**Resume**  
Nexttech er et af Nordens førende centre inden for industriel 3D-print og tilbyder en bred vifte af teknologier og materialer til additiv fremstilling. Virksomheden står over for udfordringer med omkostningsstyring, da hver printcyklus medfører unikke omkostninger afhængigt af faktorer som materialetype, udskrivningstid og emnets kompleksitet. Samtidig kræver efterbehandling ofte ekstra ressourcer, hvilket kan komplicere produktionsprocessen yderligere.

Projektet sigter mod at udvikle en digital løsning, der optimerer og visualiserer anvendelsen af 3D-printere i en industriel kontekst. Systemet skal give medarbejdere og kunder mulighed for at vælge antal printere og materialer, samtidig med at det præsenterer en grafisk fremstilling af deres valg. Dette skal ske gennem en brugervenlig grænseflade, der integreres med en database til styring af information om printere, materialer og processer.

Med udgangspunkt i systematisk IT-arkitektur og teknologier som SQL-databaser og Python-programmering er løsningen designet til at sikre skalerbarhed, fleksibilitet og brugertilfredshed. Fokus er også på omkostningsoptimering ved at inkludere specifikationer for materialer og produktionsprocesser, så kunder og medarbejdere får bedre indsigt i valgmuligheder og tilhørende omkostninger. Projektet er struktureret med en målhierarkisk tilgang og har som mål at levere pålidelige resultater, der kan bidrage til en mere effektiv og gennemsigtig anvendelse af 3D-printerteknologi i industrien.

**Projektbeskrivelse**

Dette projekt omhandler udvikling af et omkostningsberegningssystem til Nexttech, der optimerer omkostningsstyringen i virksomhedens 3D-printproduktion. Systemet skal kunne integrere data fra flere teknologier og materialer, samt tage højde for de mange variabler, der påvirker omkostningerne i 3D printprocessen. Målet er at skabe et pålideligt, fleksibelt og brugervenligt system, der kan understøtte beslutningstagning og sikre præcise beregninger af produktionsomkostninger. Projektet kræver integration af data fra forskellige kilder, herunder 3D-printprocesser, materialer og forskellige maskiner. Det er nødvendigt at bygge et system, der kan håndtere disse data effektivt og samtidig sikre, at systemet kan tilpasse sig nye teknologier og materialer, efterhånden som industrien udvikler sig.

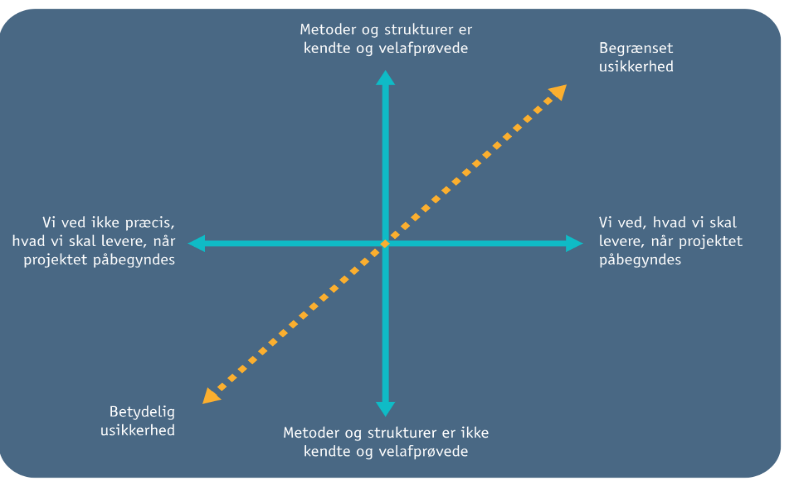
Målet med dette projekt er at levere et system, der nøjagtigt kan beregne omkostningerne for ethvert printprojekt. Dette er essentielt for at undgå fejl i budgettering og prissætning, da omkostningerne til maskiner og materialer kan være betydelige og variable.

Et af hovedmålene med projektet er at sikre, at systemet er intuitivt og nemt at bruge for medarbejderne. Visualisering af omkostningsdata, såsom diagrammer og grafer, gør det lettere at forstå omkostningsstrukturen og hjælper medarbejderne med at foretage de nødvendige justeringer hurtigt.

Systemet skal være fleksibelt nok til at tilpasse sig produktionsændringer og fremtidige teknologiske fremskridt. Det skal kunne udvides og opdateres uden større ændringer i systemets kernestruktur, hvilket sikrer, at det forbliver relevant i et dynamisk marked.

Da systemet vil indeholde følsomme data som økonomiske beregninger og historik, er sikkerhed en vigtig del af projektet. Adgangskontrolfunktioner sikrer, at kun autoriserede brugere kan få adgang til kritiske data og funktioner, hvilket beskytter virksomheden og dens kunder.

Karakteristikaene ved dette projekt viser, at det ikke kun er et teknisk IT-projekt, men kræver en dyb forståelse af 3D-printteknologi og produktionsstyring. Med fokus på præcisionsberegning, brugervenlighed og fleksibilitet, sigter projektet mod at skabe en løsning, der giver Nexttech et solidt fundament for økonomisk og driftsmæssig optimering.

**Projekts kendetegn**

**Metoder og strukturer**

Som IT-arkitektstuderende med et projekt i industriel 3D-printning anvender vi testede metoder og strukturer inden for database- og systemarkitektur. Mange af de teknologier og processer, vi beskriver (såsom SQL-databaser og 3D-printprocesser) er kendte og kan anvendes på en struktureret måde.

**Klarhed omkring levering**

Formålet og rammerne virker ret klare, selvom de kan være fleksible i detaljer og specifikationer. Projektet har specifikke mål og leverancer, såsom optimering af omkostningsregnskab men vi har ikke en ret præcis ide om, hvad der skal præsenteres som en it-studenterne.

Vi vurderede vores projektet placeret i øverste og venstre side af diagrammet, hvilket viser, at vi ikke har et generelt og præcist overblik over, hvad vi skal levere, når projektet starter. Selvom vi måske har en klar idé om målene, kan der være elementer i projektet, der er udfordrende for os som it-studerende. Projekter i læringsmiljøet rummer normalt usikkerhed, fordi der ofte er aspekter, som ikke er afprøvet for en elev. Så vi betragter projektet i høj grad som et pilotprojekt, hvor metoderne endnu ikke er helt kendte af os. Som følge heraf kan man sige, at projektet er placeret i toppen af ​​diagrammet og i venstre side, hvor vi står med usikkerhed.

**Problemformulering**

Hvordan kan Nexttech skabe et pålideligt og fleksibelt omkostningsberegningssystem, der tager højde for de mange forskellige parametre i 3D-printprocessen? Dette system skal integrere data fra flere teknologier og materialer og kunne justeres i forhold til ændringer i produktionsprocesserne. Desuden skal det være intuitivt og nemt at bruge for medarbejderne, så det bliver et effektivt værktøj i daglig drift.

**Problemstillinger**

Dette projekt står over for flere udfordringer, som skal tackles for at opnå et effektivt system. For det første skal systemet være i stand til at styre og integrere data fra forskellige kilder, som kan variere afhængigt af materialetype, procesomkostninger og maskinydelse. Dette kræver en robust databasestruktur, der kan organisere og opdatere data effektivt.

Derudover er nøjagtighed i omkostningsberegninger afgørende, da høje omkostninger forbundet med maskiner og materialer forhindrer fejl i budgettering og prissætning. Systemet skal kunne redegøre for direkte og indirekte omkostninger og sikre, at ingen væsentlige omkostninger overses.

