomkostningerne, afhænger dels af, hvordan driften afregnes, hvilket der kan være forskel på. Og dels afhænger det af, hvilke konkrete slags ressourcer software har et overforbrug af. Vi kan stille følgende to eksempler op til sammenligning for at illustrere dette.

I det første eksempel kører vi et stykke software på en server. Softwaren bruger mere processortid end nødvendigt, fordi softwaren har for høj afviklingstid. Driften i eksempel serveren afregnes ud fra processortid, så mere processortid fører til større omkostninger. Processortid er i runde træk det samme som afviklingstid. Her vil vores softwares højere afviklingstid altså have en negativ indvirkning på omkostningerne, dvs. øge omkostningerne. Altså er softwares kvalitet i forhold til afviklingstid en væsentlig faktor for driftomkostingerne, når det omhandler ressourceforbrug, istedet er kvalitetens væsentlighed afhængigt af specifikke omstændigheder, og skal derfor vurderes ud fra de specifikke omstændigheder.

I det andet eksempel kører vi det samme software på en server. Softwaren har stadig det unødvendige ressourceforbrug ift. CPU-tid. Forskellen ligger i, at i dette eksempel afregnes driften ikke ud fra CPU-tid. Istedet afregnes vi for hele den fysiske maskine. Dette vil sige, at der vil blive afregnet for den samme mængde uafhængigt af, om vi bruger serverens ressourcer. Her vil det gælde, at softwarens unødvendige ressourceforbrug, så længe det totale ressourceforbrug og krav for ressourcer holder sig indenfor serverens kapacitet, ikke bidrager yderligere til driftomkostningerne. Det betyder, at i dette eksempel vil softwarens kvalitet ift. afviklingstid og ressourceforbrug generelt ikke være en væsentlig faktor for driftomkostningerne. Vi kan altså konkludere, at kvalitet ikke nødvendigvis er en væstentlig faktor i driftomkostningern

Udover driftomkostninger har kvalitet også indflydelse på den sjette kategori med potientielle omkostninger. Dette drejer sig hovedsagligt om utilsigtede hændelser, som er opstået på grund af fejl og uforventede forhold i udviklingsprocession. Dette kunne eksempelvis være systemnedbrud, hvor en fejl i softwaren, eller en hændelse, der ikke blev taget højde for, er så katastrofalt systemmæssigt, at systemet kommer i en tilstand, hvor den ikke kan tjene sit formål. I sådanne tilfælde, ville der opstå omkostninger i form af mere løn og overarbejde til udviklerne og operations, for at løse problemet og få systemet til at virke igen.

Et andet eksempel på kvalitets indflydelse på potientielle omkostninger, kan være datalæk, hvor kunders data er blevet lækket. Siden kundedata ofte er følsomme og fortrolige, kan dette medfører til

retsager og erstatning, der skal udbetales, begge kategorieseret som potientelle omkostninger. Datalæk kan opstå pga. Fejl i software og siden lav kvalitets software vil indeholde flere fejl, vil kvaliteten i denne sammenhæng være en væsentlig faktor i potientielle omkostninger.

Sidst men ikke mindst, kan lav kvalitets software være en væsentlig faktor i den første kategori af omkostninger med løn til udviklere. Dette er fordi, som vi ovenfor har etableret, har softwarens kvalitet en inflydelse på, hvor meget arbejde der er krævet af udviklere, for at udvikle softwaren. Lavere kvalitet, vil altså gøre, at den samme feature vil kræve mere udviklingstid, hvilken er ensbetydende med mere løn til udviklere.

Derved har vi etableret, at softwares kvalitet har indflydelse på forskellige kategorier af udviklingsomkostinger. For nogle af omkostningerne, har kvalitet ikke nødvendigvis en væsentlig invirkning på omkostningerne, istedet er det op til specifikke tilfælde at vurdere. For andre kategorier af omkostninger, gælder det generelt, at lavere kvalitet af software fører til øgede omkostninger.

Specifikke aspekter: afviklingstid og korrekthed

For at forstå nærmere, nogle konkrete slags kvaliteter, der kan have indflydelse på udviklingsomkostningerne, vælger vi at undersøge to specifikke aspekter for software, som kvalitet spiller en faktor i.

Den første af de to aspekter er afviklingstid. Afviklingstiden for et stykke software, er mængden af tid, det tager at kører softwaren. Hertil bruger man betegnelserne *hurtig* og *langsom* software, hvor langsom software vil have større afviklingstid end hurtig software. Ovenfor har vi beskrevet to eksempler, hvor afviklingstid henholdvist spillede og ikke spillede en væsentlig faktor for omkostningerne. Afviklingstid, som vi har unddraget til ovenfor, har større direkte indflydelse på driftomkostninger end de andre omkostningskategorier.

Det andet aspekt er korrekt. Korrektheden af et stykke software, siger noget om, hvor mange fejl softwaren indeholder. Et stykke software helt uden fejl, og hvor alt kode gør præcist, hvad det skal og hvad man forventer, vil være 100% korrekt. Korrekthed bidrager primært til potientielle omkostninger, da konsekvenserne af lav korrekthed ofte forekommer under uforventede og tilfældige omstændighder.

Vi vil i vores rapport tage udgangspunkt specifikt i disse to aspekter eller kvaliteter. Den ene, afviklingstid, som primært har indflydelse på driftomkostninger. Og den anden, korrekthed, som primært har indflydelse på potientielle omkostninger.

Hvilke problematikker forringer kvaliteten af software?

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Ovenfor har vi undersøgt softwares kvalitet med udgangspunkt i dets indflydelse på omkostningerne. Vil vil nu undersøge kvaliteterne i sig selv lidt nærmere, og hvilke problematikker i udviklingsprocessen, der er relateret til kvaliteterne.

Hvordan påvirker afviklingstid kvaliteten af software?

Hvordan påvirker korrektheden kvaliteten af software?

For hvilke slags udviklings er problematikkerne relevante?

Som vi har konkluderet ovenfor, er forskellige kvaliteter mere eller mindre væsentlige for omkostningerne, afhængigt af hvilket slags software der udvikles, og hvordan det køres i drift. Herunder har valgt seks slags softwareprojekter, og vi vil for hver undersøge, hvor væsentligt hver af de to kvaliteter er for hver.

Cloud-udvikling

I Cloud-udvikling bliver software afviklen på en server, som man lejer af en Cloud-udbyder. Ved Cloud-hosting, bliver omkostningerne afregnet ud fra afviklingstid. Dette er en generalisering, i virkeligheden er det mere nuanceret, men generalisering tjener i dette tilfælde vores forklaring. Afregning på baggrund af afviklingstid giver mening for Cloud-hosting, da Cloud-udbyderne ikke stiller fysiske serverer tilrådighed. Istedet kører de flere kunders software på samme serverer. Fordi den tid en kundes software bruger er tid, som Cloud-udbyderen ikke kan bruge på en anden kundes software, giver det mening, at afregning bliver gjot på baggrund af tiden.

Ovenfor har vi etableret, at afviklingstid og kvalitet hænger sammen. Jo lavere kvalitet, jo større er afviklingstiden. Og i dette tilfælde, jo højere afviklingstid, desto større omkostninger, hvilket altså vil sige, at jo lavere kvalitet, desto større omkostninger i Cloud-udvikling, når det omhandler afviklingstidens aspekt af kvaliteten.

Derudover er korrekthedens aspekt af kvaliteten også en væsentligt faktor. For det første foregår Cloud-hostning, som man kan regne ud fra navnet, i skyen, dvs. på internettet. Software som kører med adgang til internettet, er i større farer for hacker angreb. Hacker angreb udnytter ofte fejl i software, til

at stjæle data eller gøre en tjeneste ubrugelig. For det andet, fordi afregning foregår på baggrund af afviklingstid, er det kritisk, at softwaren afvikles som forventet. Et eksempel på, at dette kan gå galt, er hvis software ved en fejl sætter sig fast i en uendelig løkke. Dette kan hæve afviklingstiden fra, hvad der eksempelvis kunne være millisekunder, til minutter eller timer, indtil en operations-medarbejder opdager og stopper den løbske server-instans. Derved bidrager lav kvalitet også i væsentlig grad til potientielle omkostninger.

Altså har kvalitet, overdnet set, en væsentlig indflydelse på omkostninger ved Cloud-udvikling, både i forhold til afviklingstid og korrekthed.

Embedded-udvikling

Ved embedded udvikling kører softwaren i udgangspunkt i et begrænset miljø. Dette kan være forskellige begrænser, såsom færre ressourcer end man normalvis ville have, men det kan også være noget som tilgængelighed til hardwaren. Et eksempel på begrænset tilgængelighed, kan være, hvis produktet, softwaren inkluderet, er et fysisk produkt, som installeres ude ved kunden uden adgang til internettet.

I miljø med begrænset adgang, vil softwarens korrekthed være kritisk, da rettelser af fejl er omkostningsfulde at efterinstallere, efter produktet er leveret hos kunden. Under normal omstændigheder, for at sammenligne med embedded-miljøer, hvor updateringer er nemme at rulle, vil det ikke være kritisk i samme grad, at softwaren har høj korrekthed og kvalitet generelt fra starten. Der vil det i mange tilfælde være acceptabelt, hvis softwaren i starten har lav kvalitet i hvisse aspekter, der så forbedres over tid. Men i et embedded-miljø, er det som sagt kritisk, at softwaren har høj kvalitet før den installeres. Dette tilføjer et problem, vi ikke har behandlet i sig selv hidtil. Problemet er, at udviklerne skal have en vis ide om, kvaliteten af deres software.

Kvalitet spiller derfor en væsentlig rolle i embedded-udvikling, både i aspekterne af afviklingstid og korrekthed. Og derudover, er der problematikken med, at udviklerne er nødt til, at have en vis vished om kvaliteten af softwaren, før den udgives.

Desktop-udvikling

I desktop-udvikling udvikler man software, til at køre på almindelige stationære og bærbare PC’er. Afviklingstid er i udgangspunkt ikke en væsentlig faktor for udviklingsomkostninger, da det oftest er kunden selv, der er ansvarlig, for at have tilstrækkeligt hardware. Derudover er moderne hardware oftest tilstrækkelig. Langsom software vil i udgangspunkt gå mest ud over kvaliteten fra kundens perspektiv, og ikke bidrage til øgede udviklingsomkostninger.

I desktop-udvikling kører software ofte lokalt på computere. Her vil softwaren derfor ikke være tilgængelig via. internettet i samme grad som eksempelvis Cloud-udvikling. Udover det, er desktop-

miljøer ofte forbundet til internettet, så softwaren kan opdateres løbende. Derfor er korrekthed ikke kritisk for desktop-udvikling, sammenlignet med Cloud- og embedded-udvikling.

Mobiludvikling

For mobiludvikling, dvs. udvikling af apps til smartphones og andre mobile enheder, gælder samme krav oftest for korrekthed som ved desktop-udvikling. En smartphone er forbundet til internettet, så opdateringer kan installeres løbende, men softwaren er ikke umiddelbart tilgængelig udefra, så korrekthed forbindelse af sikkerhed er ikke kritisk, sammenlignet med Cloud-udvikling.

For mobil- og desktop-udvikling gælder det ofte, at det ikke er programmet selv, der håndtere persondata og lignende. Det deligeres ofte til en tjene i skyen. Dette aspekt gør korrekthed endnu mindre væsentligt for omkostningerne.

Webudvikling

Webudvikling består ofte af to dele. Den ene del omhandler, det som kører på brugerens computer. Den anden del er den, som kører på serveren og levere data til første del. Ofte til anden del hostes i Cloud-miljø, så vi kan derfor betragte denne anden del, som Cloud-udvikling i sig selv. Første del, der kører på brugerens computer, håndtere ofte ikke følsomme data. Desuden er det muligt opdatere software meget nemt. Hvis denne første del bryder sammen, er skaden ofte ikke større, end at brugeren kan trykke på opdater-knappen og prøve igen. Ud fra dette kan vi altså konkludere, at korrekthed ikke er kritisk.

Hvordan håndterer softwareudviklere disse problematikker?

Ovenfor har vi kigget på, hvordan de to aspekter af kvalitet, har indflydelse på forskellige slags udvikling. Vi vil nu kigge på, hvordan udviklere håndtere disse problematikker.

Hvordan håndterer softwareudviklere problematikker relateret til afviklingstid?

Når det kommer til afviklingstid, har softwareudviklere nogle forskellige værktøjer de kan anvende. For det første, tager udvilklere et valg om, hvilke teknologier, deres løsning implementeres i. Nogle teknologier vil per automatik fører til bedre-ydende programmer. Nogle værktøjer til implementering, tænk programmeringssprog og værktøjerne associeret, vil bruge mere afviklingstid på at afvikle det samme program. Dette er ofte fordi, sådanne værktøjer udføre flere operationer end andre.

En anden måde for udviklere at håndtere problematikker relateret til afviklingstid er performance-optimering. Performance-optimering er en teknik eller aktivitet, hvor udvikleren omskriver sektioner af et program, til at yde bedre. Optimering i denne sammenhæng skal forstås som, at vi forsøger at minimere afviklingstiden, altså forbedre programmet. Den matematiske definition af optimering er lidt anderledes, da det der handler om, at finde den optimale værdi, og ikke bare tilnærme os det optimale. Vi bruger altså den første definition. Denne process, specifik for afviklingstid i dette tilfælde, består abstarkt af følgende: 1) mål afviklingstiden for programmet, 2) identificer hvilke sektioner, der har størst indflydelse på afviklingshastighed, 3) ret koden, for at reducere afviklingshastighed og 4) mål om rettelsen havde en positiv indflydelse. Denne løkke kører udvikleren igennem, til de er tilfredse med resultatet.

Hvordan håndterer softwareudviklere problematikker relateret til korrekthed?

Når det kommer til korrekthed, har udvikleren nogle andre værktøjer, de kan bruge. Det første værktøj, ligesom ovenfor, er teknologierne, hvori løsningen implementeres. Nogle teknologier er designet, så de er bedre end andre, til at håndtere korrekthed.

Et andet værktøj er manuel testing. Denne metode går ud på, at en udvikler, eller en anden medarbejder for den sags skyld, manuelt afprøver programmet, for at se, om programmet gør, hvad man ville forvente.

En tredje metode er automatisk testing. Her skriver udvikleren et sekundært program, til at teste det primære program. Dette kaldes Unit-tests, hvis testene dækker en enkelt komponent, altså en “unit” (enhed) af programmet, Integration-tests, hvis testene dækker flere komponenter samtidig og End-to-end-tests, hvis testen tester programmet med alle komponenterne.

Det skal siges, at der findes en pluralitet af metoder ud over de førnævnte. Vi har valgt disse, da vi vurderer, de er mest relevante for vores problemstilling.

Sammenligning af metoder

Til begge kvalitetsaspekter har udvikleren valget om implementeringsværktøj. Det er det med implementeringsværktøj, at det ikke er nemt at skifte, efter dele af produktet allerede er implementeret. Altså er det et up-front valg, udvikleren tager. Der er også det, at ethvert implementeringsværktøj kan bruges mere eller mindre hensigtsmæssigt. Det er derfor muligt, at vælge et implementeringsværktøj, som normalvis ville producere højere kvalitet, at producere lavere kvalitets software. På samme måde kan værktøjer, som man normalvis ville betragte, som at producere lav kvalitets kode, ofte også anvendes, til at producere høj kvalitets kode.

Hvis man sammenligner metoden med valg af implementeringsværktøj med metoden med performance-optimering, når det kommer til at minimere afviklingshastighed, så kan man konkludere, at de to metoder ikke er modsætninger. Istedet kan disse metoder anvendes komplimentært. Processen i at performance-optimere, er en process til at udnytte implementeringsværktøjerne bedre. Forskellen på metoden med værktøjsvalg og metoden med performance-optimering er, at ved værktøjsvalg er reduceringen essentielt automatisk. Udvikleren skal betrage sin egen kode, for at anvende denne metode. Men med performance-optimering er det krævet af udvikleren, at de identificere dele af koden og omskriver delene, så de benytter mindre afviklingstid.

Til korrekthed har udvikleren tre metoder. Den første er magen til førte metode for afviklingstid, altså valg af værktøj til implementering. Nogle implementeringsværktøjer hjælper udvikleren i større grad med at undgå bestemte slags fejl, eller at håndtere bestemte slags fejl, så konsekvenserne minimeres. Et eksempel på sådanne slags fejl, er hukkommelsesfejl. Disse fejl opstår, når programmet feljagtigt forsøger at anvende ressourcer, som ikke er allokeret, eller også at sådanne ressourcer ikke de-allokeres, når programmet er færdig med dem. Sådanne fejl kan sænke korrektheden markant. Nogle kodeværktøjer har funktionalitet, så værktøjet kan opdage på forhånd, de steder udvikleren bruger ressourcer forkert og afvise koden. Andre værktøjer har funktionalitet, hvor de ikke kan opdage på forhånd, om udvikleren misanvender ressourcer, istedet har de funktionalitet, så fejlende kan behandles bedre, hvis de opstår. Der findes også værktøjer, som ikke har sådanne funktionalitet. Her er der udviklerens egen opgave, at sørge for, udvikleren ikke misanvender ressourcer.

Der er ofte fordele og ulemper ved forskellige implementeringsværktøjer, specielt i eksemplet forklaret ovenfor med ressourcehåndtering. De værktøjer der automatisk håndtere ressourcefejl, bruger ofte mere afviklingstid. Værktøjerne, der opdager fejlagtig ressourcehåndtering på forhånd, øger ofte udviklingstiden. Tilgengæld gælder det ofte for væktøjerne, der ikke håndtere fejl med ressourcehåndtering, at de har lavere afviklingstid. Derfor afhænger valget også af omstændighederne.

Vi vil også sammenligne de to andre værktøjer til korrekthed, den anden, manuel testing, og den tredje, automatisk testing. Manuel testing består af mere eller mindre subjektive vurderinger af udvikleren, om programmet er korrekt. Der er to måder, dette ofte bliver gjort på. Det ene er gennemlæsning af koden. Det andet er afprøvning af programmet. Ved automatisk testing, skriver udvikleren et stykke kode, som helt konkret tester, at programmet giver det korrekte resultat, med et givet input. Tilgengæld kræver automatisk testing ekstra kode, sammenlignet med manuel testing.