For at medarbejderne kan bruge systemet effektivt, er brugervenlighed en nøglefaktor. Designet skal være intuitivt, og visualiseringer som diagrammer og grafer skal gøre det nemt for brugerne at forstå omkostningsstrukturen og hurtigt foretage de nødvendige justeringer.

Derudover skal systemet være fleksibelt og skalerbart. 3D-printindustrien udvikler sig hurtigt, og nye maskiner, materialer og processer kan ændre omkostningsstrukturen. Derfor bør systemet kunne tilpasses og udvides i takt med disse ændringer uden omfattende omstruktureringer.

Endelig har systemet brug for en stærk sikkerhedsfunktion, da det håndterer følsomme økonomiske data og historik. Adgangskontrol bør implementeres, så kun autoriserede brugere kan få adgang til kritiske funktioner og data i systemet.

Udvikling af et omkostningssystem for Nexttech, dette projekt involverer tekniske og strukturelle udfordringer, men giver også virksomheden et bedre økonomisk overblik over 3D-printproduktion. Et effektivt system, der integrerer de nødvendige data og giver nøjagtige beregninger, optimerer ikke kun omkostningsstyringen, men bidrager også til mere præcise beslutningsgrundlag og forbedret økonomistyring. Brugervenlige funktioner, såsom omkostningsvisualisering, hjælper personalet med at forstå og reagere på realtidsdata, mens systemets fleksibilitet sikrer, at det kan holde trit med teknologiske fremskridt og fremtidige behov.

På den måde kan Nexttech opnå mere effektiv ydeevne og økonomi, hvilket er afgørende i en konkurrencedygtig og hastigt udviklende industri som 3D-print.

**Målhierarki:**

Målhierarki sikrer, at de vigtigste interessenter i og omkring projektet kan blive enige om, hvorfor projektet kører, og hvilke konkrete resultater der forventes af det. Sammenhængen mellem disse to aspekter sikrer, at projektresultaterne understøtter de forretningsmæssige og organisatoriske behov, der har foranlediget projektet. At have et præcist mål hjælper med at gøre beslutninger nemmere og mere præcise under de mulige ændringer af projektet i implementeringsfasen. Målhierarki er ikke et simpelt værktøj til at begynde at arbejde på et projekt, men når dette hierarki er klart defineret i forhold til mål, delmål og leverancer, vil resten af projektarbejdet være meget mere håndgribeligt, når du klart ved, hvad din overordnede mål er, og hvad der skal gøres. Det bliver meget nemmere at vurdere, hvordan ændringerne vil påvirke projektet.

Målhierarki består af tre elementer: mål, delmål og leverancer.

Et billede, der indeholder tekst, diagram, skærmbillede, Plan

Automatisk genereret beskrivelseEt billede, der indeholder tekst, skærmbillede, diagram, Font/skrifttype

Automatisk genereret beskrivelse

**Mål:** Det overordnede formål med projektet, som beskriver, hvad projektet skal opnå på et strategisk niveau. Et skarpt mål skaber klarhed og retning for projektet. Det hjælper med at definere, hvad projektet skal opnå, og hvordan succes ser ud. Når målet er klart og specifikt, bliver det lettere at planlægge, prioritere ressourcer og evaluere fremskridt. Desuden gør et klart mål det muligt at håndtere ændringer og risici mere effektivt, fordi vi ved, hvad projektets kerneformål er.

Øverst i vores målhierarkiet står "Effektivisering af ressourcer", som repræsenterer projektets overordnede mål. Det er det strategiske formål, der driver projektet. Dette mål er bredt og angiver den ønskede effekt: at optimere brugen af ressourcer på tværs af hele systemet.

**Delmål:**

Under hovedmålet er der tre centrale delmål:

* **Optimering af omkostningsberegning**: Fokus på præcise og effektive metoder til at beregne omkostninger.
* **Ressourcebesparelse:** At reducere brugen af ressourcer ved at eliminere ineffektivitet.
* **Bedre overblik over produktionsomkostninger (historisk):** At sikre klarhed og adgang til tidligere data, så beslutninger kan træffes på et informeret grundlag.

Disse delmål er specifikke og relaterer til det overordnede mål. De repræsenterer konkrete aspekter af det strategiske mål, som kan måles og evalueres.

**Leverance:** De konkrete produkter eller tjenester, der skal leveres for at nå målene.

Nederst i hierarkiet ses de konkrete leverancer, som understøtter opnåelsen af delmålene:

* **IT-løsning til omkostningsberegning leveret:** En teknologisk løsning, der sikrer præcise beregninger.
* **Brugervenligt interface lanceret:** Et system, der er nemt for medarbejdere at bruge.
* **Fleksibel og skalerbar løsning**: En løsning, der kan tilpasses fremtidige behov.
* **Nye arbejdsgange etableret**: Forbedrede processer til at understøtte ressourcebesparelser.
* **Træning af medarbejdere gennemført**: Uddannelse, der sikrer, at medarbejdere kan bruge de nye værktøjer effektivt.

Leverancerne er direkte relateret til delmålene. Hver leverance bidrager til at opnå et eller flere delmål og understøtter samtidig det overordnede mål.

Diagrammet viser en klar og struktureret tilgang til at nå vores projektmål. Det forbinder det strategiske mål med konkrete handlinger og resultater. Hvis alle leverancer implementeres effektivt, vil projektet opnå sine delmål og dermed også det overordnede mål om ressourceeffektivisering.

vores målhierarki viser en struktureret tilgang til projektets overordnede mål, delmål og leverancer. Dette diagram illustrerer, hvordan de forskellige niveauer er forbundet og understøtter hinanden for at sikre projektets succes.

Ved at følge SMART-princippet kan vi minimere risikoen for fejl, misforståelser og forsinkelser, hvilket bidrager til et mere effektivt og vellykket projekt. I vores projekt kan vi godt se, at alle delmål og leverancer er klart definerede **(S)**. Succes kan måles ved at vurdere, hvor meget omkostningsberegningen er forbedret, eller hvor effektive de nye arbejdsgange er **(M).** Hierarkiet sikrer, at interessenter er enige om målene**(A).** Leverancerne er gennemførlige inden for projektets ressourcer**(R).** Hvert trin i hierarkiet kan tilknyttes deadlines for at sikre fremdrift**(T).**

Ved at sikre, at målene er både specifikke, målbare, aftalte, realistiske og tidsbestemte, bliver det lettere at planlægge, analysere risici og håndtere ændringer. For eksempel giver det jer mulighed for hurtigt at vurdere konsekvenserne af ændringer i tid eller ressourcer og tage beslutninger baseret på projektets overordnede mål og prioriteter. Denne tilgang sikrer også, at projektorganisationen kan arbejde effektivt og proaktivt mod at opnå både leverancer og udbytter, hvilket gør det muligt at nå det overordnede mål om ressourceeffektivisering.

# Projektmodel og plan

Når man påbegynder et projekt, er det vigtigt at lave den rigtige projektmodel. Projektmodellen skal være en struktureret ramme, der hjælper fra start til slut og, sikre at man opnår projektet leverance effektivt og med kvalitet. Den rette model skal hjælpe med at få et overblik over mål, roller, ressourcer og tidsplaner. Valget af den rette model er baseret på projektets type, omfang, behov og ens erfaring. Valget af modellen kan have betydning for, om man opnår succes med sit projekt eller ej.

## Valg af metode

I forbindelse med vores projekt, har vi vurderet forskellige tilgange og valgt en agil tilgang med Scrum frem for den traditionelle vandfaldsmodel. Valget er baseret på projekttypen og vores erfaring.  
Projektet er et IT/softwareudviklingsprojekt, hvor vi skal udvikle et nyt system fra bunden. Dette indebærer en lav struktur, samt at den endelige løsning ikke var klart defineret fra starten. På grund af disse forhold, kombineret med vores manglende erfaring, har vi haft brug for en fleksibel metode, som vi kan tilpasses under projektet.  
Den traditionelle vandfaldsmodel kræver, at man afslutter hver fase fuldstændigt, før man fortsætter til den næste, og den er ikke god til at håndtere justeringer under projektet. Det ville være dårlig metode, grundet vores projekt kræver vi kan være fleksible og lave løbende justeringer.  
Ved at vi har valgt en agil tilgang, og metoden Scrum, kan vi arbejde iterativt og justerer i processen bedre. Scrum giver os mulighed for at dele arbejdet op i mindre dele, som kan tilpasses løbende.

## Processen

I starten af projektet fik vi ikke lagt en ordentlig plan De første par opgaver blev løst en efter en uden overblik over hele projektet. Vi mistede meget nemt overblikket over hvilket opgaver der skulle laves, og var meget usikker på projektet. Syv uger inden projektet skulle være færdigt, måtte vi indse at vi skulle have lagt en plan, og gøre brug af Scrum. Vi startede med at få oprettet et bord i Miro. Miro er en digital samarbejdsplatform, som tilbyder forskellige skabeloner, som vi har kunne tilpasse vores behov, herunder også skabeloner for scrum.

Vi begyndte Scrum processen med at få oprettet en backlog, planlagte to ugers sprint og besluttede for at holde et Review i slutningen af hvert sprint. Den første uge blev brugt på at oprette vores board i Mio, for at skabe os et overblik, men vi så det ikke som en del af et sprint, da vi ikke var begyndt på arbejdet endnu.

Sprint 1 (uge 45-46)I det første sprint overførte vi opgaver fra backloggen og fordelte dem mellem os. Vi fokuserede på de vigtigste opgaver, som var:

* Undersøgelse af hvilken grafisk løsning der skulle laves
* Problemanalyse
* Begyndelsen af kostberegneren
* Hvad kendetegner vores projekt
* Login side
* Risikoanalyse
* Gruppens profiler og roller
* Oprettelse af database

På møderne i sprintet blev der lavet aftaler om GitHub, ansvaret for opgaver og hvordan vi holder vores møder. I slutningen af sprint fandt vi ud af at opgaverne var lidt for store, eller vi havde for mange. Vi aftalte derfor at opgaverne at opgaver som ikke var færdige, måtte overføres til næste sprint.  
Under vores Review fandt vi ud af at vi mangler overblik over sprintet, men vi så også positive ting, som at vi er kommet godt i gang med projektet, at vi er begyndt på konkrete opgaver og vi er gode til at give hinanden feedback og inputs. Noget af det vi fik lært under sprintet, var at en person ikke skal have for mange opgaver, ikke lave flere opgaver til samme sprint og at det kan være svært at gennemskue omfanget af en opgave på forhånd.

### Sprint 2 (uge 47-48)

I andet sprint færdiggjorde vi opgaverne fra det forrige, og begyndte på nye. De nye opgaver vi gik i gang med var:

* Rapporten
* Undersøgelse af sideskifte i GUI
* Historiksiden

Vi oplevede en udfordring, da to af os havde løst den samme opgave, selv om vi var opmærksomme på dette fra vores risikoanalyse. Dette blev taget op på vores møder uden konflikter. Under vores review blev problemer med kommunikationen taget op, og vi at mangler programmeringsfærdigheder. På den anden side følte vi, at programmet begyndte at tage form, og vi kan se fremgang i projektet. En af de ting, vi lærte, var, at det kan være udfordrende at kode et program uden meget erfaring, og at god kommunikation er vigtig.

Sprint 3 (uge 49-50)I sidste sprint opdagede vi, at flere vigtige opgaver manglede i vores backlog/sprint planning. Dette førte til at vi mistede overblikket, og troede vi ikke kunne få det gennemført som et program. Efter samtaler og ekstra arbejdstimer lykkedes det os at sammensætte koden og færdiggøre opgaverne. Vi planlagde at afslutte projektet med et review for at samle alt læring fra projektet.

Sprint reviewsI slutningen af hvert sprint afholdt vi et review, der skulle hjælpe os med at forbedre os til næste. Vi brugte en skabelon fra Miro og delte vores evaluering op i fire kategorier:

* Liked: Hvad gik godt?
* Learned: Hvad lærte vi?
* Lacked: Hvad manglede?
* Longed for: Hvad kunne vi have ønsket?

Vi gennemgik dem en for en og indsatte post-its for hvert punkt. På den måde kunne vi beskrive vores tanker for sprintet. Vi ville skabe overblik over, hvad der fungerede godt i sprintet, og hvad vi kunne forbedre.

RefleksionVed at bruge Scrum som projektstyringsmetode kunne vi organisere, justere og håndtere projektet løbende. Scrum passede godt til vores behov, da vi ofte havde svært ved at forudse kommende opgaver på grund af vores manglende erfaring. Den agile metode gav os mulighed for at rette op på fejl og styre projektet tilbage på rette spor.

Havde vi havde valgt vandfaldsmodellen, ville projektet sandsynligvis ikke være blevet gennemført, da vi ikke kunne tilpasse løbende.

**Hvad fungerede godt?**

* Udledning af opgaver for hvert sprint skabte en fælles forståelse af målene.
* Næsten daglige møder holdt alle opdaterede omkring opgaver.
* Sprint reviews hjalp os med at lære af erfaringer og forbedre os. Dette var vigtigt for at vi kunne tage det med til næste sprint.
* Selv om der opstod udfordringer, håndtere vi dem godt.

**Hvad oplevede vi af udfordringer?**

* Manglende overblik på grund af store og uoverskuelige opgaver.
* Overestimerede hvor lang tid opgaverne ville tage, og blev derfor ofte ikke færdig i sprintet.
* Manglende erfaring med projektstyring og programmering gjorde sprintende mindre effektive.

Selvom dette var vores første projekt ved brug af scrum, fandt vi ud af hvor god en metode det er. Igennem processen lærte vi hvor stor en betydning møder, reviews og planlægning, har for projektet. Vi vil fremadrettet kunne bruge denne erfaring til at forbedre den måde vi kører scrum på, og skabe et stærkere fundament.

**Risikoanalyse**

**Hvad er en risikoanalyse?**

Under et projekt vil der altid opstå uforudsete hændelser, som enten positivt eller negativt kan påvirke projektet. Det er derfor vigtigt at være forberedt på forskellige risici, så de kan håndteres bedre.

**Hvorfor lavede vi en risikoanalyse?**

Vi valgte at lave en risikoanalyse for at identificere, analysere og håndtere mulige hændelser, der kan påvirke projektet. Analysen skal hjælpe os med at håndtere risici, hvis der skulle opstå uforudsete hændelser.

På grund af vores manglende erfaring og svage datagrundlag vil vores risikoanalyse være mindre præcis, og vi kan risikere at overse kritiske risici. Vi er derfor opmærksomme på, at der stadig kan opstå hændelser, vi ikke er forberedte på.

**Værktøjer**

I forbindelse med analysen har vi brugt to værktøjer. Vi har lavet et risikoregister, som hjælper os med at registrere og beskrive de enkelte hændelser. Derudover anvender vi en risikomatrix til at vurdere risikoværdier, som derefter kan registreres i risikoregistret. Disse værktøjer giver os en god kombination til at analysere, dokumentere og håndtere risici.