Vurdering: værktøjer

Til performance-optimering er der to umiddelbare problematikker. Den første er, hvordan identificeret kode burde omskrives. Dette kræver ekspertise fra udviklerens side. Det lægger uden for rapportens ambitioner, at undersøge dette nærmere. Den anden er at identicere de sektioner af kode, med problematisk afviklingstid. Dette kan en udvikler gøre, vha. manuel vurdering, dog kan dette være ukorrekt og upræcists. I rapportens indledende undersøgelser, undersøger vi udvikleres evne til, at lave denne vurdering manuelt. Denne identificering kan også gøres automatisk med et værktøj. Et værktøj vil kunne fortælle præcist, hvilket kode der bliver afviklet hvor meget. Dette kan i stor grad hjælpe

udvikleren, til at identificere problematisk kode, og udelukke andet kode. Derfor vurderer vi, det er værd, at benytte et værktøj, til at identificere kodesektioner med høj afviklingstid i processen af performance-optimering.

Til at håndterer korrekthed, beskrev vi to metoder. Den første er manuelt testing. Vores hypotese er, at med manuel testing er der stor risiko, for at overse fejl. Det er svært for en menneskelig tester, at udføre den samme test to gange på samme måde. Vi vælger derfor, at arbejde videre med, at manuel testing ikke er tilstrækkeligt, hvis korrekthed er en prioritet.

Den anden metode er automatisk testing. Vi har forklaret, at dette involvere, at lave små testprogrammer, som henholdsvis hedder Unit-, Integration- og End-to-end-tests, afhængigt af hvor stor del af koden de tester. Der er en problematik ved automatisk testing. Det er, at automatiske test-programmer, også kaldet test-suites eller bare tests, ofte ikke tester alt koden. Altså, tests afvikler koden, for at teste, at koden gør det rigtige. Hvis noget af koden ikke afvikles af tests, bliver dette kode derfor ikke testet. Hvor meget af koden tests tester, kalder man ofte code-coverage, altså kodedækkelse, dvs. hvor meget af koden, testen dækker. Nedenfor har vi beskrevet en undersøgelse, vi har udført, som undersøger udvikleres evne til manuelt at vurdere code-coverage.

Hvilke værktøjer kan udviklere anvende, for at øge kodekvaliteten?

Vi vi nu kigge på, hvilke værktøjer en udvikler kan anvende, til at assisterer med de metoder, vi beskrev ovenfor.

Hvilke værktøj kan anvendes ift. afviklingstid

Til at reducere afviklingstid, har vi ovenfor etableret, at man ville bruge performance-optimering. En del af performance-udvikling, har vi etableret, er identificering af problematisk kode. Vi har også etableret, at man kan benytte værktøjer til, at hjælpe med at identificere problematisk kode. Værktøjet, som vi vil præsentere, er en flame-graph. En flame-graph er et værktøj, som producere en oversigt af et program, hvor man kan se, hvilke af programmets funktioner optager hvor stor del af programmets afviklingstid. Med et flame-graph-værktøj er det muligt, at se hvilke dele af koden, der har størst indflydelse på afviklingstid. Ud fra dette, kan man nemmere identificere hvilke dele af koden, man burde inddrage i performance-optimering, og hvilke dele man burde eksludere.

Hvilke værktøjer kan anvendes ift. korrekthed?

Til at håndtere korrekthed har vi ovenfor etableret, at man ofte vile bruge automatisk testing. Derudover har vi etableret, at en problematik ved automatisk testing, er at vurdere code-coverage for

tests over koden. Et værktøj man kan bruge til at assistere med dette, er et code-coverage-værktøj. Et code-coverage-værktøj følger med i afviklingen, når man afvikler koden. Her tæller den så, hvor mange gange, hvert stykke kode bliver kørt. Resultaten viser den så til udvikleren, som så kan aflæse, hvilke dele af koden, testene ikke tester.

Vi har nu etableret, at der er omkostninger i forbindelse med softwareudvikling. Inkluderet i omkostningerne er også overhead. Dette overhead er relateret til den generelle kvalitet af softwaren. Vi har undersøgt, hvordan denne kvalitet kan påvirke specifikke slags softprojekter. Derudover har vi studeret to aspekter af kvalitet nærmere. Disse to aspekter er afviklingtid og korrekthed. Vi har undersøgt metoder, hvorpå udviklere kan arbejde med disse problematikker relateret til disse to aspekter. Metoderne er performance-optimering for afviklingstid og testing for korrekthed. Vi har undersøgt, hvilke værktøjer, man kan bruge til at assistere med disse metoder. Vi har udvalgt to værktøjer, flame-graph-værktøj til afviklingstid og code-coverage-værktøj til korrekthed. Vi vil nedenfor undersøge, hvordan man kan lave sådanne værktøjer, derfor lyder vores problemformulering som følgende.

Målgruppeanalyse

For at kunne lave tilstrækkelige værktøjer, er vi nødt til at undersøge værktøjets målgrupper. I udgangspunkt er målgruppen softwareudviklere, specifik programmører. Programmører vil allerede have grundlæggende forståelse for koncepterne kode, korrekthed og afvikling. Programmører, som vi vurderer det, behøver ikke uddybende forklaring, i koncepterne bag værktøjerne. Istedet er det vigtig for programmører, at de kan bruge værktøjerne effektivt.

En anden målgruppe er ikke-tekniske medarbejdere. Med ikke-tekniske medarbejdere mener vi, medarbejdere, som ikke er programmører, og derfor ikke har samme grundlæggende forståelser for de grundlæggende koncepter. Det kan i nogle tilfælde give mening, at ikke-tekniske medarbejdere også kan bruge disse værktøjer. Eksempelvis vil det der være muligt, at uddelegere en kvalitetsundersøgelse til en ikke-teknisk medarbejder, hvis værktøjerne ikke kræver programmeringsteknisk forståelse.

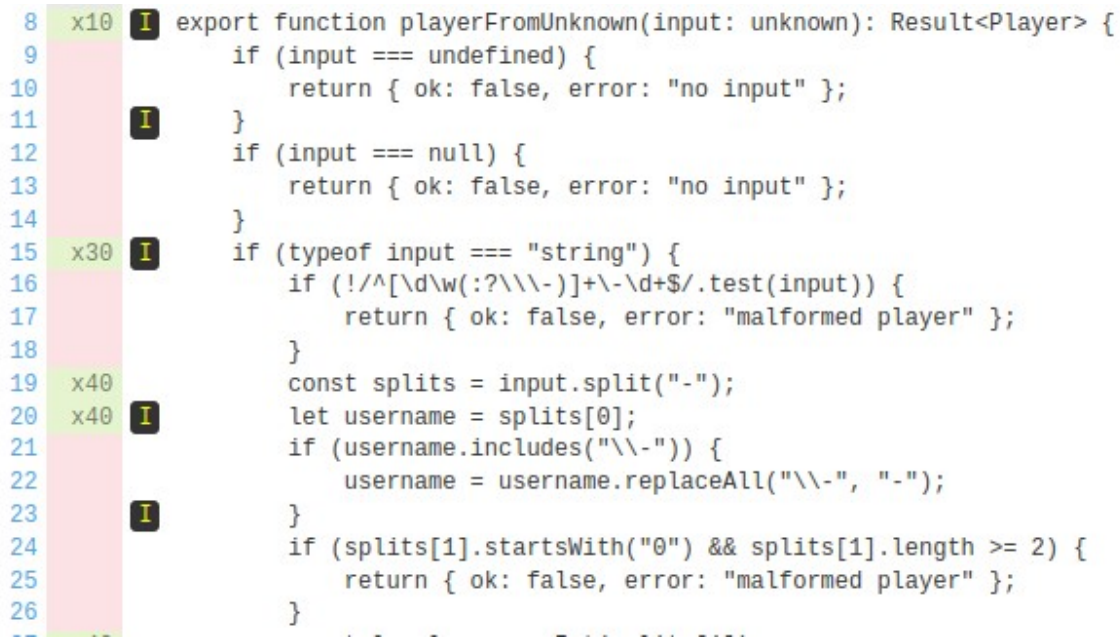
Altså vil vores primære målgruppe være programmøre, og så har vi en sekunder målgruppe, som er ikke-tekniske medarbejdere.

Markedsanalyse

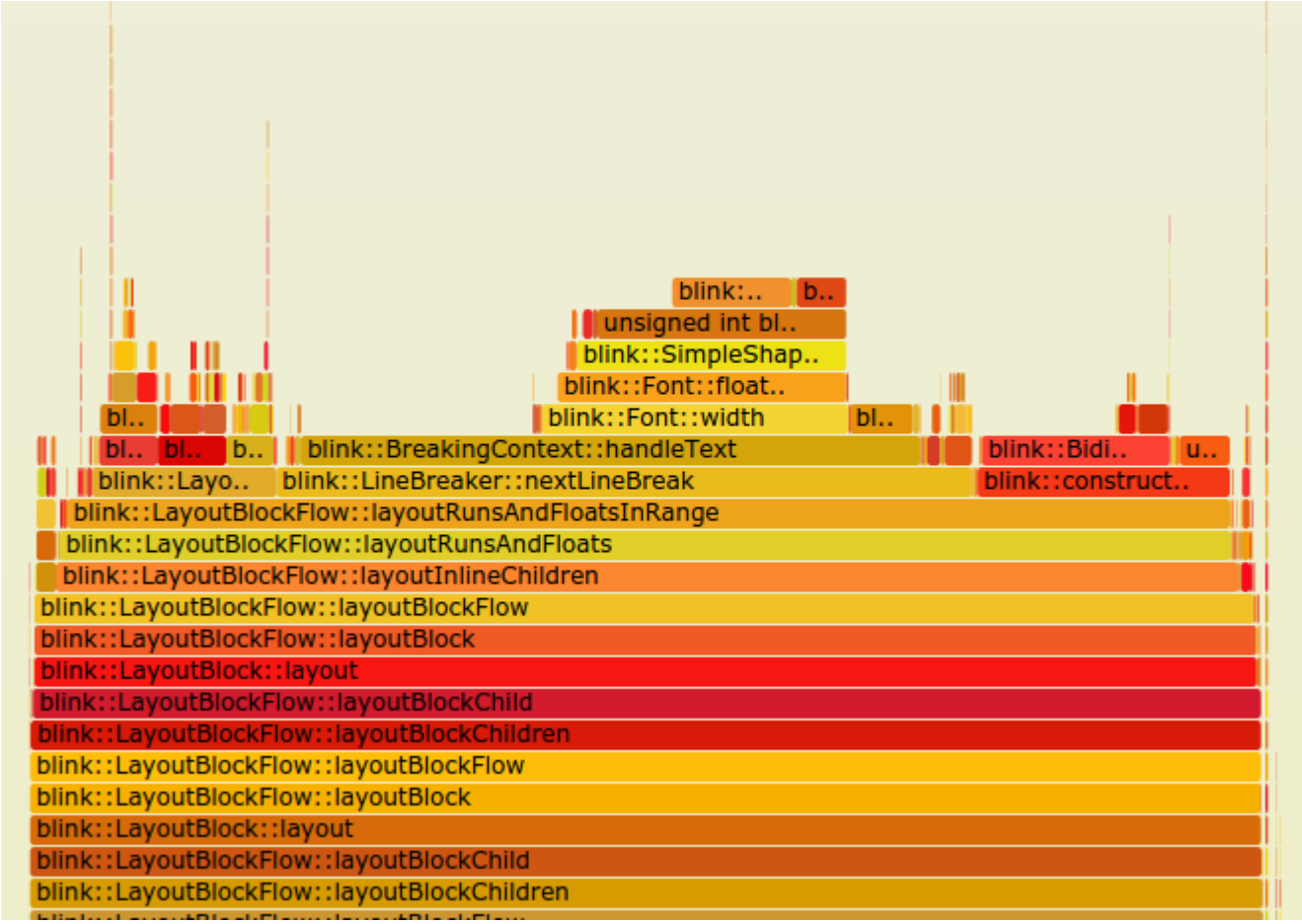
Der findes eksisterende code-coverage-værktøjer og flame-graph-værktøjer.

Code-coverage-værktøjer er ofte en del af tredjeparts testing-frameworks, som man inddrager i ens projekt. Ofte bygger disse værktøjer oven på funktionalitet i programmeringssproget eller afviklingsmiljøet. Et eksempel på en tech-stak, altså en samling teknologier, er programmeringssproget Typescript og afviklingsmiljøet Deno, som også inkluderer et testing-framework. Deno bidrager, udover at kunne afvikle koden, med funktionalitet til at skrive og afvikle automatiserede tests til Typescript-kode. Denos testing-værktøj har indbygget funktionalitet til at generere en *coverage report*,

som er en code-coverage-analyse, som man kan få vist grafisk i en browser. Her viser den det af programmets kode, som er under test, hvor alt kode, der ikke afvikles er vises med rød margin.



Der findes også eksisterede flame-graph-værktøjer. Et eksempel er Perf-værktøjet på Linux. Dette værktøj kan sættes til at analysere et kørende native-program, dvs. et program der kører direkte på CPU'en. Ud fra dette kan den genererer en call-stack-analyse, som man kan få vist som en flame-graph-analyse i en browser som Firefox.



De eksisterende værktøjer vi har analyseret, består af sammensætninger af forskellige teknologier. Udover det, kræver værktøjerne, specielt Perf, en del opsætning, for at komme til at virke. Hvis man har lavet et program, er det markant mere arbejde, hvis man derefter vil lave code-coverage- eller flame-graph-analyser af programmet med de teknologier, vi har undersøgt og præsenteret ovenfor.

Undersøgelser

Indledende undersøgelser

Undersøgelse af udvikleres evne til at vurdere afviklingstid af kode uden værktøjer

Vi ønsker at undersøge behovet for værktøjer som flame graph til identificering af problematiske dele af programmets kode i forhold til afviklingstid. Dette vil vi gøre ved at undersøge en gruppe softwareudvikleres evne til at udføre denne proces uden værktøjer som flame graph.

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Undersøgelsen er konstrueret på følgende måde. Hver af udviklerne i vores testgruppe, fik til opgave at vurdere afviklingstiden i et udleveret stykke kode.

Følgende er koden de blev præsenteret for.

```
fn add(a, b) {
   + a b
}

fn main() {
   let result = 0;
   let i = 0;
   loop {
      if >= i 10 {
         break;
      }
      result = add(result, 5);
      i = + i 1;
   }
}
```

Følgende er resultaterne for undersøgelsen.

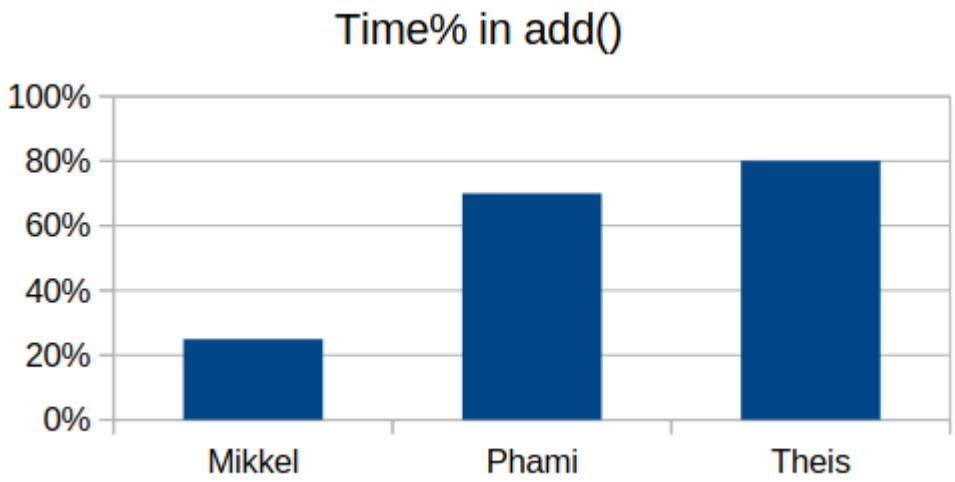
Programmer	**Time% in main()**	**Time% in add()**
Mikkel	95%	25%
Phami	100%	70%
Theis	100%	80%
Average	98%	58%
Avg. Err.	1%	47%

Følgende er resultatet af en flame graph-rapport vi generede ud fra koden.

Time% in main()	**Time% in add()**
97,7%	11,7%

Grunden til at tiden i **main()** ikke er 100%, er fordi programmet laver noget opsætning og oprydning før og efter **main()** afvikles.

Interessant her, er testgruppens gæt på mængden af afviklingstid brugt i **add()**-funktionen.



Her kan det ses, at gættene varierer meget. Der er en forskel på 80% - 25% = 55% mellem det højeste og det laveste gæt. 66% af testgruppen har fejlvurderet hvilken del af koden, der er mest afviklingstid i.

Vores testgruppe var ikke særlig stor, og vi har ikke vurderet niveauet af hver udvikler. Vi præsenterede kun udviklerne for et enkelt program, som i sig selv ikke repræsenterer et virkelighedsnært program både i størrelse og funktion. Testgruppen fik minimal introduktion før deltagelse, det fremstod eksempelvis meget uklart for udviklerne, hvorfor 100% af tiden ikke blev brugt i main().

Alligevel giver dette en indikation af, at udviklere i stor grad har svært ved, at vurdere hvilke dele af koden, der benytter mest afviklingstid, og dermed har de svært ved at vurdere, hvor de skulle anvende performance-optimeringsproces, hvis afviklingstiden skulle formindskes.

Undersøgelse af udvikleres evne til at bedømme code coverage uden værktøjer

Vi vil også undersøge, i hvilken grad en softwareudvikler er i stand til at vurdere code coverage. Dette vil vi undersøge, ved at give en testgruppe udviklere et program, som de skal skrive en softwaretest til. Vi vil derefter bruge et code coverage-værktøj til at undersøge, hvor stort udslag der er i udviklernes evne til at vurdere code coverage uden værktøj.