Vores risikoregister består af kolonnerne: ID, dato, årsag, hændelse, konsekvens, sandsynlighedsværdi, konsekvensværdi, risikoværdi, handlinger, betydning af handling og re-estimering af risikoværdi. I en generel risikoanalyse ville man også forvente kolonner som risikoejer og forventet tidspunkt for hændelsen. Vi har dog valgt ikke at udpege en specifik risikoejer, da vores erfaring er begrænset. I stedet har vi besluttet at håndtere risici i fællesskab for bedre at kunne tage handling. Af samme grund har vi undladt at inkludere forventet tidspunkt for hændelsen.

Et billede, der indeholder skærmbillede, kvadratisk, Rektangel, Farverigt

Automatisk genereret beskrivelse

Risikomatrixen er opbygget med sandsynlighed, konsekvens og risikoværdi. Når vi vurderer sandsynligheden og konsekvensen af en hændelse, får vi en risikoværdi og farve. Farverne grøn, gul og rød giver os et visuelt overblik over, hvilke hændelser der skal håndteres først, og hvilke der kan overvåges. Farverne afspejler også vores risikovillighed. Vi har valgt en middel risikovillighed, da vi er villige til at tage nogle risici for at opnå vores ønskede mål, men samtidig ønsker at minimere større risici, der kan føre til projektet fejler.

Vi har lavet to matrixer for at få et overblik over vurderingerne før og efter handlinger.

**Vores proces**

Det første trin i analysen var at identificere de forskellige risici, der kunne påvirke projektet. Når man identificerer hændelser, er det vigtigt at være konkret, da dette gør det muligt at vurdere og håndtere dem bedre. Vi har både individuelt og i fællesskab tænkt og undersøgt, hvilke hændelser der kunne opstå under projektet.

Efter hændelserne blev identificeret, så vi på årsagerne og konsekvenserne af hver hændelse. Det er vigtigt at forstå, hvorfor en hændelse opstår, og hvilken betydning den har for projektet.

Med årsagen, hændelsen og konsekvensen beskrevet, vurderede vi sandsynligheden og konsekvensværdien for hver hændelse. Til det brugte vi risikomatrixen og vurderede med tal fra 1-5. Risikoværdierne blev derefter registreret i risikoregistret, og gav os er overblik over hvilke der kræver mest opmærksomhed.

Efter at have skabt et overblik over hændelserne og deres vurderinger, gik vi videre til at håndtere dem. For hver hændelse snakkede vi handlinger, der kunne reducere konsekvensen eller sandsynligheden. Valget af handling var baseret på risikoværdien.

* Hvis risikoværdien var høj (12-25), udarbejdede vi en plan B og forebyggede.
* Hvis risikoværdien var lav, nøjedes vi med at overvåge hændelsen.

Efter håndteringen noterede vi, hvordan handlingerne påvirkede sandsynligheden eller konsekvensen, og vi revurderede derefter hændelsernes risikoværdi.

**Kravspecifikation:**

# Prototypen

Vores projekt bygger på at vi skal lave en applikation, som kan lave 3D prints beregninger. Prototypen er opbygget af en GUI applikation som gør brug af en database. For at skabe den grafiske brugerfalse har vi brugt Tkinter og CustomTkinter. Applikationen giver brugere mulighed for at logge ind, oprette brugere, lave 3D printer beregninger og se historikken for beregningerne.

## Moduler & biblioteker

For at kunne bygge vores applikation har brugt en række moduler og biblioteker i Python.

* **Tkinter og CustomTkinter as c**: Bruges til GUI
* **SQLite3**: Databasehåndtering
* **Pillow**: Håndtering af billeder
* **Matplotlib**: Grafer
* **Messagdebox**: Popup vindue med feedback

## Koden

**Opstart af programmet**

**Et billede, der indeholder tekst, Font/skrifttype, skærmbillede

Automatisk genereret beskrivelse**

For at starte programmet bruger vi denne kode.



Koden her fortæller os at programmet skal starte hvis dette er hoved scriptet. Hvis scriptet var blevet importeret ind i et andet, ville det ikke være Main. Dette har vi gjort for at sikre at det er hovedprogrammet som der bliver kørt.



Her oprettes der en instans til appklassen. manufacturing.db er vores database, som bliver taget med videre til App klassen som et argument db\_name.



Mainloop holder vores applikation kørende.

**App klassen**

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype

Automatisk genereret beskrivelse

Klassen app fungerer som hovedvinduet og styrecenteret for programmet. Det er det første vindue som der bliver åbnet, når programmet køres. De andre klasser (LoginMenu, Mainmenu, Adminmenu, GUISetup og Historik) fungere som frames der bliver vist og styret i App.

Et billede, der indeholder tekst, Font/skrifttype, skærmbillede, linje/række

Automatisk genereret beskrivelse

Klassen arver fra en anden klasse c.CTK, som er en klasse fra bibloteket. Det betyder at app automatisk får alle funktioner og metoder der er i customtkinter. Dette bruges til at åbne vinduet for brugerfladen.

Et billede, der indeholder tekst, Font/skrifttype, skærmbillede

Automatisk genereret beskrivelse

Følgende ligner kode bruger brugt til at sætte vinduet titel, størrelse og farvetema.



Her bliver der skabt en ny instans af Loginmenu klassen. Hvilket betyder at den indsætter Login framen på det nuværende vindue. Der bliver videregivet to argumenter som er self og db\_name. Self er instansen for klassen App, og db\_name er vores database.  
Linjen er placeret i klassen App, fordi det første man skal se når man åbner for applikationen, er login\_menu.

**Login menu**

Et billede, der indeholder tekst, skærmbillede

Automatisk genereret beskrivelse

Login menuen består af en masse widgets, som vi har brugt til at designe den. To af de widgets er entry til indtastning af brugernavn og adgangskode.



Når brugeren ønsker at logge ind, og trykker på loginknappen, kaldes funktionen log. For at log funktionen kan virke, tager vi brugernavnet og adgangskoden med over som parametre.

**Login funktion**

Et billede, der indeholder tekst, skærmbillede, software, Font/skrifttype

Automatisk genereret beskrivelse

Funktionen her bruges til at tjekke om de indtastet oplysninger fra login menuen er i vores database. Grunden til at den er oprettet som funktion var at vi oplevede nogle problemer med argumenter da vi skulle sætte vores kode sammen. Vi kører kun denne funktion når der bliver logget ind, og ikke andre tidspunkter i programmet, og ville derfor have været bedst placeret som en metode i menu klassen.

Koden starter med at hente de indtaste oplysninger fra menu siden, og tilkobler sig til databasen.



Sql forsørgelsen starter med at hente alle kolonner fra tabellen bruger. Tabellen bruger består af brugernavn, adgangskode og type id. Koden her bruger kun brugernavn og adgangskoden. Den tager brugernavnet og adgangskoden og erstatter ”?”, så den finder, kun rækker hvor begge matcher den værdi.  
Grunden til at den også skulle hente type id, var for at se om brugeren var administrator, eller en standard bruger. Meningen var at hvis brugeren var administrator, skulle man have adgang til admin siden, og hvis brugeren var oprettet som standard, skulle den ikke blive vist. Vi oplevede problemer med dette, og måtte fjerne det under sammensætning af vores kode.