Undersøgelser er konstrueret på følgende måde. Hver udvikler i testgruppen blev præsenteret for 2 programmer. De skulle herefter lave en software test, hvor udviklerne blev instrueret til for det første, at teste programmets korrekthed og for det andet maksimere code coverage af testen. Her er koden programørerne blev præsenteret for:
Program 1:

```ts
export type Player = {
    username: string;
    level: number;
};

export type Result<T> = { ok: true; value: T } | { ok: false; error: string };

export function playerFromUnknown(input: unknown): Result<Player> {
    if (input === undefined) {
        return { ok: false, error: "no input" };
    }
    if (input === null) {
        return { ok: false, error: "no input" };
    }
    if (typeof input === "string") {
        if (!/^[\d\w(:?\\\-)]+\-\d+$/.test(input)) {
            return { ok: false, error: "malformed player" };
        }
        const splits = input.split("-");
        let username = splits[0];
        if (username.includes("\\-")) {
            username = username.replaceAll("\\-", "-");
        }
        if (splits[1].startsWith("0") && splits[1].length >= 2) {
            return { ok: false, error: "malformed player" };
        }
        const level = parseInt(splits[1]);
        return { ok: true, value: { username, level } };
    } else if (typeof input === "object") {
        if (Object.keys(input).length < 2) {
            return { ok: false, error: "malformed player" };
        }
        if (Object.keys(input).length > 2) {
            return { ok: false, error: "too much information" };
        }
        if (!("username" in input && "level" in input)) {
            return { ok: false, error: "malformed player" };
        }
        if (typeof input.username !== "string") {
            return { ok: false, error: "malformed player" };
        }
        if (typeof input.level !== "number") {
            return { ok: false, error: "malformed player" };
        }
        const { username, level } = input;
        return { ok: true, value: { username, level } };
    } else {
        return { ok: false, error: "malformed player" };
    }
}
```

Program 2:

```ts
const charVal: {[key: string]: number} = {
    "0": 0, "1": 1, "2": 2, "3": 3,
    "4": 4, "5": 5, "6": 6, "7": 7,
    "8": 8, "9": 9, "a": 10, "b": 11,
    "c": 12, "d": 13, "e": 14, "f": 15,
};
const base2Digits = ["0", "1"];
const base8Digits = [ ...base2Digits, "2", "3", "4", "5", "6", "7"];
const base10Digits = [ ...base8Digits, "8", "9"];
const base16Digits = [ ...base10Digits, "a", "b", "c", "d", "e", "f"];

export function stringToInt(text: string): number {
    if (text.length === 0) {
        return NaN;
    }
    if (text[0] === "0") {
        if (text.length === 1) {
            return 0;
        } else if (text[1] == "b") {
            return parseDigits(text.slice(2), 2, base2Digits);
        } else if (text[1] == "x") {
            return parseDigits(text.slice(2), 16, base16Digits);
        } else {
            return parseDigits(text.slice(1), 8, base8Digits);
        }
    }
    return parseDigits(text, 10, base10Digits);
}

function parseDigits(
    numberText: string,
    base: number,
    digitSet: string[],
): number {
    let value = 0;
    for (const ch of numberText) {
        value *= base;
        if (!digitSet.includes(ch)) {
            return NaN;
        }
        value += charVal[ch];
    }
    return value;
}
```

Resultatet var, at der på det første program var gennemsnitlig code coverage, altså gennemsnittet af hver udviklers test på 80% og 90% i andet program. Dette vil altså sige, at henholdsvis 1 - 80% = 20% og 1 - 90% = 10% i gennemsnit af hvert program blev ikke testet. I alt er resultatet i undersøgelsen, at 15% kode ikke bliver testet, når udviklere skriver tests.

Resultaterne kan dog være lidt misvisende i denne undersøgelse på grund af visse fejlkilder. Programmerne var små eksempel programmer udviklet for sig, og dermed er det muligt, de ikke er virkelighedsnære. Udviklerne i testgruppen har haft begrænset tid til at udføre opgaven, og det ville være rationelt at tænke, at deres resultater i stor grad afspejlede mængden af tid til opgaverne.

Vi kan herudfra konkludere, at der er en vis sandsynlighed for at code coverage ikke er 100% i tests, som softwareudviklere generelt bruger til at teste deres kode. Vi kan også konkludere, at der kan være en vis sammenhæng mellem tid brugt på at undersøge tests code coverage og og testenes code coverage. Derfor ville det give mening at benytte et værktøj, som automatisk kunne generere en code coverage-rapport til udvikleres tests.

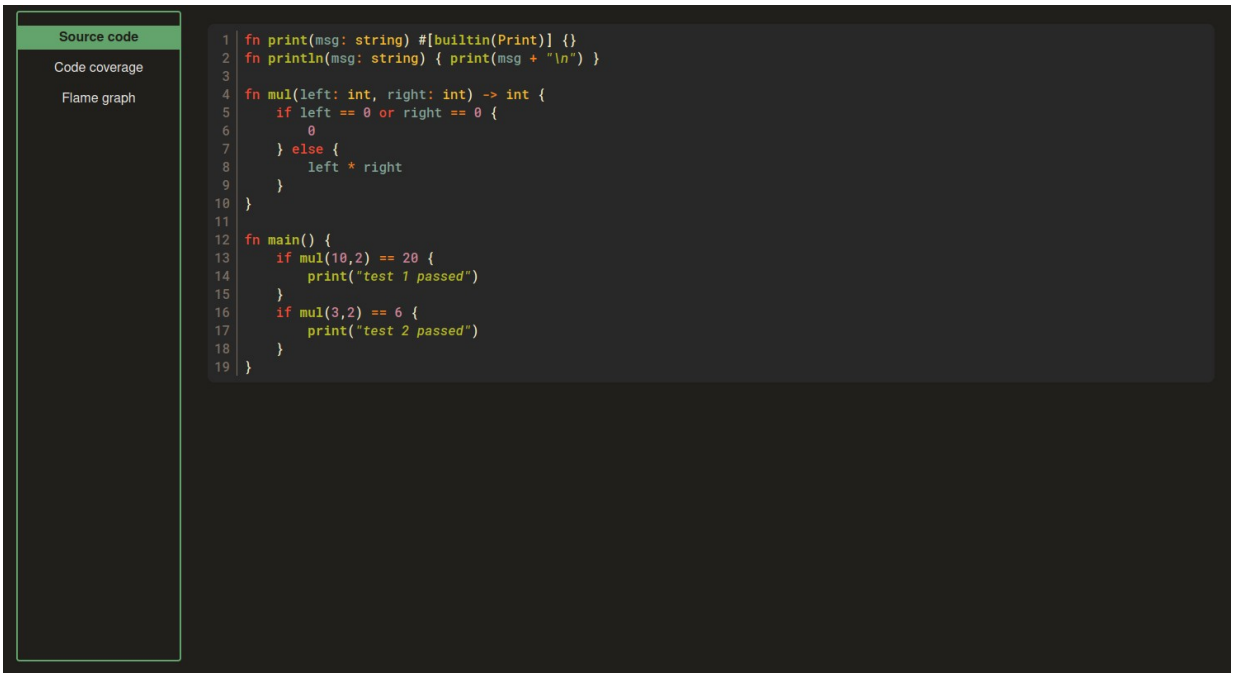
Problemformulering

Hvordan kan man lave et værktøj, til at øge kvaliteten af software?

Design

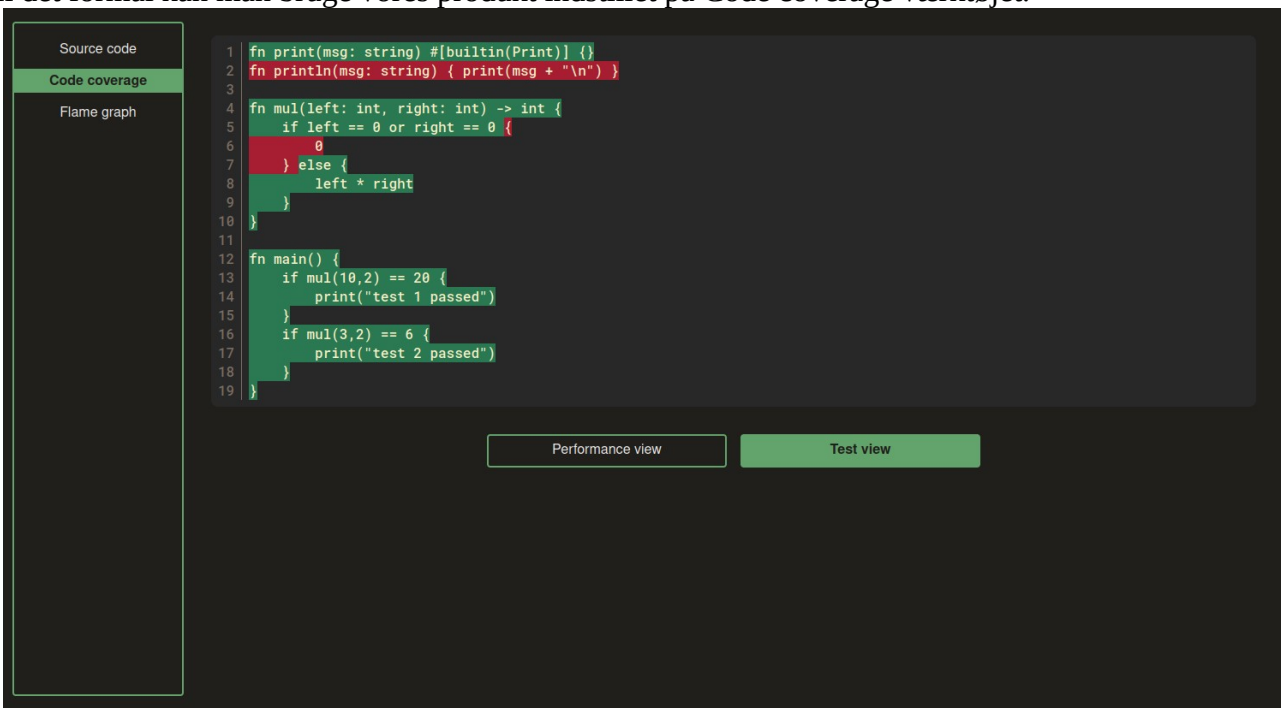
En udvikler har et program, som han har skrevet. Hans program kører lige nu meget langsomt, og han er usikker på om hans kode er korrekt, og han vil derfor gerne have nogle anbefalinger til, hvordan han kan forbedre det.

Med vores produkt indstillet til at bruge Source Code værktøjet, kan en udvikler få overblik over deres program med syntax highlighting og linjetal, som hjælper med at kunne analysere koden i hånden.



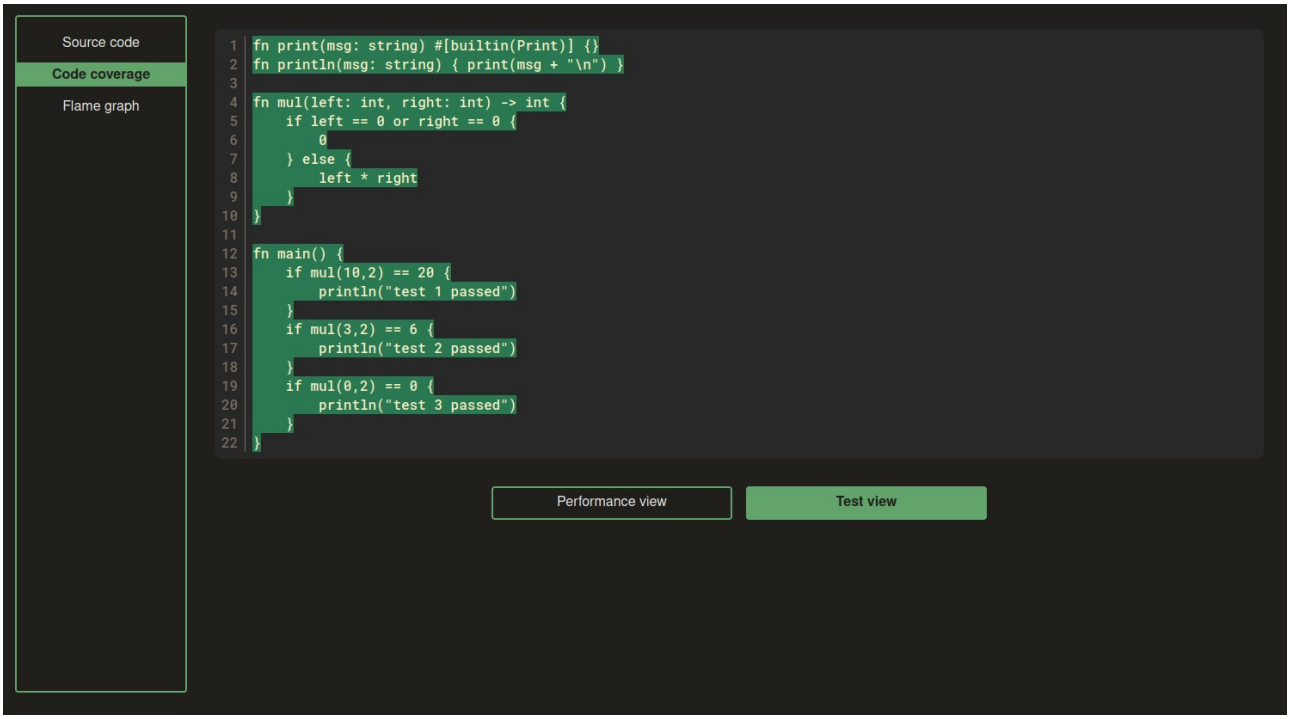
Det er dog ikke altid, at analysere i hånden er godt nok. Udviklere skriver ofte derfor det man kalder unit tests, et kort stykke kode der tester at andet kode er ordenligt implementeret; deri ligger der et problem: Hvordan sikrer man at en testkode grundigt dækker alle muligheder i programmet?

Til det formål kan man bruge vores produkt indstillet på Code coverage værktøjet.



Vi kan her se, eftersom at det er markeret i rødt, at udvikleren aldrig tester, hvad der sker, hvis en af inputsne er 0; derudover kan udvikleren også se, at de aldrig bruger **println** funktionen.

Udvikleren kan nu derfor skrive en test, og huske at bruge **println** funktionen, og bruge produktet igen:

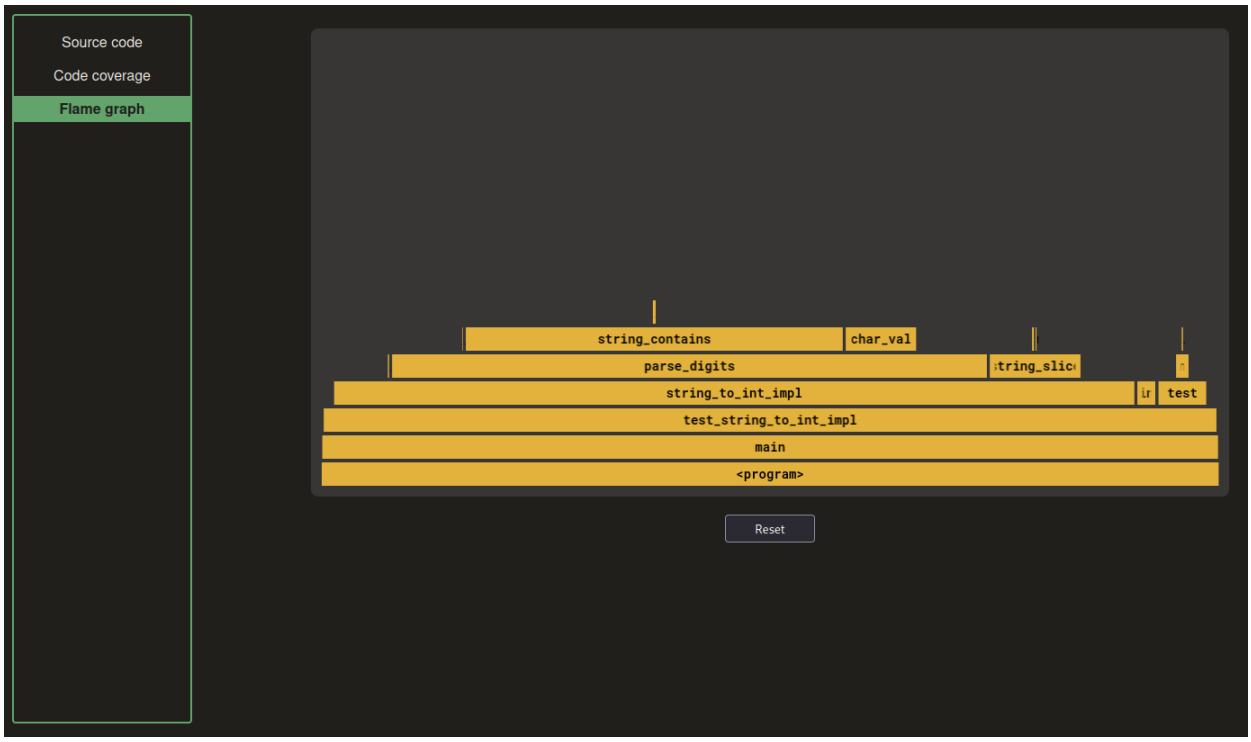


Udvikleren kan nu være sikker på, at deres kode virker som den skal, eftersom deres tests dækker alle mulighederne.
Til sidst er der optimering af hastighed. Til det kan man bruge vores produkt indstillet på ”Flame graph” indstillingen.

Den viser en flame graph, som beskriver, hvordan programmet bruger dele af dens afviklingstid.

Man kan her se, at størstedelen af tiden bruges i **mul** funktionen, og næststørstedelen bliver brugt i **println**. Det giver her derfor mening at **mul** funktionen er den første kandidat til at blive optimeret.

Et mere kompliceret eksempel er draget ud fra dette program, som konverterer tekst til tal:



Her kan man se, at det er smartest at optimere **string_contains** først, da det er den, som programmet bruger størstedelen af tiden på.

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Gennemgang af design

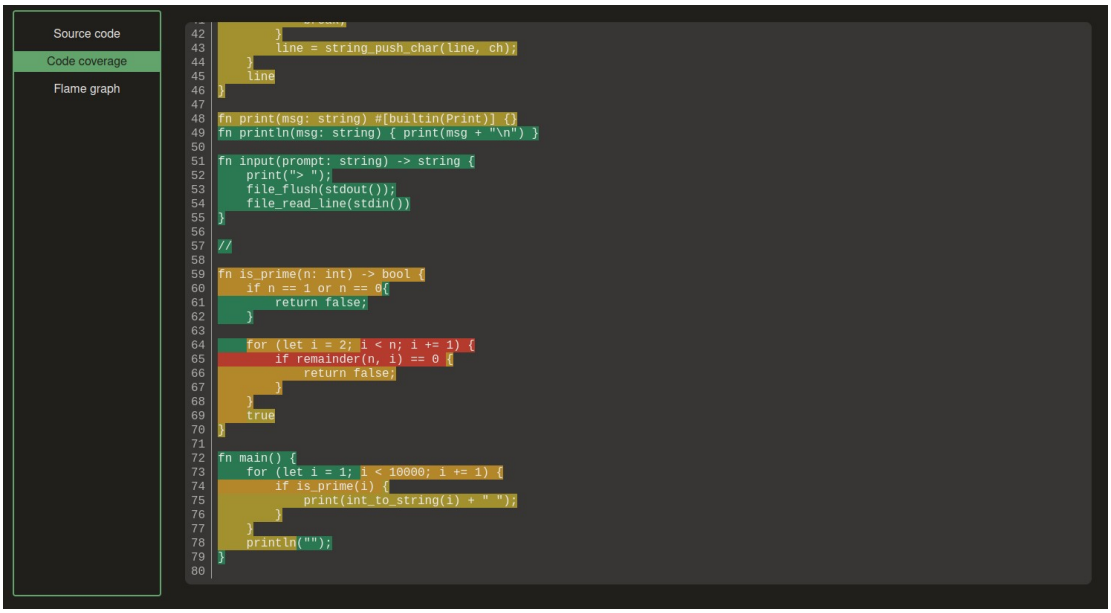
Vi har et system, der fungerer som en schweizerkniv af diagnosteringsværktøjer til vores sprog, slige. De 3 værktøjer er således:

Source code

Source code, som vist på billedet nedenunder, viser kildekoden til ens slige program. Den er formatteret til at ligne et standard koderedigeringsværktøj. Man kan bruge Source code, til at gennemgå koden af det program, som man kører.

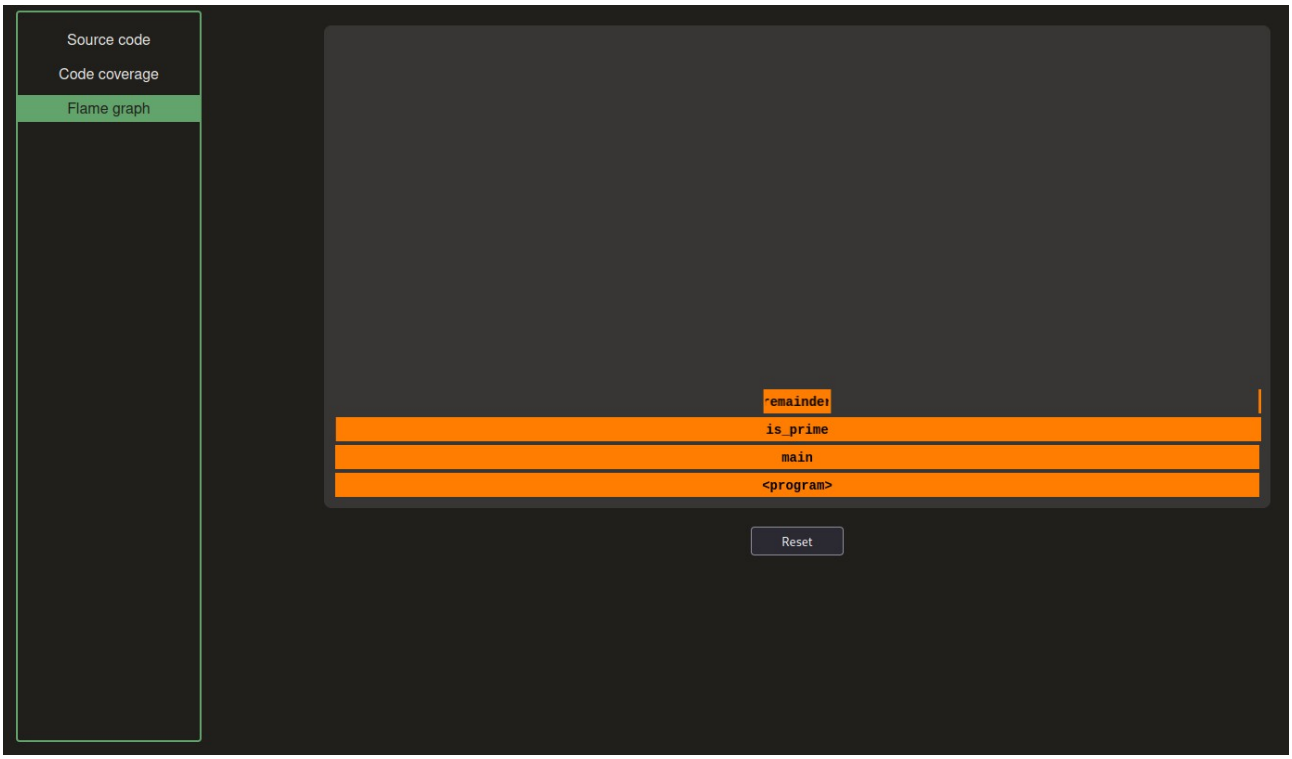
Code coverage

Code coverage, vist på billedet, viser en, hvor mange gange en linje i ens program bliver kørt. Den bruges til at udregne, om kode man har skrevet til at teste ens slige program, dækker alt kode som der testes. Her er grøn brugt til det kode, der køres mindst, og rødt til det kode der køres mest.



Flame graph

En flame graph, vist på billedet, bliver brugt til at vise, hvor programmet har brugt størstedelen af sin tid, med håbet om at kunne vise brugeren,

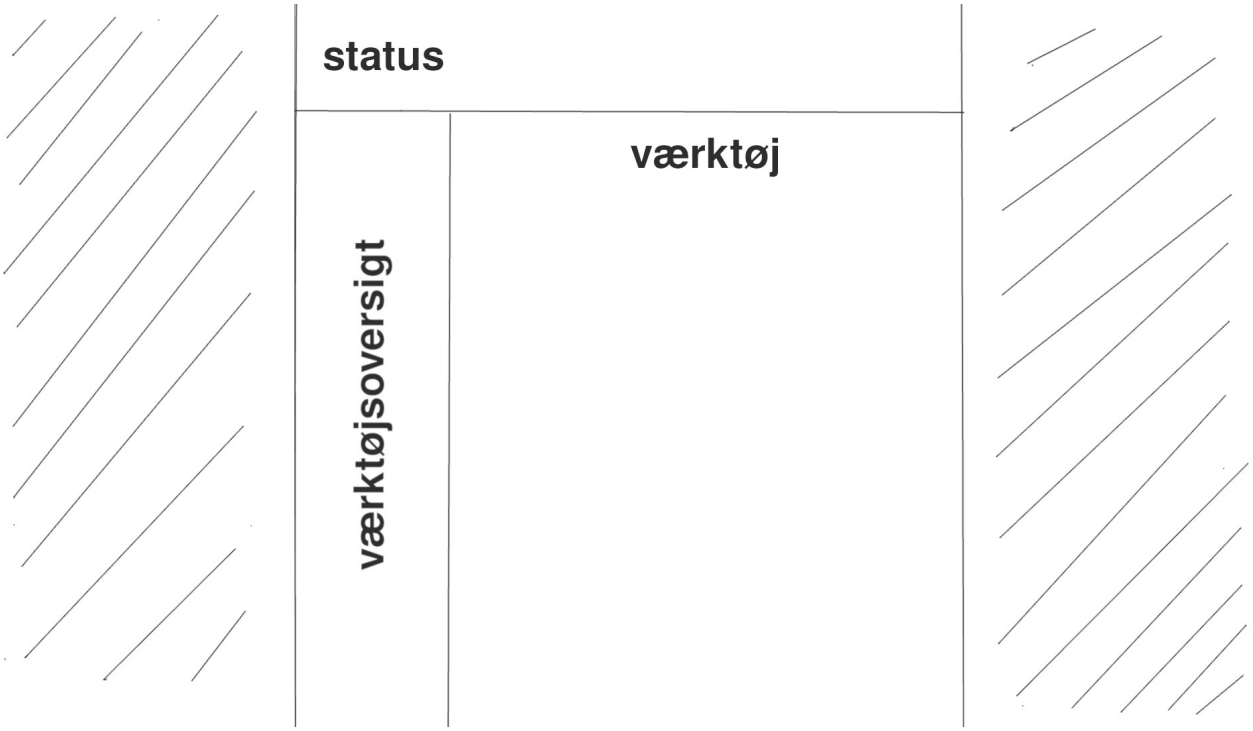


T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

UI-design

Layout

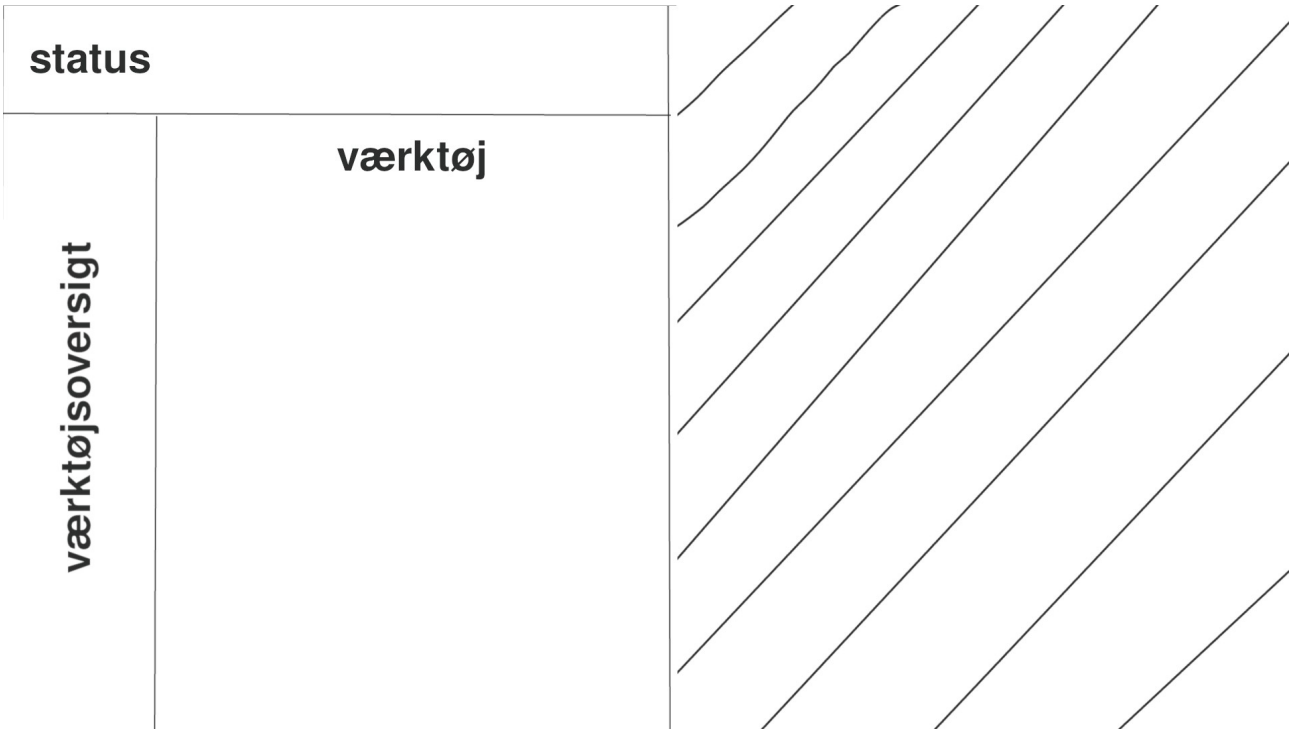
Vores design er opbygget med lignende layout:



Hvor statuslinjen, ”status”, viser programmets nuværende status, ”Running” eller ”Done”, og hvor værktøjsoversigten viser hvilke værktøj man kan vælge, og hvilket værktøj man har valgt. Det valgte værktøj vises så i ”værktøj”.

Vi har valgt at have noget død plads (striberne) i siderne af vores design, da en person der kigger på skærmen naturligt placerer sine øjnene midt på. Det giver derfor mening at placere det vigtigste i midten af skærmen.

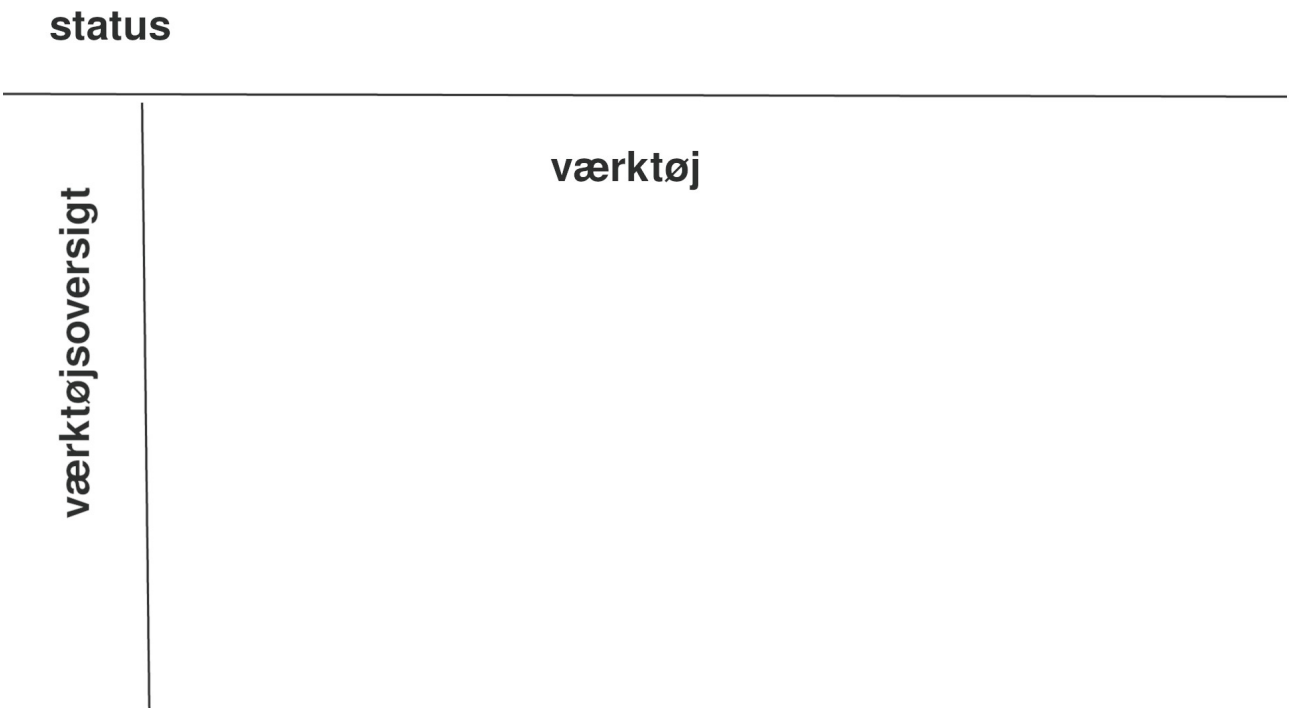
Derudover har vi valgt at centrere det vha. det døde plads frem for at bruge 100% af skærmen, da værktøjet ikke kunne fylde 100% af skærmen, og det derfor ville ende med at ligne således:



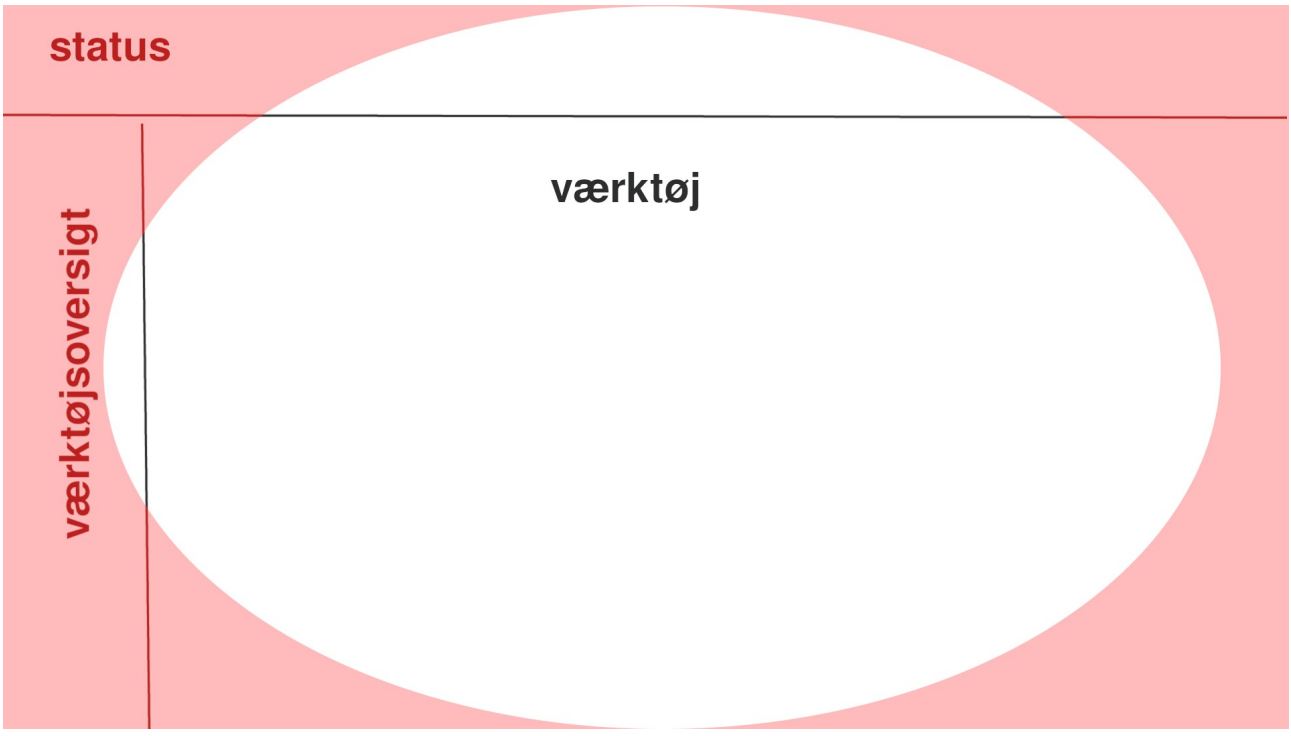
Hvilket ville betyde at brugeren ofte skulle kigge til venstre på skærmen frem for bare midtpå, hvilket er en unnaturlig bevægelse (ift. at kigge lige ud) og derfor mindre intuitivt for brugeren.