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype

Automatisk genereret beskrivelse

Hvis login lykkes, rydder den alle widgets i vinduet, for at gøre plads til de nye. Det gør ved hjælp af en for-lykke som går alle widgets igennem, og bruget destroy funktionen til at fjerne dem. Når alle widgets er blevet fjernet, starter den en ny instans af mainmenu.  
Hvis login fejler, viser den en pop-up besked med ”fejl i brugernavn eller adgangskode”. Efterfølgende fjerner den det indtastede brugernavn og adgangskode, så brugeren kan prøve igen.

**Mainmenu**

**Et billede, der indeholder skærmbillede, tekst, software, Multimediesoftware

Automatisk genereret beskrivelse**

Mainmenu består af 3 knapper, hvor alle bruges til at gå ind på følgende sider (frames), historik, admin og beregneren. Hver knap har sin egen funktion, som minder om hinanden.

**Navigation mellem frames.**

**Et billede, der indeholder tekst, skærmbillede, Font/skrifttype, linje/række

Automatisk genereret beskrivelse**

Vi gør brug af 4 funktioner som gør det samme. Her fjerner de widgets i vinduet og laver en instans af en ny klasse.  
En bedre løsning var kun at have en funktion, og hver gang man kaldte den, bruge et argument for hvilken klasse (frame) der skal bruges.

**Adminmenu**

**Et billede, der indeholder tekst, skærmbillede

Automatisk genereret beskrivelse**

Admin menuen giver brugeren mulighed for at oprette nye brugere. Her bruger vi entrys for at lave et brugernavn og adgangskode. Man skal også vælge om brugeren skal være admin, eller normal bruger. Valget har ikke nogen betydning for vores prototype, men vil have når vi videreudvikler på programmet. Efter der er indtastet brugernavn, adgangskode og valgt rettigheder, kan brugeren bruge opret knappen for at få brugeren oprettet. Når der bliver trykket på opret knappen, bruger den funktionen opret\_bruger.

**Opret bruger**

**Et billede, der indeholder tekst, skærmbillede, software

Automatisk genereret beskrivelse**

****

Den henter brugernavn, adgangskode og brugerens rolle (admin eller normal bruger). Ved oprettelse af brugeren, gør vi brug af værdierne 1 for admin og 2 for standard bruger. Grunden til det, er fordi i vores database, indikere værdien 1 admin og 2 standard bruger. Det første den tjekker er om brugernavnet allerede eksisterer i databasen. Hvis det gør, vises en fejlbesked, og processen stopper. Hvis brugerens navn ikke allerede findes, indsætter den værdierne ind i databasen, og giver en besked om at brugeren er blevet oprettet, og inputfelterne ryddes.

#### **Opsætning af database til håndtering af maskiner og omkostninger**

#### **Databaseopdeling og 3. normalform**

Databasen følger principperne for **3. normalform** (3NF) for at sikre dataintegritet og minimere redundans. Hver tabel repræsenterer en klart defineret enhed. For eksempel:

Et billede, der indeholder tekst, Font/skrifttype, skærmbillede, linje/række

Automatisk genereret beskrivelse

**CREATE TABLE Machine (**

* Denne linje starter oprettelsen af en ny tabel i databasen med navnet Machine.
* Tabellen skal bruges til at organisere og gemme data, der relaterer sig til maskiner.

**MachineID INTEGER PRIMARY KEY AUTOINCREMENT,**

* ***MachineID***: Det er navnet på en kolonne i tabellen, som bruges til at identificere hver enkelt maskine entydigt.
* ***INTEGER***: Denne kolonne skal indeholde heltal.
* ***PRIMARY* *KEY***: Marker denne kolonne som primærnøglen for tabellen. En primærnøgle betyder, at hver værdi i denne kolonne skal være unik, og den må ikke være tom.
* ***AUTOINCREMENT***: Når en ny række tilføjes til tabellen, bliver værdien i denne kolonne automatisk øget med 1, så hver række får en unik identifikator.

**MachineName TEXT NOT NULL,**

* ***MachineName***: Det er navnet på en kolonne, der skal gemme navnet på maskinen.
* ***TEXT***: Denne kolonne skal indeholde tekstdata (string).
* ***NOT NULL***: Dette angiver, at denne kolonne ikke må være tom. Hver række i tabellen skal have en værdi for maskinens navn.

**ProcessType TEXT NOT NULL**

* ***ProcessType***: Det er navnet på en kolonne, der beskriver hvilken type procesmaskinen kan udfører.
* ***TEXT***: Denne kolonne skal også indeholde tekstdata.
* ***NOT* *NULL***: Som for MachineName, skal denne kolonne have en værdi for hver række i tabellen.

Tabellen Machine vil altså:

* Opbevare oplysninger om maskiner, såsom deres unikke ID (MachineID), navn (MachineName), og hvilken type proces de udfører (ProcessType).
* Sikre dataintegritet ved hjælp af en primærnøgle og NOT NULL-begrænsninger.

For nogle af tabellerne har det ikke været tilstrækkeligt med én primærnøgle, da flere maskiner, for eksempel kan bruge flere forskellige materialer (mange-til-mange relation). Derfor anvender vi en kombination af to primærnøgler, som sammenkobles gennem en relationstabel. Et eksempel på dette er:

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype, nummer/tal

Automatisk genereret beskrivelse

***CREATE TABLE MachineMaterialCost (***

* Opretter en ny tabel med navnet MachineMaterialCost.
* Denne tabel fungerer som en **relationstabel**, der forbinder de to tabeller: Machine og Materials.

***MachineID INTEGER,***

* **MachineID**: En kolonne, der refererer til maskinens unikke ID fra tabellen Machine.
* **INTEGER**: Angiver, at værdien skal være et heltal.

***MaterialID INTEGER,***

* **MaterialID**: En kolonne, der refererer til materialets unikke ID fra tabellen Materials.
* **INTEGER**: Angiver, at værdien skal være et heltal.

***Cost REAL***

* **Cost**: En kolonne, der angiver prisen eller omkostningen for at bruge et specifikt materiale med en specifik maskine.
* **REAL**: Bruges til at gemme numeriske værdier med decimaler, som f.eks. en pris eller en kostpris.

***Unit TEXT,***

* **Unit**: En kolonne, der specificerer, i hvilken enhed omkostningerne beregnes (f.eks. "kg")
* **TEXT**: Denne kolonne gemmer tekst (string).

***PRIMARY KEY (MachineID, MaterialID),***

* Angiver en sammensat primærnøgle, der består af både MachineID og MaterialID.
* Dette sikrer, at hver kombination af maskine og materiale er unik i tabellen.
* Den sammensatte primærnøgle er nødvendig, fordi vi har en mange-til-mange-relation:
  + En maskine kan bruge mange forskellige materialer.
  + Et materiale kan bruges af mange forskellige maskiner.
* Tabellen opretter et unikt par for hver relation mellem en maskine og et materiale.

***FOREIGN KEY (MachineID) REFERENCES Machine (MachineID),***

* Opretter en fremmednøgle, der forbinder MachineID i denne tabel til MachineID i tabellen Machine.
* Dette betyder, at MachineID i MachineMaterialCost skal matche en eksisterende MachineID i Machine.

***FOREIGN KEY (MaterialID) REFERENCES Materials (MaterialID)***

* Opretter en fremmednøgle, der forbinder MaterialID i denne tabel til MaterialID i tabellen Materials.
* Dette betyder, at MaterialID i MachineMaterialCost skal matche en eksisterende MaterialID i Materials.