Derudover, selv hvis værktøjet kunne tage 100% af pladsen, ville det fjerne opmærksomheden fra bl.a. status og værktøjsoversigt:

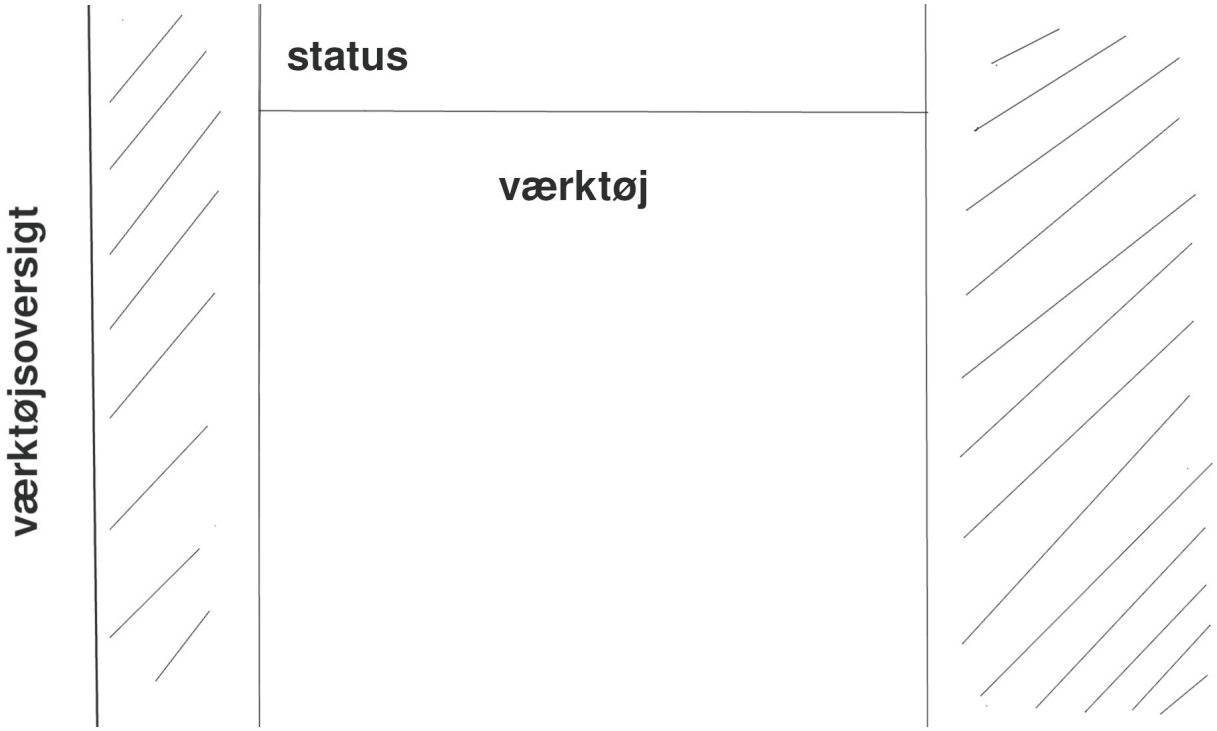
T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia



Da man nu naturligt har øjnene i midten af skærmen, som fremvist her, med de røde zoner er dem man naturligt ikke fokuserer på, hvis man kigger midt på skærmen.

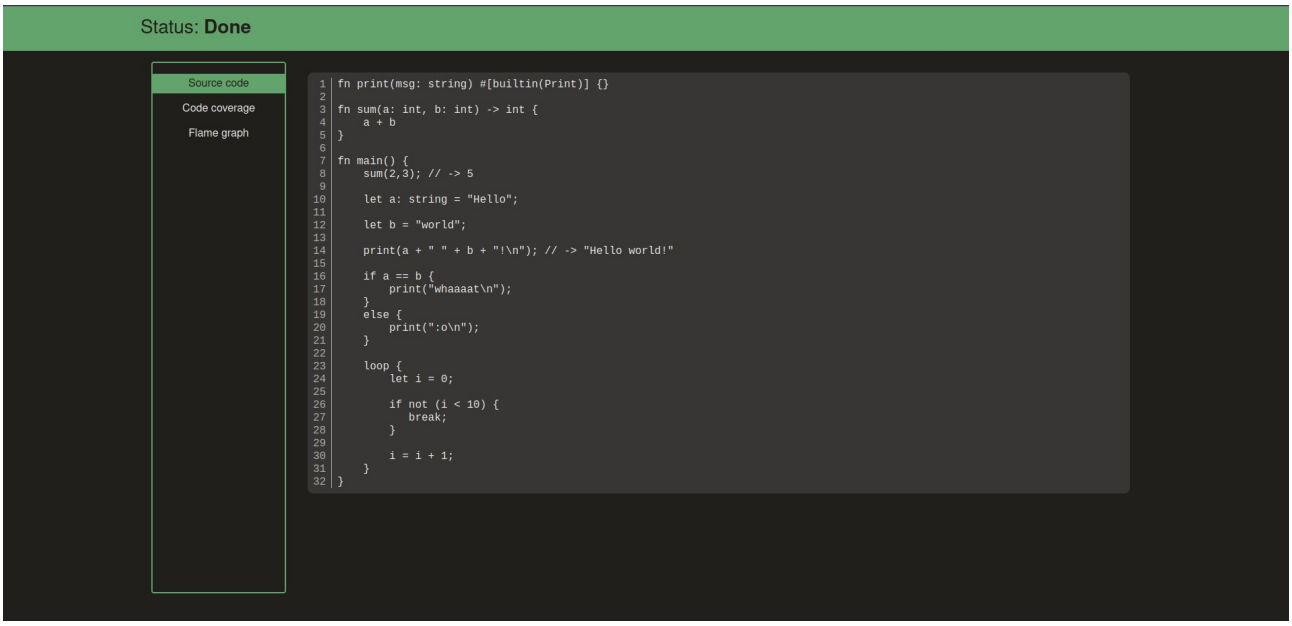


Vi har ikke valgt at bruge dette døde plads til at adskille værktøjet fra værktøjsoversigten, altså således:



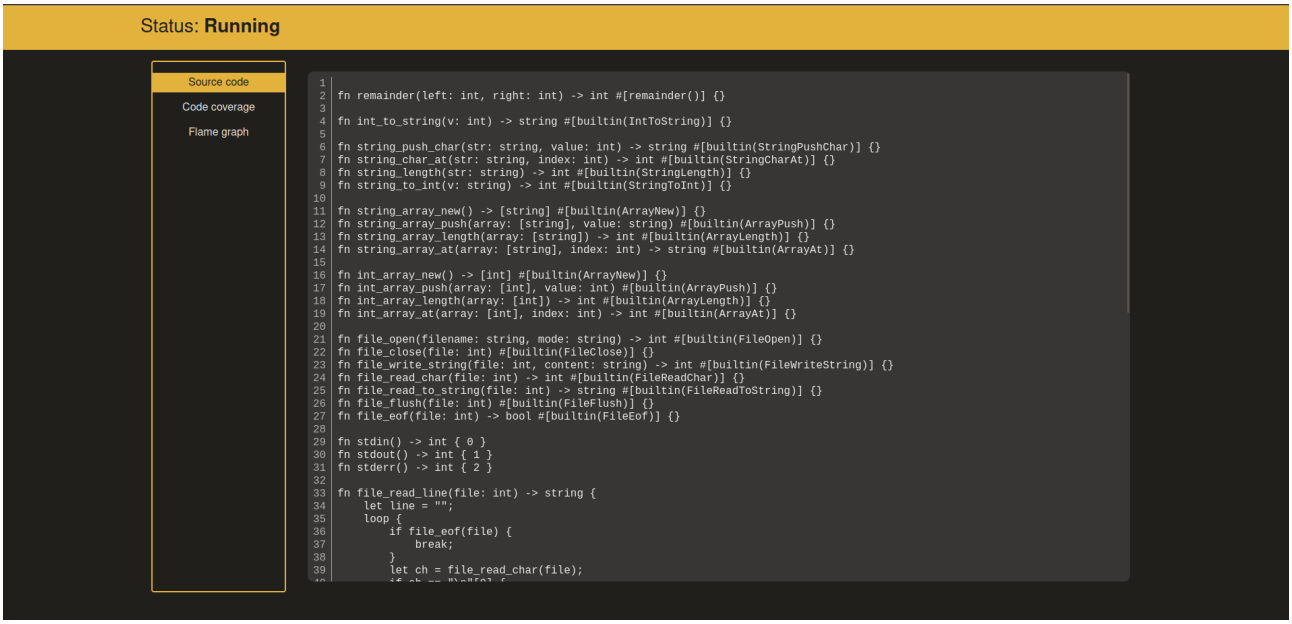
Både grundet det tidligere nævnte problem ift. hvor brugerens øjne falder, og da værktøjsoversigten er en vigtig del af værktøjet, er de altså tæt tilknyttet, og derfor giver det ikke mening at have dem afskildt fra hinanden, da når brugeren ser på værktøjet skal de gerne selv kunne lave forbindelsen mellem værktøjsoversigten og værktøjet der er taget i brug. Rent fagligt bruger vi altså derfor gestaltloven om nærhed.

Vores implementation ser ud således:



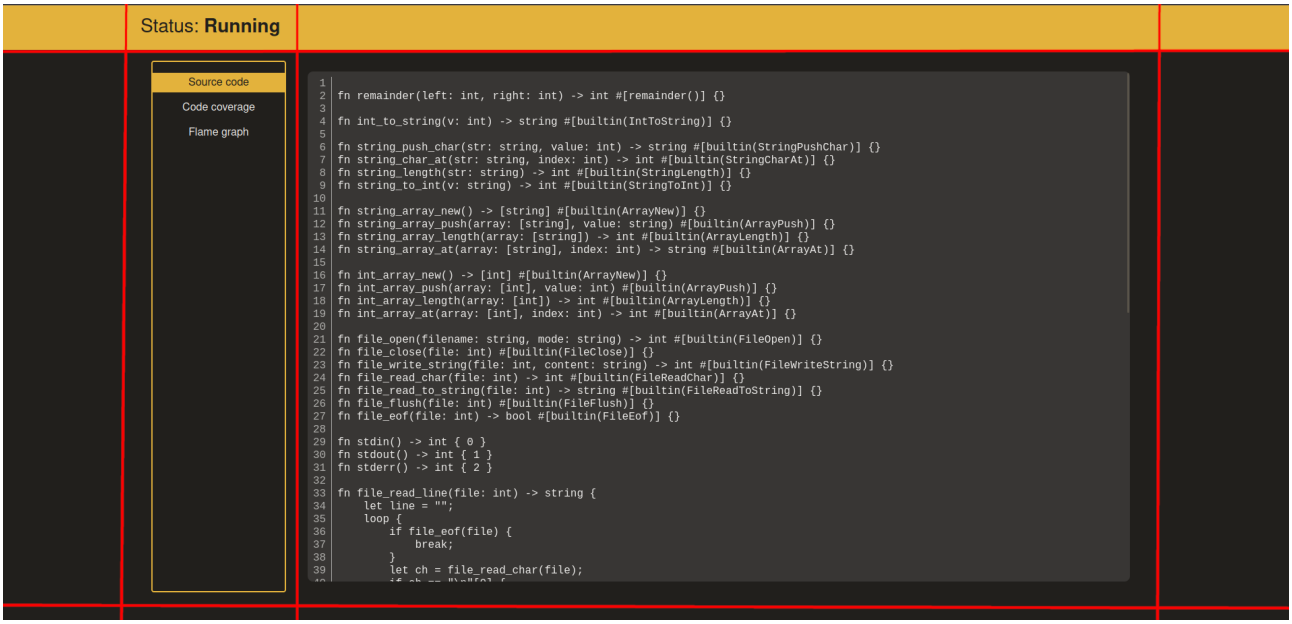
Her kan man se en tydelig forbindelse mellem det valgte værktøj, i dette tilfælde ”Source code”, og det værktøj der bliver vist.

Vores primære farve brugt reflekterer og forstærker programmet’s status, som vist her:

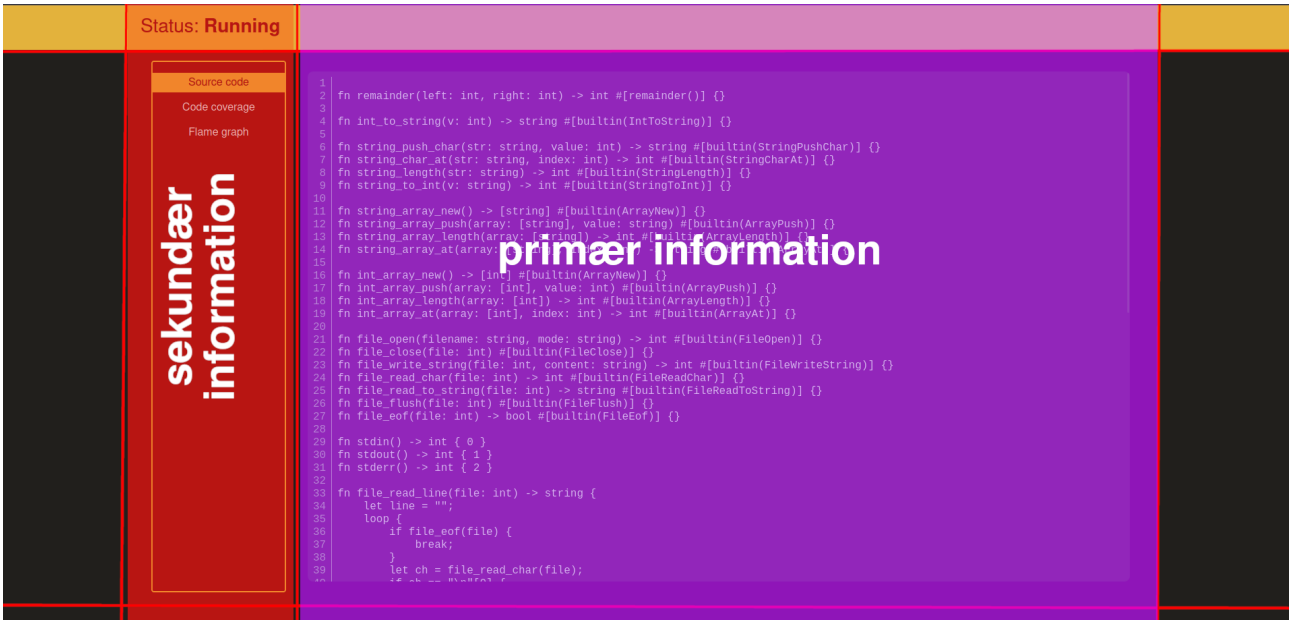


Altså bliver brugeren påmindet om status af programmet hvorenten de kigger, ved at se om hvorvidt det fremhævede er gult eller grønt. Det gør også, at man kan hurtigt se status bare ved at se farven af statuslinjen; rent fagligt er der altså tale om gestaltloven om forbundethed.

Vi har valgt at ikke centrere statuslinjen, eftersom at øjnene danner et naturligt opdeling ud fra de afgrænsninger vi har skabt, således:



Som gør at man naturligt kigger i det afgrænsede område for sekundær information som f.eks. status eller værktøjsoversigt:



Vi har her anvendt gestaltloven om nærhed, til at skabe en gruppering af den relativt statiske sekundære information og dynamiske primær information.

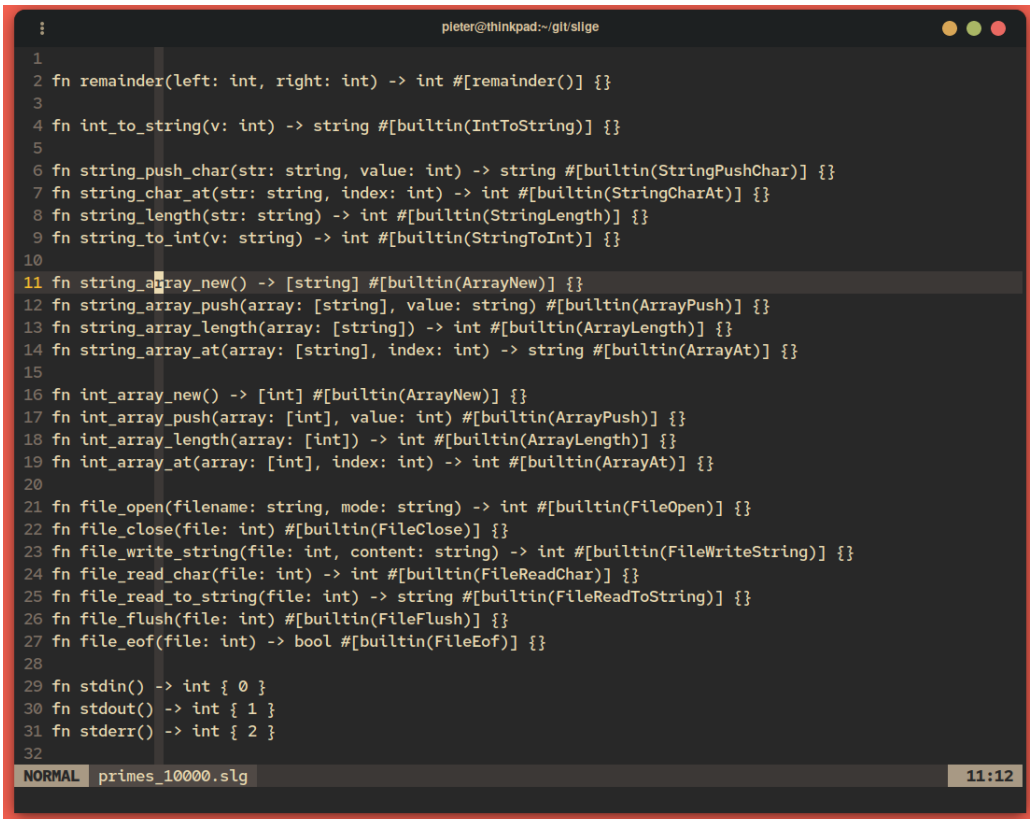
Vi har brugt vores primære farve, som reflekterer status, til at fremhæve vigtige elementer, f.eks. bruger vi det i værktøjsudvalgtet, til at vise det valgte værktøj, eller på statuslinjen, sammen med fed skrift, til at vise den nuværende status af programmet. Det gør at øjnene får hjælp til at falde på det vi synes er vigtigt, men som ellers kunne glippes, altså status og valgte værktøj; selve værktøjet bliver fokuseret på i sig selv, bare ved at det ligger i midten, og optager størstedelen af pladsen, derfor behøver vi ikke at fremhæve det yderligere, da det er der hvor øjnene naturligt falder.

Vi anvender i værktøjsoversigten også loven om lukkethed, ved at lukke værktøjsmulighederne ind i en kasse for sig, og da de det valgte værktøj også ændrer værktøjet der bliver vist, skabes der er en forbindelse mellem valgene værktøjsoversigten og værktøjerne der kan vælges.

Værktøjet

I værktøjet relateret til source code har vi valgt at vise teksten med linjetal og en monospace typografi, altså, typografi hvor alle bogstaver, tal og symboler har samme bredde.