**Hvordan fungerer relationstabellen?**

1. **Mange-til-mange-relation**:
   * Tabelstrukturen løser problemet med mange-til-mange-relationer mellem Machine og Materials:
     + En maskine kan kræve flere forskellige materialer for at fungere.
     + Et materiale kan bruges af flere maskiner.
2. **Sammensat primærnøgle**:
   * Primærnøglen består af både MachineID og MaterialID. Dette sikrer, at der ikke er dubletter for en given kombination af maskine og materiale.
3. **Fremmednøgler**:
   * Fremmednøglerne sikrer, at dataene i relationstabellen altid er valide:
     + MachineID skal eksistere i Machine.
     + MaterialID skal eksistere i Materials.

#### **Popularisering af tabeller**

Her har vi valgt at popularisere tabellerne ved hjælp af **parametre** frem for direkte INSERT INTO-kommandoer, da denne tilgang giver flere fordele i forhold til sikkerhed og fleksibilitet. Dette betyder, at vi indsætter data ved at bruge placeholders (’?’) og derefter leverer værdierne som parametre. For eksempel:

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype

Automatisk genereret beskrivelse

Ved at bruge placeholders til popularisering af tabeller har vi opnået en mere sikker, fleksibel og effektiv måde at indsætte data på. Dette gør det muligt at tilpasse databasen til ændringer uden behov for omfattende kodeændringer. Samtidig er metoden skalerbar, hvor dataindsættelser sker, hvilket gør den til et fremtidssikret valg.

Til at beregne omkostningerne pr. cm3 har vi været nødsaget til at omregne nogle af de enheder og priser, som var givet fra Excel arket. Dette er gjort i selve databasen som følger:

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype, nummer/tal

Automatisk genereret beskrivelse

**INSERT INTO MaterialCostPerCM3 (MachineID, MaterialID, CostPerCM3)**

* Indsætter data i tabellen MaterialCostPerCM3.
* Der specificeres tre kolonner:
  + MachineID: Refererer til den maskine, der bruger materialet.
  + MaterialID: Refererer til det specifikke materiale.
  + CostPerCM3: Beregnet omkostning pr. kubikcentimeter af materialet.

**SELECT**

* Vælger data, der skal indsættes i MaterialCostPerCM3. Hver valgt række svarer til en række, der bliver indsat.

**mmc.MachineID,**

* Vælger maskinens ID fra MachineMaterialCost (mmc) og bruger det til MachineID i den nye tabel.

**mmc.MaterialID,**

* Vælger materialets ID fra MachineMaterialCost og bruger det til MaterialID i den nye tabel.

**CASE**

* Starter en betinget logik for at beregne CostPerCM3, afhængigt af enhedstypen (mmc.Unit).

**WHEN mmc.Unit = '$/kg' THEN mmc.Cost / (materials.Density \* 1000)**

* Hvis enheden er dollars pr. kilo ($/kg), beregnes kostprisen pr. kubikcentimeter ved at:
  + Dividere prisen (mmc.Cost) med densiteten pr. kubikcentimeter:
    - materials.Density **\*** 1000: Tæthed (i g/cm³) omregnet til kg/m³.

**WHEN mmc.Unit = 'unit' THEN NULL**

* Hvis enheden er unit (stykspris), indsættes NULL for CostPerCM3, fordi beregningen muligvis ikke er relevant eller skal håndteres særskilt.

**WHEN mmc.Unit = '$/L' THEN mmc.Cost / 1000**

* Hvis enheden er dollars pr. liter ($/L), beregnes kostprisen pr. kubikcentimeter som:
  + mmc.Cost / 1000, fordi der er 1000 cm³ i 1 liter.

**WHEN mmc.Unit = '$/10kg' THEN (mmc.Cost / 10) / (materials.Density \* 1000)**

* Hvis enheden er dollars pr. 10 kg ($/10kg), beregnes kostprisen pr. kubikcentimeter som:
  + Først omregnes prisen til 1 kg: mmc.Cost / 10.
  + Derefter divideres det med materials.Density \* 1000.

**ELSE NULL**

* Hvis ingen af de nævnte enhedstyper matcher, indsættes NULL for CostPerCM3.

**END AS CostPerCM3**

* Afslutter CASE-udtrykket og navngiver det beregnede resultat som CostPerCM3.

**FROM MachineMaterialCost AS mmc**

* Angiver tabellen MachineMaterialCost (med alias mmc) som datakilde.

**JOIN Materials ON mmc.MaterialID = Materials.MaterialID**

* Udfører en JOIN mellem tabellerne MachineMaterialCost og Materials:
  + Matcher rækker, hvor MaterialID i begge tabeller er ens.
  + Dette er nødvendigt for at hente materialets tæthed (Density), der bruges i beregningerne.

Efter kørsel af dette stykke kode, vil tabellen MaterialCostPerCM3 indeholde:

* Maskinens ID (MachineID).
* Materialets ID (MaterialID).
* Kostprisen pr. kubikcentimeter (CostPerCM3), beregnet baseret på enheden og materialet densitet

Dette gør det muligt at udføre beregninger af materialeomkostninger på tværs af maskiner og materialer.

Et billede, der indeholder tekst, nummer/tal, skærmbillede, Font/skrifttype

Automatisk genereret beskrivelse

**\*Indsæt login database\***

**Beregner**

For at kunne benytte informationerne fra databasen i vores beregner, skal vi lave en forbindelse mellem vores database og programmet. Dette er gjort på følgende måde:

Et billede, der indeholder tekst, Font/skrifttype, skærmbillede, linje/række

Automatisk genereret beskrivelse

**class Database:**

* Definerer en ny klasse ved navn Database. Klassen indeholder metoder til database-forespørgsler.

**def \_\_init\_\_(self, db\_name):**

* Initialiserer en ny instans af Database.
* db\_name er en parameter, der angiver navnet eller stien til databasefilen.

**self.db\_name = db\_name**

* Gemmer databasefilnavnet i instansvariablen self.db\_name for at gøre det tilgængeligt i andre metoder.

Et billede, der indeholder tekst, Font/skrifttype, linje/række, skærmbillede

Automatisk genereret beskrivelse

**def fetch\_dropdown\_data(self):**

* Definerer en metode uden parametre (udover self), der henter data til en dropdown-menu.

**conn = sqlite3.connect(self.db\_name)**

* Opretter en forbindelse til databasen ved hjælp af sqlite3 og den gemte db\_name.
* conn bruges til at kommunikere med databasen.

**cursor = conn.cursor()**

* Cursoren bruges til at udføre SQL-forespørgsler.

**machines = [row[0] for row in cursor.execute("SELECT MachineName FROM Machine")]**

* Udfører en SQL-forespørgsel, der henter MachineName fra tabellen Machine.
* cursor.execute("SELECT MachineName FROM Machine") returnerer maskinnavnet fra tabellen Machine.
* [row[0] for row in ...] opbygger en liste over maskinnavne og henter det første element i hver række, dvs. selve maskinnavnet.

**conn.close()**

* Lukker forbindelsen til databasen.

**return machines**

* Returnerer listen over maskinnavne, der skal bruges i dropdown-menuen i GUI'en.

**Opdatering af drop-down menu baseret på Machine**

Hvad gør koden??!?!?!?! / formål

Formålet er at hente materialer, der er kompatible med den angivne maskine, samt maskinens processtype. Samt opdatere dropdownmenuen i guien

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype, nummer/tal

Automatisk genereret beskrivelse

**def fetch\_compatible\_data(self, machine\_name):**

* Definerer en metode, der tager machine\_name som parameter.