Dette er for at forstærkte, at det som brugeren kigger på, er kode, da det ligner en traditionel kode-redigeringsprogram, som vist:

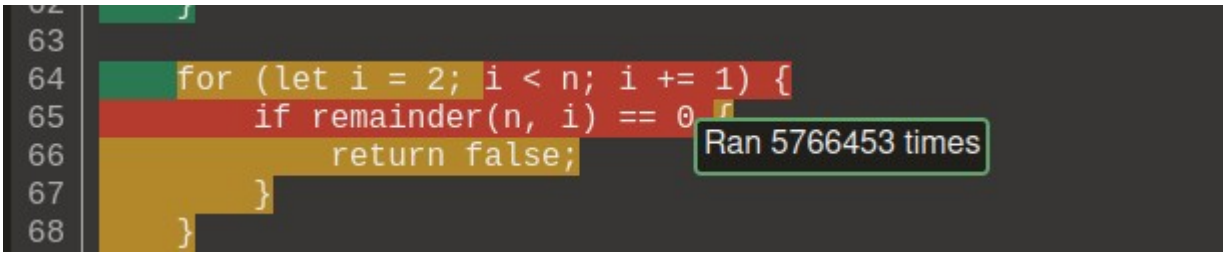


Vi bruger derfor brugerens allerede trænet evne til at genkende programmeringsredigeringsværktøjer, til at vise vores kode, da de med det samme kan se, at det er kode, som de kigger på, uden at skulle tænke dybere over det.

I vores code-coverage værktøj, viser vi det selvsamme, udover at vi har vist farver på, hvilket kode der bliver kørt mest.

```
        }
        line = string_push_char(line, ch);
    }
    line
}

fn print(msg: string) #[builtin(Print)] {}
fn println(msg: string) { print(msg + "\n") }

fn input(prompt: string) -> string {
    print("> ");
    file_flush(stdout());
    file_read_line(stdin())
}

//

fn is_prime(n: int) -> bool {
    if n == 1 or n == 0{
        return false;
    }

    for (let i = 2; i < n; i += 1) {
        if remainder(n, i) == 0 {
            return false;
        }
    }
    true
}

fn main() {
    for (let i = 1; i < 10000; i += 1) {
        if is_prime(i) {
            print(int_to_string(i) + " ");
        }
    }
    println("");
}
```

Farven bliver valgt på en skala mellem grøn, gul, og rød, til at vise hvilket kode der bliver kørt mest. Vi har valgt grøn, gul og rød da vi derfor kan bruge brugerens eksisterende intuition at grøn, gult og rød er god, middel, og dårlig. Man kan derudover også holde musen over, for at se yderligere information:



Vi vurderer at dette er intuitivt på baggrund af, at man ofte bruger musen på at ”pege” på ting, og det derfor, givet at der ikke er andre muligheder, er sådan man finder yderligere information.

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Implementering

Kompiler

En kompiler er et program der oversætter et programmeringsprog til maskinsprog. Et programmeringsprog er det en programmør programmere i. Maskinsproget er det sprog maskinen bedst forstår. Maskinsproget er uhensigtsmæssigt for programmøren at skrive i, fordi det er fjernt for, hvordan mennesket tænker. Computere er gode til tal. Mennesket er gode til at tænke i koncepter. For eksempelt i maskinsproget, hvis man skal kalde en funktion, skal man kende den præcise ram-adresse. I et programmeringsprog skal man kun huske navnet på funktionen. Derfor skriver programmøren ikke i direkte i maskinsprog og det er derfor essentielt for programmøren at bruge en kompiler til at kompilere programmeringsproget

Slige-kompileren

I vores kompiler kan man opnå dette gennem 5 trin. Det første trin er lexeren. Lexeren kigger på hvert element i koden og fortæller omskriver det til tokens. En token beskriver typen, positionen, længden og den enventuelle værdi af elementet. Det næste trin er parseren. Her bliver lexerens output oversat til en træ struktur.

De næste to trin analyseres der på træ strukturen gennem resolveren og checkeren. I parserens træ struktur eksistere der idents, også kald identifiers. Idents er et navn. Det kan for eksempel være et funktionsnavn eller et variable navn. Efter en ident er blevet parset i parseren, refererer idents ikke til der, hvor de er defineret. Resolveren kigger på alle idents og finder der, hvor de er defineret. Derefter ændre den tilføjer den i træstrukturen der hvor ident'en er defineret. Ident'en er ny blevet til et symbol. Det næste analyseringstrin altså checkeren handler om at validere typerne i koden. Her kigger vi i træstrukturen og tjekker om de definerede typer passer.

De sidste trin er byte-kode generering. Her tager vi den nuværende træstruktur og konvertere det til byte-kode. Byte-koden er et format vi har lavet som kompilereren kompilere programmet ned til, og som vores runtime miljø kan afvikle. Vores byte-kodeformat er stack-baseret, det betyder at instruktionerene operere på værdier ligesom et stak papir. Hvis man skal lægge to tal sammen, så lægger man den første værdi på stack'en, derefter den anden værdi. Så udføre man en addtion-operation, som fjerne de to øverste værdier fra stack'en, lægger dem sammen, og lægger resultatet på stack'en.

Denne stack er vores byte-kode. Byte-koden bliver læst og forstået af vores runtime. I runtime'en har vi en VM (virtual maskine) hvori programmets instruksioner bliver kørt.

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Datastrukturer

I starten af processen arbejder vi med Tokens i sekventiel form altså at man får Tokens en efter en. Det bliver konverteret til en træ struktur som man for eksempel arbejder på ved hjælp af tree triversal, hvor man bruger rekursion til at besøge alle træets forgreninger. Til sidst er det igen blevet konverteret til en sekvens hvori der er instruksioner.

Lexerens opgave er at dele teksten op i små logiske bidder. Men tokens har ikke struktur.
I parseren laver vi tokens om til AST som er en træ struktur. Det gør vi fordi AST'en skal opbevare kodens struktur. Senere i resolveren laver vi AST'en om fra en acyclis træ struktur til en cyclis træ struktur. Dette gør teknisk set at det ikke er et AST, men vi betegner det som en AST, da det er nemmest. Data strukturen går herved fra at være en træ struktur til at være en graf struktur, hvor at flere noder i grafen kan henvise til den samme node. Dette er forid at vi både gemmer kodestrukturen som vi havde før, samtidig med at navne også peger på de noder, navnet henviser til. Dette er måden vi eksempelvis håndtere variabel-navne.

En af fordelene ved at byte-koden er sekventiel er, at vi nemt kan gemme dataen i en fil. Dette gør det nemt at aflevere kompileren til runtime. Træstrukturen derimod at gemme i en fil, da den indeholder dels struktur, som er svært at serialisere, og så indeholder den også AST'ens noder henvisninger/referencer til andre noder. Dette gør det sværere at serialisere.

Valg af teknologier

Vi har valgt at bruge Deno og Typescript til implementeringen af kompileren. Det har vi brugt siden Typescript er typestærkt og kompileren vi ville implementere skulle være typestærkt. Vi brugte også Typescript siden alle i gruppen kender det og har arbejdet med det før.

Slige-kompileren og GNU GCC kompileren

For at sammenligne vores implementation har vi tænkt os sammenligne med en C-kompiler specifikt GNU GCC kompiler. Det er to forskellige sprog der kompiles. C-kompileren kompiler C-syntax og vores kompiler Slige-syntax. Der er nogle teoretiske forskelle på syntaxen i C og syntax i Slige, som har en indflydelse på, hvordan kompileren skal implementeres. Men disse er udenfor ambitionerne i denne rapport at dykke ned i. Slige-kompileren parser alt koden før den går videre til næste trin, modsat C-kompileren der kompiler helt ned til bytecode for hver linje syntax. Det betyder, at man eksempelvis ikke kan definere en funktion i linjerne under kaldet af funktionen. Fordi at Slige-sproget er en multiparse kompiler kan dette godt lade sig gøre, dog er der en lille nuance i C-kompileren, som vi kommer ind på nedenfor.

Slige-kompileren kompiler til Slige-bytecode, hvor at C-kompileren kompilere til den fysiske maskines kode. Hvor at Slige's bytecode er lavet til at køre i Slige's runtime. Slige-kompileren kompiler direkte til Slige-bytecode, C-kompiler i stedet for at kompile direkte til bytecode, kompiler den til det der kaldes IR (Intermediate Representation). AST'en i Slige-kompileren er også en form for intermediate representation, men IR henviser til en specifik form for virtuel bytecode, som C-kompileren optimere

på, før den kompilere den virtuelle bytecode til maskinkode. Dette kaldes optimerings-passes som gør C-kompilerens output af højere kvalitet. Dette har vi ikke implementeret i Slige-kompileren, da vi for det første ikke har et problem med afviklingstid på den nuværende implementation. Og for det andet fordi optimerings-passes kræver mere arbejde at implementere end, at vi vurdere til at være nødvendigt.

Runtime og web

Produktet består også af et runtime-miljø, altså afviklingsmiljø, til at afvikle programmet. Programmet den afvikler er i form af den byte-kode, som kompileren producerer. Runtime'en består af en virtual maskine, et flame-graph-komponent, en code-coverage-komponent og et fjernstyrings-/datatransport-komponent, som vi vil kigge på.

Produktet består også af en web-app. Dette er produktets brugergrænseflade, er den som viser diverse værktøjer til brugeren og som brugeren kan interagere med.

Runtime'ens filer ligger i mappen 'runtime/', og web-appens filer ligger I mappen 'web/' i produktet.

Virtuel maskine

Runtime'ens hovedkomponent er den virtuelle maskine (VM), som afvikler programmet. Programmet er i form af sekventiel byte-kode. Måden VM'en afvikler programmet, er ved at itererer over hvert instruktion og afvikle den specifikke instruktion. VM'en har også en værdi-stack, som er de værdier, programmet arbejder på. VM'ens implementering ligner følgende psuedo-koden.

```
while pc < program.length
	match program[pc] // instruktion ved nuværende lokation
		...
		case PushInt
			value = program[pc + 1] // efter PushInt ligger værdien
			stack_push(value) // læg værdien på stack'en
			pc += 2 // fortsæt til næste instruktion, PushInt's størrelse er 2 værdier
		case Add
			left = stack_pop() // tag de 2 øverste værdier fra stacken
			right = stack_pop()
			result = left.as_int() + right.as_int() // læg dem sammen som int-værdier
			stack_push(result)
			pc += 1 // Add's størrelse er 1 værdi
		...
```

VM'en holder styr på, hvilken instruktion, dvs. lokation i programmet, den er kommet til, med en variabel, der hedder program-counter (PC), som kan oversættes til instruktions-tæller. Nogle instruktioner manipulere denne tæller, så det er muligt, at hoppe rundt i programmet. Denne instruktion hedder Jump. Dette er måden løkker er implementeret. Det er også muligt at hoppe, hvis en betingelse er opfyldt med JumpIfTrue-instruktionen. Dette er måden forgreninger med if-statements er implementeret. Implementering af disse, kan forklares med følgende psuedo-kode.

```
while pc < program.length
	match program[pc] // instruktion ved nuværende lokation
		...
		case Jump
```

```
                location = stack_pop() // få lokationen vi skal hoppe til fra stack'en
                pc = location.as_address()
        case JumpIfTrue
                location = stack_pop()
                condition = stack_pop() // få resultaten af betingelsesudtrykket
                if condition.as_bool() == true
                        pc = location.as_address() // hop, hvis sandt
                else
                        pc += 1 // ellers, fortsæt videre, hvis ikke sandt
        ...
```

VM'ens kode ligger i filerne 'runtime/vm.hpp' og 'runtime/vm.cpp' og er beskrevet i c++-klassen 'sliger::VM'.

Code-coverage

Runtime'en består også af en code-coverage-bygger. I programmet er der inkluderet en instruktion, der hedder SourceMap. Den fortæller VM'en, at et specifikt sted i byte-kodeprogrammet, korresponderer til et specifikt programmets kildekode. SourceMap instruktionen oplyser linje- og kolonnenummer og indeks i filen, dvs. det specifikke tegns placering i filen. Hver gang VM'en passerer en SourceMap-instruktion, noterer den ned, hvor mange gange den har støt på lokationen. Specifikt inkrementerer den en tæller, på den lokation, den forlader. Det kan illustreres med følgende psuedo-kode.

```
loop
        match program[pc] // instruktion ved nuværende lokation
                ...
                case SourceMap
                        index = program[pc + 1].as_int()
                        line = program[pc + 2].as_int()
                        column = program[pc + 3].as_int()
                        new_location = { index, line, column };
                        code_coverage_report_leave(location)
                        current_location = new_location
                        i += 4
```

Code-coverage-komponenten kan så ud fra dette producere en liste af kildekodelokationer og for hver lokation, hvor mange gange koden er afviklet. Ved at vi også håndterer start- og slut-lokationer, dækker denne liste hele programmets kildekode.

Vha. datatransportkomponentet, sender vi dataen fra runtime'en til webappen, hvor brugergrænsefladen vises. Sammen med code-coverage-dataen bliver programmets kildekode også send til webappen. Webapp konstruerer, ud fra code-coverage-dataen og kildekoden, et Pre-element, med farvede stykker tekst. Et Pre-element er et komponent i browseren, til at vise pre-formateret tekst som programmeringskode. Textstykkerne er Span-elementer, hvilket er et komponent til tekstykker, som vi giver en farve og sætter ind i Pre-elementet. Farven vælger vi ud fra logaritmisk interpolation. Følgende er et udsnit af Typescript-koden, til at udregne farven.

```
type Color = { r: number; g: number; b: number };
```

```
// lineær interpolation af første grad
function lerp2(ratio: number, start: number, end: number) {
    return (1 - ratio) * start + ratio * end;
}

// lineær interpolation af anden grad
function lerp3(ratio: number, start: number, middle: number, end: number) {
    return (1 - ratio) * lerp2(ratio, start, middle) +
        ratio * lerp2(ratio, middle, end);
}

// lineær interpolation af anden grad for hver farve: rød, grøn, blå
function colorLerp(
    ratio: number,
    start: Color,
    middle: Color,
    end: Color,
): Color {
    return {
        r: lerp3(ratio, start.r, middle.r, end.r),
        g: lerp3(ratio, start.g, middle.g, end.g),
        b: lerp3(ratio, start.b, middle.b, end.b),
    };
}

function colorToString(color: Color): string {
    return `rgb(${color.r}, ${color.g}, ${color.b})`;
}

function perfColor(ratio: number): {
       return colorToString(colorLerp(ratio, GREEN, YELLOW, RED));
}

// valgte farver
const GREEN = { r: 42, g: 121, b: 82 };
const YELLOW = { r: 247, g: 203, b: 21 };
const RED = { r: 167, g: 29, b: 49 };
...
       // find det maximale afviklinger for en lokation i code-coverage-dataen
       const maxCovers = entries
              .map(entry => entry.covers)
              .toSorted();
              .at(-1); // få sidste af de sorterede værdier, dvs. den største værdi

...
              if (mode === “performance) {
                     span.style.backgroundColor =
                            perfColor(Math.log10(entry.covers) / Math.log10(maxPerfCovers));
              } else if (mode === “test-coverage”) {
                     span.style.backgroundColor =
                            entry.covers > 0
                                   ? colorToString(GREEN)
                                   : colorToString(RED);
              }
```

Altså kan det ses, at vi udregninger farven med logaritmisk interpolation, som vi udregner med linear interpolation af tredje grad, men hvor input-værdien regnes logaritmisk ud fra mængden af afviklinger. Det kan også ses, at der er 2 forskellige modes. For performance-mode’en benytter vi den udregnede farve. I test-coverage-mode’en bruger vi grøn, når koden er kørt, og rød når den ikke er.

Runtime'en indeholder også et flame-graph-komponent. Ligesom code-coverage-komponentet, sidder dette også i VM'en. Flame-graph-komponentet fungerer, ved at holde styr på funktionskald og returnering fra funktioner. VM'en har en akkumulativ instruktionstæller. Denne tæller tæller op ligesom den anden instruktionstæller, men den skifter hopper ikke ved Jump- og lign instruktioner. Flame-graph-komponentet bruger denne tæller. Når en funktion kaldes, gemmer komponentet tællerens værdi og funktionens id. Når funktionen returneres fra igen, gemmer komponentet så mængden af instruktioner, funktionen afviklede, som vi betragter som tiden, det tog at køre funktionen. Tiden her er altså i instruktioner, og den er regnet ved at tage tælleres nuværende værdi og trække værdien ved start fra. Denne forskelsværdi kommer så i en akkumulativ tæller specifikt for funktionen. Flame-graph-komponentet gæmmer ikke alle funktionsafvikler for den samme funktion sammen. Hvis funktion A kaldes af både funktion B og funktion C, opretter komponentet 2 graph-noder. Implementeringen i runtime'en kan forklares med følgende psuedo-kode.

```
while pc < program.length
	match program[pc]
		...
		case Call
			fn_address = stack_pop().as_address();
			...
			flame_graph_report_call(address, instruction_counter)
			...
			pc = fn_address
		case Return
			...
			flame_graph_report_return(instruction_counter)
			...
...
struct FGNode { // flame-graph-node
	...
	uint32_t fn;          // funktionens id/indeks/addresse
	int64_t acc = 0;      // akkumulativ nodespecifik instructionstæller
	int64_t ic_start = 0; // instructionstællerens værdi, ved starten af funktionskaldet
	size_t parent;        // funktionen, som udførte funktionskaldet
	...
	std::vector<size_t> children = {}; // funktioner som denne funktion kalder
};
...
	inline void report_call(uint32_t fn, int64_t ic_start)
	{
		// lav ny node eller find noden, hvis funktionen er kaldt før på samme måde
		size_t found = find_or_create_child(fn);

		// gem instruktionstællers værdi ved start
		this->nodes[found].ic_start = ic_start;
		...
	}

	inline void report_return(int64_t ic_end)
	{
		// udregn tiden, dvs. forskellen på instruktionstællerens værdi fra start til slut
		int64_t diff = ic_end - this->nodes[this->current].ic_start;

		// læg tiden sammen med tidligere tider
		this->nodes[this->current].acc += diff;
		...
	}
```

Dermed skabes der altså en træstruktur, som viser hver funktions tidsmæssige komposition af underliggende funktionskald under afvikling. Dataen bliver sent til webappen sammen med et opslagsliste af funktionsaddresser og funktionsnavne, så webappen kan vise navne istedet for funktionsaddresser.

Runtime'en er implementeret i C++. Vi har valgt dette programmeringssprog, da et af dets principper er, at kode skrevet i sproget effektivt by default, og derved undgår vi i større grad, at skulle tænke på runtime'ens ydelse. Udover C++ har vi anvendt et byggesystem med Make, som passer godt til de Linux-miljø'er, vi har udviklet produktet i. Dette gør dog, at runtime'en er sværere at kører på anderledes miljøer platforme, eksempelvis Windows. Til webappen har vi anvendt Typescript, HTML og CSS, som vi kan afvikle i en normal webbrowser som Firefox. Vi har anvendt afviklingsmiløet Deno, da det med sine kodeværktøjer gør det nemt, at bruges som webserver, til at kører vores applikation i browseren, og samtidig fungerer som et transportlag mellem kompileren, runtime'en og webappens brugergrafiske del.

slige-sprog

Vi har valgt at designe et programmeringssprog specifikt til dette værktøj. Dette har vi valgt at gøre af flere årsager. Den primære årsag er af implementeringsmæssige årsager. Ved selv at definere sproget, kan vi designe sproget, så det er nemmere at implementere. Vi har valgt selv at skrive parseren, da vi har nogle specielle krav. Et alternativ, hvis vi ikke havde disse krav, kunne være eksisterende parsers eller parser generatorer. Sprogets design er derfor gået hånd-i-hånd med kompilerens implementering.

Følgende eksempel er af Slige-sproget.

```
fn main() -> int {
        let a = 5;
        let c = sum(a, 5);
        println(“c = ” + itos(c));
}

fn sum(a: int, b: int) -> int { a + b }
```

Eksemplet består af 2 funktioner: main og add. Funktionen add tager 2 parametre: a og b, begge integer-heltal, som den lægger sammen. Det sidste udtryk i en kodeblok, designeret af { og }, returneres. Funktionen main laver en variabel a, som den assignere 5 til. Derefter definerer den en variabel c, hvori den gemmer værdi af sum-funktion, som den kalder med værdien af a og tallet 5. Til sidst kalder den println-funktionen, som skriver text til skærmen. Funktionen itos omdanner en integer-værdi til en tekststring (string), som bliver samlet med textstringen med “c = ”. Når dette kode afvikles, bliver teksten “c = 10” skrevet til brugeren.

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Designvalg/målgruppe

Vi har ovenfor etableret, at vores primære målgruppe er programmører. Derfor har vi designet et sprog til programmører. Eksempelvis er designationsordet til funktioner forkortet til **fn**, da programmører allerede ved, dette betyder funktion.

Sprogens grammatik er designet på sådanne måde, at sproget er nemt at parse, altså nemt for kompileren at forstå. Eksempelvis gælder det for mange af konstruktionerne, at parseren ud fra det første token kan gætte, hvordan det skal parses. Dette gælder eksempelvis for fn-, let- og if-konstruktioner. Dog gælder dette ikke alle steder i grammatikken, eksempelvis for binære udtryk, som er udtryk som addition med 2 værdier. Vi vil gerne kunne skelne mellem udtrykkene 2 + 3 * 4 og (2 + 3) * 4, da udtrykkene matematisk set giver forskellige svar, henholdsvis 14 og 20. Dette kræver en lidt mere advanceret parser, men gør sproget nemmere at læse og anvende, sammenlignet eksempelvis med prefix udtryk, som er nemmere at parse, men sværre at læse. Udtrykkene fra før, vil udtrykkes med prefix henholdvis som + 2 * 3 4 og * + 2 3 4. Vi vurderede, det ekstra arbejde var det værd, for at implementerer.

Sproget anvender automatisk hukkomelses-håndtering. Som diskuteret ovenfor, gør automatisk håndtering af ressourcer det nemmere at håndtere programfejl, specifikt fejl i forhold til hukommelse og ressourcer. Den største ulempe ved automatisk håndtering er, at det ofte er langsommere end alternativerne. Det kræver også et stort afviklingsmiljø. Siden vi allerede har et større afviklingsmiljø og værktøjets egen afviklingshastighed ikke er en prioritet for os, har vi vurderet, at dette var den bedste løsning. Vi kan sammenligne med, hvis værktøjet kunne garentere korrekthed i forhold til ressourcer. Automatisk håndtering er nemmere at implementere end en garenterende kompiler. Dog vil en garenterende kompiler kunne producere mere optimeret output, dvs. output af højere kvalitet. Men fordi sprogets egen afviklingstid ikke er et problem for os, er dette ikke en bedre løsning.

Evaluering

Vi vil herunder evaluere både produktet og vores arbejdsprocess gennem projektet.

Evaluering af produkt

Vi har udviklet et produkt, til at hæve kvaliteten af software i softwareudvikling. Produktet er et enkelt værktøj eller miljø, som samler flere værktøjer i et værktøjer. Disse værktøjer er et code-coverage-værktøj, et flame-graph-værktøj og et programmeringssprog med kompiler og runtime. For at evaluere produktet, er vi nødt til at undersøge dels brugbarheden og anvendeligheden af vores produkt.

For at undersøge dette, har vi lavet nogle afsluttende undersøgelser. Hvis disse undersøgelser viser, at vores produkt kan bruges til hæve kvaliteten af et stykke software, ved at assistere udvikleren, har vi svaret på vores problemstilling.

Evaluering af intuitivitet og brugbarhed

Første undersøgelse omhandler generel brugbarhed af produktet. Ud fra erfaringer vi har gjort os og vores egen vurdering, er vores produkt intuitivt for programmører. Vi har derfor valgt at undersøge, i hvor høj grad ikke-programmeringstekniske personer forstår og kan anvende vores produkt.

Undersøgelsen var bygget op på følgende måde. En forsøgsperson får udlevet en computer, hvor på produktet er startet op med et eksempelprogram. Forsøgspersonens opgave er så, at navigere igennem værktøjets brugergrænseflade og forklare komponenter, de støder på.

Første forsøgsperson, A, påbegynder undersøgelsen. Det tager A mellem 10 og 20 sekunder, at danne sig et overblik A peger ud, at der på skærmen er en statusbar, noget kildekode og en menu-liste i den rækkefølge på skærmen.

A navigerer derefter til ‘Code coverage’-siden. A forklarer, at den viser, hvad der er kørt, hvor mange gange. A bruger derefter musen, til at se og aflæse, at et stykke var kørt 1 gang. A spørger derefter, om tilladelse til at navigere videre i menu’en. A har derfor overset ‘Test-view’-knappen i bunden af skærmen. Knappen peges ud for A. A er nu præsenteret af kode, som enten er rød eller grøn, og A opdager, rød kode ikke bliver kørt. A siger så, at “hvis det ikke køres, er det ikke godt”. Det fremstod tydeligt, at A ikke kendte sammenhængen til kodetesting i forvejen, men istedet konkluderede ud fra den røde farve.

A navigerede derefter til ‘Flame-graph’-siden. A er nu præsenteret af en flame-graph af kodeeksemplet. A siger, de kan se kodens “funktionsstruktur”. A blev spurgt om at uddybe, og kunne ikke forklare, at flame-graph’en viste afviklingstid. A fandt ikke ud af, det er muligt at trykke på flame-graph’ens barer, for at navigere i graph’en. Dog fandt A ud af, at pege med musen og aflæse at parse_digit og string_slice udgør henholdsvis 72% og 12% af string_to_int_impl, men kunne ikke forklare, hvad det betød.

A blev nu briefet i, hvad produktets værktøjer gjorde og hvad de kunne anvendes til, og blev spurgt om, at komme med feedback, med denne kontekstuelle viden inddraget. A mener, værktøjet giver mening til programmering. A mener, det er meget nemt og intuitivt at sammenligne afviklingstider af forskellige dele af et program. A mener også, at værktøjet er intuitivt anvendeligt, til at vurdere korrekthed.

A blev herefter takket for sin deltagelse.

I forhold til design af brugergrænseflade, opdagede A ikke knapperne på ‘Code coverage’-siden til at skifte mellem ’Performance-view’ og ‘Test-view’. Derodover efterspurgte A en besrivelse i grænsefladen, som giver en lille forklaring til hver element. Specielt forskellen på ‘Performance-view’ og ‘Test-view’ på ‘Code coverage’-siden uklar. Udover det, efterspurgte A, at man kunne se 100% af afviklingstiden på flame-grafen. Dette spørgsmål forstod operatøren ikke, og det hentyder til, at A ikke helt forstod, hvad flame-graph’en viste. For at konkludere på dette, skal vi tage højde for, at A er ikke programmeringsteknisk eller havde kenskab til hverken produktet, slige-sproget eller sådanne slags værktøjer i forvejen. A havde ikke før gjort sig tanker og problematikker i forhold til afviklingstid og korrekthed for software. Dette var den situation, A var i, da de påbegyndte forsøget. Derfra til at A kunne se en umiddelbar værdi i værktøjen, behøvede ikke en særlig stor salespitch. A havde brug for forklaring af, hvad de forskellige værktøjer gik ud på, hvilket også ses, ved at A efterspurgte information i brugergrænsefladen. Derudover havde A ingen problemer med at navigere siden, læse teksten, aflæse værktøjerne og syntes ikke det grafiske indtryk var uacceptabelt. Konklusionen er, at A, uden baggrundsviden og med lille forklaring, kunne se anvendeligheden og værdien af produktet, hvis A selv var en del af den primære målgruppe.

Herefter påbegyndte forsøgsperson B samme undersøgelse. B bliver spurgt om, at navigere siden og sige, hvad de ser, hvad de tror det skal bruges til, og om de ser anvendeligheden af det. B starter på forsiden og siger, de ser noget kode.

Derefter navigerer B til ‘Code coverage’-siden. B kigger lidt rundt og flytter musen hen over koden. Derfter forklare B, hvordan der en en tæller... som siger hvor mange gange koden er kørt.. og der er noget rødt... og noget gult... farverne viser, hvor mange gange koden er blevet kørt. B deducere altså, at værktøjet viser, hvor mange gange koden er afviklet. B finder ‘Test-view’-knappen uden hjælp, men ved ikke helt, hvad det viser. “Grøn godt, rødt skidt?”.

Herefter navigere B videre til ‘Flame graph’-siden. Procentsats... tiden der bruges af hver funktion. Også her deducere B altså, hvad værktøjer viser. B siger, de hellere vil have én bar for hele programmet. Dette er fordi, mener B, at værktøjet ikke giver det fulde overblik. B forstår ikke, hvad ‘Reset’-knappen gør.

B fik derefter at vide, at deres rolle nu var performance-optimizer og blev forklaret, hvad performance-optimering handler om. Her vil B anvende ‘Performance-view’ i ‘Code coverage’-siden til at se, hvad der er kørt mest. B synes vores code-coverage-værktøj med ‘Performance-view’ er et bedre værktøj til performance-optimering end vores flame-graph.

Derefter fik B rollen som tester og blev forklaret, hvad det går ud på. B forstod med det samme, at code-coverage-værktøjed i ‘Test view’ passede godt til denne opgave. Det var fordi, ifølge B, at det er meget nemt at se, hvilket kode man ikke tester, hvilket gør det meget nemt at se, hvilket kode man så skal skrive tests til.

Herefter blev B takket for sin deltagelse.

B er mere programmeringsteknisk inklineret end A. Dog har B aldrig arbejdet med vores produkt, lignende værktøjer, slige-sproget, eller nogle af problematikkerne generelt. Derfor vurderer vi, As og Bs niveauer begge ligger i den sekundære målgruppe. B kunne fint navigere rundt, læse alt tekst og finde alle knapper. Code-coverage-værktøjet gav umiddelbart god mening for B, både til testing, men også til performance-optimering. B foreslog at forsiden med kildekode, skulle integreres ind i ‘Code coverage’-siden. Her viser B en hvis form for forståelse, ved at kunne sammenligne sidernes funktionalitet. B havde en smule kritik for flame-graph’en. B kunne godt se, hvad graph’en viste og hvordan det relaterede til kildekoden. Men B synes ikke graph’en var brugbar til performance-optimering. Istedet ville B have en enkelt bar i graph’en, så den blev mere overskuelig. B har derved vist forståelse for værktøjet i sådanne grad, at B kan vurdere dets anvendelse, og hvordan B hellere vil have det. B mente designet var fint og at produktet er intuitivt.

I undersøgelsen var der et par svagheder. Forsøgspersonerne havde ikke meget tid, til at sætte sig ind i den problematik, som produktet forsøger at løse. Dette krævede derfor en del forklaring fra operatørens side. Dette kunne have gjort, at operatørens biaser blev imposeret ned på forsøgspersonerne via. forklaringerne, som derved kunne have givet et forvrænget indtryk af værktøjet, sammenlignet med en situation, hvor forsøgspersonerne selv skulle finde denne information. En anden svaghed, er at vi ikke

sammenlignede med personer fra andre målgrupper. Vi har derfor kun et billede af brugbarhed i undersøgelserne ud fra denne ene målgruppe.

Vi konkluderer ud fra dette, at selve designet af vores værktøjer er tilstrækkelig i sådanne grad, at selv ikke-tekniske personer, har mulighed for at forstå, hvad værktøjernes usecase er og hvordan de konceptuelt kan anvendes. Hvis forsøgspersonernes målgruppe, var vores primære, ville vi inddrage dele af deres kritik og forslag i vores designbeslutninger. Vi ville lave noget information på brugergrænsefladen, så hvert vigtige element har en associeret forklaring. Derudover vil vi fjerne eller lave flame-graph-værktøjet om, da personer i denne målgruppe, i får meget værdi ud af den. Dog vælger vi ikke at inddrage dette feedback og kritik, da vi ikke føler det relevant for produktets primære målgruppe.

Evaluering af code-coverage

I denne undersøgelse vil vi undersøge anvendeligheden af produktet lidt nærmere. Vi vil undersøge, dels som produktet gør en forskel, når det anvendes og hvor intuitivt produktet er at anvende. Dette vil vi gøre specifikt for code-coverage-værktøjet af et program.

Forsøgsperson C får udleveret en computer. På computeren er der et eksempelprogram. Eksempelprogrammet, såvel som code-coverage og flame-graph kan ses i bilag 1. C får forklaret programmet, som er en funktion med underfunktioner, og et test-suite til funktionen. C får nu til opgave, at pege alt kode ud i funktionerne, som ikke testes af testene. C bruger adskillige sekunder, 30 til 40 på at gennemse koden på 20 til 30 linjer. C finder den ene af de 2 utestede dele.

Herefter åbnes produktet og C præsenteres for det. C aflæser nu fra code-coverage-værktøjet, at der er 2 utestede dele. C syntes værktøjet gav god mening, både konceptuelt og i forhold til vores specifikke implementation. C og operatøren diskuterede også, at værktøjet kunne se gennem hele koden på millesekunder, hvor C bruge adskellige sekunder på det. Selvom C mente, C selv var god til at spotte utestede kodedele, hvis det var nødvendigt, mente C dog, at de gerne ville benytte sådanne værktøj, hvis de fik sådanne opgave igen.

C blev takket for sin deltagelse.

C har programmeringsteknisk forståelse og har arbejdet med lignende problematikker før. Det er tydeligt ud fra undersøgelsen, at code-coverage-værktøjet gjorde en forskel.

Herefter bliv forsøgsperson D sat for samme undersøgelse. D bruger flere minutter på at anskue koden og finder ikke de utestede dele. D præsenteres for værktøjet, og aflæser med det samme, de utestede dele. D meddelte umiddelbart, at han både synes konceptet om code-coverage var en god ide, og at vores implementering gjorde det meget nemt, overskueligt og hurtigt at overskue, hvor meget ens tests dækker ens gode.

D takkes for sin deltagelse.

D er også programmeringsteknisk, men har abejdet i mindre grad med problematikkerne før. Her er det endnu mere tydeligt, at værktøjet gjorde en forskel.

Svagheder ved undersøgelsen kunne være, at programeksemplet ikke blev udviklet organisk, eller at det ikke var forsøgspersonerne dem selv, der udviklede programmet. Dette kunne have forvrænget resultaten, da udviklerne der har udviklet koden, ofte har nemmere ved at overskue koden, en andre udviklere. Derudover blev programeksemplet udviklet af udviklerne af produktet. Dette kan give muligheder for, at udviklerne laver et program, som viser de resultater, de gerne vil se, men ikke reflekterer virkeligheden.

Vi konkluderer ud fra undersøgelsen. At vores code-coverage-værktøj uden tvivl gør en positiv forskel, for programmører i målgruppen. Selvom svaghederne er relevante, var resultatet fuldstændig klart. Til korrekthedsanalyse giver vores værktøj ikke bare konceptuelt mening. Det er også praktisk anvendeligt.

Problemer under implementering

Softwareudvikling er en disciplin i konstant problemløsning. Derfor er vi stødt på og har arbejdet med mange forskellige problemer. Vi vil nedenfor undersøge specifikke eksempler på problemer vi har oplevet under udviklingen af produktet.

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Process af implementering

Vi startede projektet med finde på et emne, hvor vi kunne lave et softwareprodukt. Dette valgte vi, da det både er vores interesse og ekspertise. Specifikt undersøgte vi problemer, som involverede kompilerudvikling. Det første stykke tid, bliv brugt på at forklare og diskutere med hinanden internt i gruppen, om de forskellige ideer. Her kom vi frem til et værktøj, der kombinerer et code-coverage-værktøj og et flame-graph-værktøj i et med eget programmeringssprog.

Iterativ udvikling

Derefter gik vi i gang med at undersøge implementering. Vi gjorde dette, ved faktisk at påbegynde implementering med lidt forskellige hurtige designbeslutninger. Dette gjorde vi af 2 årsager. Den første, for at undersøge, om hvorvidt det faktisk var muligt at implementere, og hvor ambitiøse vi kunne tillade os at være, i tiden vi havde til det. Og for det andet, for at danne ideer til design og andre aspekter, vi ville kunne inddrage i projektet.

Herefter lavede vi vores indledende undersøgelser. Disse undersøgelser var til, for at undersøge hvorvidt vores problemstilling var reel, og hvor stor grad, det som værktøjerne løser, faktisk er et problem vi kunne påvise. Vi var nu fastsatte på, hvad projektet skulle omhandle.

Derfra fortsatte vi det egentlige design og implementering af produktet. Grunden til vi både har designet og implementeret samtidig, er fordi vores arbejdsprocess er så iterativ, med så små iterationer, at ikke har betragtet de to som seperate arbejdsopgaver. Vi har kunne gøre dette, da produktet udelukkende består af software. Dette betyder, vi ikke har skulle indkøbe materialer eller lave et fysisk produktet, som vi ikke kunne lave om på. Istedet har vi med software altid mulighed for at ændre softwaren, hvis det er nødvendigt.