**materials = [row[0] for row in cursor.execute(''' ... ''', (machine\_name,))]**

* Udfører en SQL-forespørgsel, der henter navne på materialer kompatible med machine\_name.
* **SELECT Materials.MaterialName**: Vælger kolonnen MaterialName fra Materials-tabellen.
* **FROM MaterialCostPerCM3**: Starter forespørgslen fra MaterialCostPerCM3-tabellen.
* **JOIN Machine ON Machine.MachineID = MaterialCostPerCM3.MachineID**: Forbinder Machine og MaterialCostPerCM3 via MachineID.
* **JOIN Materials ON Materials.MaterialID = MaterialCostPerCM3.MaterialID**: Forbinder Materials og MaterialCostPerCM3 via MaterialID.
* **WHERE Machine.MachineName = ?**: Begrænser resultaterne til den maskine med navnet machine\_name.

**\*Indsæt en overgang\***

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype, nummer/tal

Automatisk genereret beskrivelse

**def fetch\_cost\_details(self, query, params):**

* Definerer en metode til at udføre en SQL-forespørgsel, som skal bruges senere.
* **query**: En SQL-forespørgsel som en streng med pladsholdere (?).
* **params**: En tuple af parametre, der skal indsættes i forespørgslen.

**cursor.execute(query, params)**

* Udfører den givne SQL-forespørgsel med de angivne parametre.
* **Parameterisering** sikrer fleksibilitet og beskyttelse mod SQL-injektion.

**result = cursor.fetchall()**

* Henter alle rækker fra resultatsættet og gemmer dem i result.

**conn.close()**

* Lukker forbindelsen til databasen.

**return result**

* Returnerer resultatsættet som en liste af tuples.

### **Databasen i beregningerne**

Vores program benytter tabellerne i databasen; Disse oplysninger hentes via SQL-queries og bruges direkte i beregningerne. Her er hvordan det fungerer overordnet set:

1. **Henter grundlæggende data**
   * Data som materialets pris pr. cm³, maskinens byggehastighed, og faste omkostninger hentes fra databasen.
   * Disse data bruges som input til beregninger, der kombinerer flere variabler for at finde de samlede omkostninger.
2. **Relationstabeller**
   * Tabellen MaterialCostPerCM3 fungerer som en **relationstabel**, der binder maskiner (Machine) og materialer (Materials) sammen og tilføjer ekstra data (pris pr. cm³).
   * Tabellen BuildRate indeholder byggehastigheder, der er unikke for hver maskine.
   * Tabellen FixedCosts tilføjer opsætnings- og fjernelsesomkostninger.
3. **Kombinering af data**
   * **JOIN** bruges til at kombinere relevante data fra flere tabeller. Dette sikrer, at beregningerne tager højde for flere faktorer.

Denne overordnet arkitektur tillader at programmet er fleksibelt, skalerbart og dynamisk.

**Beregner**

Et billede, der indeholder tekst, Font/skrifttype, linje/række, nummer/tal

Automatisk genereret beskrivelseHer kan vi se at beregneren er opsat som en class der tager data fra databasen, herefter definere en metode til at beregne omkostningerne.

Et billede, der indeholder tekst, Font/skrifttype, nummer/tal, linje/række

Automatisk genereret beskrivelse

Dataene bliver hentet ved hjælpe af SQL-forespørgsler (queries), som vist ovenfor. For at lave selve beregningerne har det været nødvendigt at opsætte nogle regnestykker. Følgende er et eksempel på dette:



**cost\_per\_cm3 = self.database.fetch\_cost\_details(queries["cost\_per\_cm3"], (machine\_name, material\_name))**

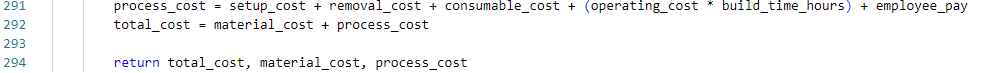
* Udfører SQL-forespørgslen queries["cost\_per\_cm3"] for at hente prisen pr. kubikcentimeter (CostPerCM3) for det angivne maskin- og materialenavn.
* Parametre: (machine\_name, material\_name), som bliver udfyldt længere oppe.

**cost\_per\_cm3 = cost\_per\_cm3[0][0] if cost\_per\_cm3 else 0**

* Hvis cost\_per\_cm3 ikke er tom, hentes værdien fra den første tuple (cost\_per\_cm3[0][0]).
* Hvis forespørgslen ikke returnerer noget, sættes cost\_per\_cm3 til 0.

**material\_cost = cost\_per\_cm3 \* volume\_cm3**

* Beregner den samlede materialeomkostning ved at gange prisen pr. kubikcentimeter med det totale volumen.



Til sidst bliver alle disse mellemregninger samlet for at kunne beregne *procesomkostninger, materialeomkostninger og totaleomkostninger.*

**Historik**

Denne Python-kode definerer en klasse Historik, som repræsenterer et brugergrænsefladevindue (UI) for at vise, redigere og håndtere historiske data fra en SQLite-database. Klassen er bygget med customtkinter (forkortet som c) og ttk til UI-komponenter

Et billede, der indeholder tekst, software, Multimediesoftware, skærmbillede

Automatisk genereret beskrivelseHistorik er en ny side (en CTkFrame), der repræsenterer historiksiden. parent refererer til MainApp, og db\_name angiver databasenavnet.

**super().\_\_init\_\_(parent):** initialiserer basisklassen.

**self.app** og **self.db\_name** gemmer henholdsvis forælderen og databasefilens navn for senere brug.

**tilbage knap:** Indlæser en pil-ikonbillede og opretter en knap, som kalder funktionen tilbage for at vende tilbage til hovedmenuen. Knapplaceret i grid-layoutets øverste venstre hjørne (row=0, column=0).

**Label**: Opretter en overskrift med teksten "Historik".

Et billede, der indeholder tekst, software, Font/skrifttype, skærmbillede

Automatisk genereret beskrivelse

Et billede, der indeholder tekst, skærmbillede, software

Automatisk genereret beskrivelse

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype, Multimediesoftware

Automatisk genereret beskrivelse

Treeview: Indeholder en tabel med kolonner som "PrintNumber", "PrintDate" osv. Det bruges til at vise historiske data fra databasen.

Et billede, der indeholder tekst, skærmbillede, software

Automatisk genereret beskrivelse

Lade historik fra databasen: Opretter en forbindelse til SQLite-databasen og initialiserer en cursor for at udføre SQL-forespørgsler.

Kører en SQL-forespørgsel for at hente data fra tabellen PrintHistoryData.

Et billede, der indeholder tekst, skærmbillede, software, Multimediesoftware

Automatisk genereret beskrivelse

Et billede, der indeholder skærmbillede, tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst, skærmbillede

Automatisk genereret beskrivelse

Et billede, der indeholder skærmbillede, tekst

Automatisk genereret beskrivelse

**CustomTkinter (c)**

CustomTkinter er et moderne og fleksibelt GUI-bibliotek, der bygger oven på Tkinter, men tilbyder bedre design og flere tilpasningsmuligheder.

**Hvorfor bruge det?**

Det giver dig moderne widgets (som knapper, frames osv.), der har et bedre udseende end standard Tkinter.

Støtter temaer og farvetilpasning uden komplekse tilpasninger.

Gør det nemt at skabe en professionel brugergrænseflade.