En anden grund til at design og implementering har været så tæt i vores process, er fordi en stor del af vores design har ‘levet’ i implementeringen. Eksempelvis har vores data-strukturer, som AST’en eller byte-koden været dokumenteret som enumerationer og opslagstabeller i selve implementeringen. Dette har vi kunne, da koden sig selv, har været tilstrækkelig information for os internt i gruppen. Det har haft den fordel, at hvis vi ændrede implementeringen, så blev specifikationen ændret i samme handling.

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Konklusion

Vi har undersøgt omkostningerne for softwareudvikling. Vi har kigget på specifikke dele af udviklingsprocessen af software, der bidrager til omkostningerne og hvordan disse dele har indflydelse på omkostningerne. Vi har konkluderet at kvalitet er en væsentlig faktor i omkostningerne for software udvikling, hvilket betyder, øget kvalitet i software kan sænke omkostningerne. Vi har konkluderet at afviklingstid og korrekthed af software er væsentlige aspekter af kvalitet for software. Vi har undersøgt, i hvilke tilfælde disse 2 aspekter har indflydelse på omkostningerne. Her har vi eksempelvis kigget på cloud-, desktop-, embedded- og mobil-software, og hvordan de 2 aspekter uafhængigt af hinanden har mere og mindre indflydelse på hvilke slags software. Derfra har vi kigget på metoder, udviklere kan bruge til at håndtere de 2 aspekter. Og så har vi kigget på værktøjer, som assisterer udviklere i at bruge disse metoder. Vi har udvalgt de 2 værktøjer, code-coverage og flame-graph til at assistere med henholdsvis korrekthed og afviklingstid. Derfra har vi fortsat med følgende problemformulering: Hvordan kan man lave et værktøj, til at øge kvaliteten af software?

Vi har udført en målgruppeanalyse, for at undersøge hvilke slags personer, disse problematikker er relevante for, og hvilke slags personer, produktet burde designet til. Her konkluderede vi, at vores primære målgruppe er programmeringstekniske softwareudviklere, og at vores sekundære målgruppe er ikke-tekniske softwareudviklere. Ud fra problemanalysen og målgruppeanalysen har vi udført en markedsanalyse, for at undersøge eksisterende værktøjer, som løser samme problem. Her har vi fundet et håndfuld værktøjer, som løser samme problem, men på utilfredsstillende måder. Vi valgte derfor, at implementerer vores egne værktøjer.

Vi udførte nogle indledende undersøgelser og ud fra det en idegenereringsprocess. Her undersøgter vi, hvor brugbart sådanne produkt ville være og hvad behovende for produktet var. Her kom vi frem til, at vores produkt burde være en all-in-one integreret løsning, som indeholder både code-coverage- og frame-graph-værktøjer. Produktet burde have sit eget sprog, sin egen kompiler og sit eget runtime-miljø. På denne måde, ville vi kunne tilbyde værktøjernes anvendelse med minimal yderligere opsætning.

Ud fra dette, har vi designet et produkt, som fremhæver de konceptuelle aspekter i code-coverage og flame-graph, men også præsenterer dem på en intuitiv, brugervenlig og effektiv måde. Vi har designet et programmeringssprog til vores produkt, som er simpelt at implementere og giver mening for målgruppen.

Vi har implementeret en kompiler, en runtime og en webapp-brugergrænseflade, som kan afvikle programmer i programmeringsproget og anvende code-coverage og flame-graph. Vi har anvendt forskellige datastrukture og dataformer, til at implementere vores program, på hensigtsmæssige måder. Vi har anvendt teknisk viden og innovative løsninger til at implementere en runtime, som kan afvikle koden, samtidig med at understøtte værktøjerne. Og så har vi anvendt designprincipper og heuristikker til at designe og implementere en effektiv og elegant brugergrænseflade.

Med det implementerede produkt har vi udført evaluerende undersøgelser. Her har vi konkluderet, ikke bare, at det konceptuelle bag code-coverage og flame-graph giver mening, men også at vores implementering af vores produkt er intuitivt og anvendeligt på de relevante problemer. Derved har vi både undersøgt og fundet ud af hvordan kan man lave et værktøj, til at øge kvaliteten af software?

Perspektivering

I en talk til Cpp-con i 2022 mente Herb Sutter man kunne gøre programmeringssproget C++ 10x simplere og mere sikkert (kilde 1). Problematikken talen omhandlede er, at 90% af fejl i programmer skrevet i C++ stammer fra den samme lille kategori. Denne slags fejl er hukommelse- og ressourcefejl, hvor hukommelse og ressourcer bruges forkert af programmet, på grund af fejl introduceret i udviklingen af programmet.

Grunden til at folk bruger C++ er ofte, for at minimere afviklingstiden af deres programmer. Altså er C++ en af de implementeringsværktøjer, man kan vælge, til at sænke afviklingstiden. Vi vurderer, at disse problematikker er relaterede til projektet problemer, i det vi også håndtere afviklingstid og korrekthed.

Herb Sutters pointere, at de 90% af fejlene kan undgås. Dette kan man gøre, fordi det er muligt at designe et programmeringssprog, så man helt garenterer, det ikke kan ske. Dette har vi diskuteret en smule i projektets problemanalyse. Det er ikke kun Herb Sutter, der siger dette. Faktisk har det været en trend de sidste par år. Trenden går på, at udvikle ressourcesikre og hukommelsessikre programmeringssprog. Dette kaldes ofte memory safe.

Dette er en anden tilgang end vores løsning. I denne tilgang, designer et sprog, som ved hjælp af en kompiler, der implementerer det designe sprog, kan garentere, at man ikke kan skrive fejlagtige

programmer med præcist disse slags fejl. Dette er til sammenligning med sprog, som det vores produkt består af, som istedet bare giver bedre muligheder, for at håndtere fejlene.

Fordi fejlene slet ikke kommer ind i koden, vil koden stadig have en høj afviklingstid. Dette er til sammenligning med vores produkt, som er afhængig af, at programmet afvikles på en runtime, som er langsomere for afviklingstid. Tilgengæld, som vi diskuterede i problemanalysen, vil programmer være sværre at skrive i programmeringssprog, som garentere sikkerhed for disse fejl. Dette er fordi, kompilere for disse sprog, har brug for total indsigt i forskellige aspekter af kodens struktur. Dette gør, at programmøren skal overholde mere stramme regler, end ellers, som gør udviklingen sværere og derfor også mere omkostningsfuld. Dette kan sammenlignes med vores produkt, som benytter en runtime, og derfor ikke har de samme strenge krav, for inputprogrammet.

I forhold til korrekthed. Vores produkt hjælper med at skrive automatiserede tests. Med automatiserede tests, er det muligt at fange alle slags fejl. Dette er fordi automatiserede tests reelt kigger på, hvad der faktisk sker, når koden afvikles. Dette er til sammenligning med disse programmeringssprog, hvor kompileren ikke afvikler koden, istedet er der bare et sæt regler programmøren skal overholde, som så gør, at kompileren kan give nogle garentier. Men dette gælder kun for de 90% af fejl. De resterende 10% omhandler alt muligt andet en fejlhåndtering af ressourcer. Disse problematikker vil disse programmeringssprog ikke hjælpe med i samme grad.

Vores vurdering er, at der er nogle tilfælde, hvor Herb Sutters løsning er bedre. Dette er eksempelvis tilfælde, hvor både afvklingstid og korrekthed er kritisk. Der er også tilfælde, hvor værktøjer som vores produkt ville være bedre. Eksempelvis hvis højere udviklingsomkostninger er uacceptabelt. Derfor er det på case-by-case basis, hvilken metode der er bedst. Værktøjer som code-coverage og flame-graph har været tilstede længere tid, end udviklingen i disse slags sprog. Derfor er der også mulighed for, at disse slags sprog bliver den bedste løsning i fremtiden.

Litteratureliste

1. Sutter, H., *My CppCon 2022 talk is online: “Can C++ be 10x simpler & safer … ?”*, https://herbsutter.com/2022/09/19/my-cppcon-2022-talk-is-online-can-c-be-10x-simpler-safer/, besøgt 20-12-2024

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Bilag

Bilag 1 – Eksempelprogram til afsluttende undersøgelser

Output i terminalen:

```
Runtime at 127.0.0.1:13370
 * test: should convert zero -> ok
 * test: should convert decimal -> ok
 * test: should convert binary -> ok
Devtools at http://localhost:8000/
 * test: should convert octal -> ok
 * test: should fail -> ok
```

Program:

```
fn string_to_int_impl(text: string) -> int {
    let base_2_digits = "01";
    let base_8_digits = base_2_digits + "234567";
    let base_10_digits = base_8_digits + "89";
    let base_16_digits = base_10_digits + "abcdef";

    let len = string_length(text);
    if len == 0 {
        return -1;
    }
    if text[0] == "0"[0] {
        if len == 1 {
            0
        } else if text[1] == "b"[0] {
            parse_digits(string_slice(text, 2, -1), 2, base_2_digits)
        } else if text[1] == "x"[0] {
            parse_digits(string_slice(text, 2, -1), 16, base_16_digits)
        } else {
            parse_digits(string_slice(text, 1, -1), 8, base_8_digits)
        }
    } else {
        parse_digits(text, 10, base_10_digits)
    }
}

fn parse_digits(text: string, base: int, digit_set: string) -> int {
    let val = 0;
    let len = string_length(text);
    for (let i = 0; i < len; i += 1) {
        let ch = text[i];
        if not string_contains(digit_set, ch) {
            return -1;
        }
        val = val * base;
        val += char_val(ch);
    }
    val
}

fn char_val(ch: int) -> int {
    if ch >= "0"[0] and ch <= "9"[0] {
        ch - "0"[0]
    } else if ch >= "a"[0] and ch <= "f"[0] {
        ch - "a"[0] + 10
    } else {
        -1
```

```
    }
}

fn test_string_to_int_impl() -> bool {
        test("should convert zero",      assert_int_equal(string_to_int_impl("0"), 0))
    and test("should convert decimal",   assert_int_equal(string_to_int_impl("10"), 10))
    and test("should convert binary",    assert_int_equal(string_to_int_impl("0b110"), 6))
    and test("should convert octal",     assert_int_equal(string_to_int_impl("071"), 57))
    //and test("should convert hex",       assert_int_equal(string_to_int_impl("0xaa"), 170))
    and test("should fail",              assert_int_equal(string_to_int_impl("john"), -1))
    //and test("should fail",              assert_int_equal(string_to_int_impl(""), -1))
}

fn assert_int_equal(value: int, target: int) -> bool {
    if value != target {
        println("assertion failed: " + itos(value) + " != " + itos(target));
        return false;
    }
    true
}

fn test(name: string, assertion: bool) -> bool {
    println(" * test: " + name + " -> " + if assertion { "ok" } else { "failed" });
    assertion
}

fn main() {
    let ok = test_string_to_int_impl();
    if not ok {
        println("tests failed!");
    }
}

// ...

fn print(msg: string) #[builtin(Print)] {}
fn println(msg: string) { print(msg + "\n") }
fn string_push_char(str: string, value: int) -> string #[builtin(StringPushChar)] {}
fn string_length(str: string) -> int #[builtin(StringLength)] {}
// ...
fn string_slice(str: string, from: int, to: int) -> string {
    let result = "";
    let len = string_length(str);
    let abs_to =
        if to >= len { len }
        else if to < 0 { len + to + 1 }
        else { to };
    for (let i = from; i < abs_to; i += 1) {
        result = string_push_char(result, str[i]);
    }
    result
}

fn string_contains(str: string, ch: int) -> bool {
    let len = string_length(str);
    for (let i = 0; i < len; i += 1) {
        if str[i] == ch {
            return true;
        }
    }
    false
}
// …
```

T. P. Hollebeek, S. F. Jakobsen, M. T. Kongsted
Teknikfag B - 20ia

Code coverage – Performance-view:

```
fn string_to_int_impl(text: string) -> int {
    let base_2_digits = "01";
    let base_8_digits = base_2_digits + "234567";
    let base_10_digits = base_8_digits + "89";
    let base_16_digits = base_10_digits + "abcdef";
    let len = string_length(text);
    if len == 0 {
        return -1;
    }
    if text[0] == "0"[0] {
        if len == 1 {
            0
        } else if text[1] == "b"[0] {
            parse_digits(string_slice(text, 2, -1), 2, base_2_digits)
        } else if text[1] == "x"[0] {
            parse_digits(string_slice(text, 2, -1), 16, base_16_digits)
        } else {
            parse_digits(string_slice(text, 1, -1), 8, base_8_digits)
        }
    } else {
        parse_digits(text, 10, base_10_digits)
    }
}
fn parse_digits(text: string, base: int, digit_set: string) -> int {
    let val = 0;
    let len = string_length(text);
    for (let i = 0; i < len; i += 1) {
        let ch = text[i];
        if not string_contains(digit_set, ch) {
            return -1;
        }
        val = val * base;
        val += char_val(ch);
    }
    val
}
fn char_val(ch: int) -> int {
    if ch >= "0"[0] and ch <= "9"[0] {
        ch - "0"[0]
    } else if ch >= "a"[0] and ch <= "f"[0] {
        ch - "a"[0] + 10
    } else {
        -1
    }
}
fn test_string_to_int_impl() -> bool {
        test("should convert zero",       assert_int_equal(string_to_int_impl("0"), 0))
    and test("should convert decimal",    assert_int_equal(string_to_int_impl("10"), 10))
    and test("should convert binary",     assert_int_equal(string_to_int_impl("0b110"), 6))
    and test("should convert octal",      assert_int_equal(string_to_int_impl("071"), 57))
    //and test("should convert hex",        assert_int_equal(string_to_int_impl("0xaa"), 170))
    and test("should fail",               assert_int_equal(string_to_int_impl("john"), -1))
    //and test("should fail",               assert_int_equal(string_to_int_impl(""), -1))
}
fn assert_int_equal(value: int, target: int) -> bool {
    if value != target {
        println("assertion failed: " + itos(value) + " != " + itos(target));
        return false;
    }
    true
}
fn test(name: string, assertion: bool) -> bool {
    println(" * test: " + name + " -> " + if assertion { "ok" } else { "failed" });
```

```
    assertion
}
fn main() {
    let ok = test_string_to_int_impl();
    if not ok {
        println("tests failed!");
    }
}
//
fn print(msg: string) #[builtin(Print)] {}
fn println(msg: string) { print(msg + "\n") }
fn string_push_char(str: string, value: int) -> string #[builtin(StringPushChar)] {}
fn string_length(str: string) -> int #[builtin(StringLength)] {}
// ...
fn string_slice(str: string, from: int, to: int) -> string {
    let result = "";
    let len = string_length(str);
    let abs_to =
        if to >= len { len }
        else if to < 0 { len + to + 1 }
        else { to };
    for (let i = from; i < abs_to; i += 1) {
        result = string_push_char(result, str[i]);
    }
    result
}
fn string_contains(str: string, ch: int) -> bool {
    let len = string_length(str);
    for (let i = 0; i < len; i += 1) {
        if str[i] == ch {
            return true;
        }
    }
    false
}
```

Code coverage – Test-view:

```
fn string_to_int_impl(text: string) -> int {
    let base_2_digits = "01";
    let base_8_digits = base_2_digits + "234567";
    let base_10_digits = base_8_digits + "89";
    let base_16_digits = base_10_digits + "abcdef";
    let len = string_length(text);
    if len == 0 {
        return -1;
    }
    if text[0] == "0"[0] {
        if len == 1 {
            0
        } else if text[1] == "b"[0] {
            parse_digits(string_slice(text, 2, -1), 2, base_2_digits)
        } else if text[1] == "x"[0] {
            parse_digits(string_slice(text, 2, -1), 16, base_16_digits)
        } else {
            parse_digits(string_slice(text, 1, -1), 8, base_8_digits)
        }
    } else {
        parse_digits(text, 10, base_10_digits)
    }
}
fn parse_digits(text: string, base: int, digit_set: string) -> int {
    let val = 0;
    let len = string_length(text);
    for (let i = 0; i < len; i += 1) {
```

```
        let ch = text[i];
        if not string_contains(digit_set, ch) {
            return -1;
        }
        val = val * base;
        val += char_val(ch);
    }
    val
}
fn char_val(ch: int) -> int {
    if ch >= "0"[0] and ch <= "9"[0] {
        ch - "0"[0]
    } else if ch >= "a"[0] and ch <= "f"[0] {
        ch - "a"[0] + 10
    } else {
        -1
    }
}
fn test_string_to_int_impl() -> bool {
        test("should convert zero",       assert_int_equal(string_to_int_impl("0"), 0))
    and test("should convert decimal",    assert_int_equal(string_to_int_impl("10"), 10))
    and test("should convert binary",     assert_int_equal(string_to_int_impl("0b110"), 6))
    and test("should convert octal",      assert_int_equal(string_to_int_impl("071"), 57))
    //and test("should convert hex",        assert_int_equal(string_to_int_impl("0xaa"), 170))
    and test("should fail",               assert_int_equal(string_to_int_impl("john"), -1))
    //and test("should fail",               assert_int_equal(string_to_int_impl(""), -1))
}
fn assert_int_equal(value: int, target: int) -> bool {
    if value != target {
        println("assertion failed: " + itos(value) + " != " + itos(target));
        return false;
    }
    true
}
fn test(name: string, assertion: bool) -> bool {
    println(" * test: " + name + " -> " + if assertion { "ok" } else { "failed" });
    assertion
}
fn main() {
    let ok = test_string_to_int_impl();
    if not ok {
        println("tests failed!");
    }
}
// ...
fn print(msg: string) #[builtin(Print)] {}
fn println(msg: string) { print(msg + "\n") }
fn string_push_char(str: string, value: int) -> string #[builtin(StringPushChar)] {}
fn string_length(str: string) -> int #[builtin(StringLength)] {}
// ...
fn string_slice(str: string, from: int, to: int) -> string {
    let result = "";
    let len = string_length(str);
    let abs_to =
        if to >= len { len }
        else if to < 0 { len + to + 1 }
        else { to };
    for (let i = from; i < abs_to; i += 1) {
        result = string_push_char(result, str[i]);
    }
    result
}
fn string_contains(str: string, ch: int) -> bool {
    let len = string_length(str);
    for (let i = 0; i < len; i += 1) {
```

```
        if str[i] == ch {
            return true;
        }
    }
    false
}
```

Flame-graph:

Bilag 2 – Projektet kode

Følgende er sammensat, som en PDF-fil. Derfor vil linjenumrene være ukorrekte herfra.