**Eksempler på widgets brugt i din kode:**

**c.CTk**: Hovedvinduet i applikationen (MainApp).

**c.CTkFrame**: En container/widget, der fungerer som ramme for andre widgets (main\_frame og knaprammer).

**c.CTkButton**: Moderne knapper med hover-effekter og mulighed for at tilføje ikoner.

**c.CTkLabel**: Tekstetiketter med mulighed for tilpasning af skrifttype og stil.

**Ekstra funktionalitet fra CustomTkinter:**

**Tematilpasning:** Du kan ændre applikationens udseende ved at vælge farvetemaer som "dark-mode" eller "light-mode".

**Responsivt layout:** Widgets som CTkFrame justerer sig automatisk for forskellige vinduesstørrelser.

**SQLite3**

sqlite3 er et indbygget Python-modul, der giver dig mulighed for at interagere med SQLite-databaser. Det er letvægts og kræver ingen ekstern opsætning.

**-**Perfekt til små eller mellemstore applikationer, hvor du har brug for at gemme struktureret data (som historik eller brugerinput).

-Ingen behov for en separat database-server.

-Enkel integration med Python.

**Eksempler på brug i din kode:**

conn = sqlite3.connect(self.db\_name): Åbner en forbindelse til databasen manufacturing.db.

**Cursor til at udføre forespørgsler:**

cursor.execute('SELECT ...'): Udfører en SQL-forespørgsel for at hente data fra PrintHistoryData.

**Tkinter (indbygget i Python)**

Tkinter er det standard Python-bibliotek til GUI-programmering. Selvom du bruger CustomTkinter, trækker det på Tkinter-funktionalitet i baggrunden.

**Brugte funktioner i din kode:**

**ttk.Treeview**: En avanceret tabel-widget, der bruges til at vise data i rækker og kolonner.

I din kode bruges Treeview til at vise historiske udskrifter med kolonner som PrintNumber, MachineName osv.

**simpledialog**: Bruges til at få input fra brugeren via en pop-up:

new\_volume = simpledialog.askfloat("Edit Volume", "Enter new Volume:", initialvalue=volume)

**4. PIL (Pillow)**

Pillow er et populært Python-bibliotek til billedbehandling. Det er en moderniseret version af det tidligere PIL.

tilbage\_logo = c.CTkImage(Image.open('c:/Users/.../Back-arrow.png'), size=(20, 20))

**Image.open**: Indlæser et billede fra din harddisk.

**c.CTkImage**: Konverterer billedet til et CustomTkinter-kompatibelt format, så det kan bruges som ikon på en knap.

Det gør det nemt at manipulere billeder (ændre størrelse, rotere, konvertere formater osv.). Selvfølgelig! Her er en forklaring på de moduler og biblioteker, du har brugt i din kode:

**CustomTkinter (c)**

CustomTkinter er et moderne og fleksibelt GUI-bibliotek, der bygger oven på Tkinter, men tilbyder bedre design og flere tilpasningsmuligheder.

Det giver dig moderne widgets (som knapper, frames osv.), der har et bedre udseende end standard Tkinter.

Støtter temaer og farvetilpasning uden komplekse tilpasninger.

Gør det nemt at skabe en professionel brugergrænseflade.

**Eksempler på widgets brugt i min kode:**

**c.CTk**: Hovedvinduet i applikationen (MainApp).

**c.CTkFrame**: En container/widget, der fungerer som ramme for andre widgets (main\_frame og knaprammer).

**c.CTkButton**: Moderne knapper med hover-effekter og mulighed for at tilføje ikoner.

**c.CTkLabel**: Tekstetiketter med mulighed for tilpasning af skrifttype og stil.

**Ekstra funktionalitet fra CustomTkinter:**

**Tematilpasning:** Du kan ændre applikationens udseende ved at vælge farvetemaer som "dark-mode" eller "light-mode".

**Responsivt layout:** Widgets som CTkFrame justerer sig automatisk for forskellige vinduesstørrelser.

**SQLite3**

sqlite3 er et indbygget Python-modul, der giver dig mulighed for at interagere med SQLite-databaser. Det er letvægts og kræver ingen ekstern opsætning.

-Ingen behov for en separat database-server.

-Enkel integration med Python.

**Eksempler på brug kode:**

**Opret forbindelse til databasen:**

conn = sqlite3.connect(self.db\_name)

Åbner en forbindelse til databasen manufacturing.db.

**Cursor til at udføre forespørgsler:**

cursor.execute('SELECT ...')

Udfører en SQL-forespørgsel for at hente data fra PrintHistoryData.

**Opdater databasen:**

cursor.execute('UPDATE PrintHistoryData SET Volume = ? WHERE PrintNumber = ?', (new\_volume, print\_number))

conn.commit():Ændrer en række i databasen og gemmer ændringerne med commit().

**Tkinter (indbygget i Python)**

Tkinter er det standard Python-bibliotek til GUI-programmering. Selvom du bruger CustomTkinter, trækker det på Tkinter-funktionalitet i baggrunden.

**Brugte funktioner i din kode:**

**ttk.Treeview**: En avanceret tabel-widget, der bruges til at vise data i rækker og kolonner.

I din kode bruges Treeview til at vise historiske udskrifter med kolonner som PrintNumber, MachineName osv.

**messagebox**: Bruges til at vise beskeder til brugeren, f.eks. ved reprint-funktionen:

messagebox.showinfo("Reprint", "Reprint started successfully!")

**simpledialog**: Bruges til at få input fra brugeren via en pop-up:

new\_volume = simpledialog.askfloat("Edit Volume", "Enter new Volume:", initialvalue=volume)

**PIL (Pillow)**

Pillow er et populært Python-bibliotek til billedbehandling. Det er en moderniseret version af det tidligere PIL.

**Eksempel i din kode:**

tilbage\_logo = c.CTkImage(Image.open('c:/Users/.../Back-arrow.png'), size=(20, 20))

**Image.open**: Indlæser et billede fra din harddisk.

**c.CTkImage**: Konverterer billedet til et CustomTkinter-kompatibelt format, så det kan bruges som ikon på en knap.

Det gør det nemt at manipulere billeder (ændre størrelse, rotere, konvertere formater osv.).

Perfekt til at tilføje ikoner og grafik i GUI-applikationer.

**Python standardbibliotek**

**os eller pathlib (ikke direkte i koden, men relevant hvis du arbejder med filer):** Hjælper med filstier og at finde filer som Back-arrow.png.

**sys (implicit i Tkinter):** Bruges af Tkinter til at håndtere GUI-events og systeminteraktion.

Perfekt til at tilføje ikoner og grafik i GUI-applikationer.

Ved at kombinere CustomTkinter, sqlite3, og Pillow skaber du en:

**Attraktiv GUI** (med moderne widgets fra CustomTkinter).

**Effektiv datahåndtering** (med sqlite3).

**Visuelt tiltalende applikation** (med grafiske elementer fra Pillow).

**Sådan sætter du det hele sammen:**

**Definer funktioner for at hente og ændre data** – du bruger SQL til at hente og opdatere data i din database (sqlite3).

**Brug et GUI-framework** som CustomTkinter til at opbygge et vindue og widgets, som brugeren kan interagere med (f.eks. knapper, tabeller).

**Link data med brugerinteraktioner**

når brugeren f.eks. klikker på en knap eller vælger en række i tabellen, kører du relevante SQL-forespørgsler for at opdatere databasen og vise de nyeste data. Når du kombinerer Python, SQLite og en GUI (f.eks. CustomTkinter), får du et kraftfuldt program, der både kan håndtere databasetransaktioner og give brugeren en interaktiv og brugervenlig grænseflade.

**Diskussion**

**Login-siden**

Login-siden viser et minimalistisk design med en stærk kontrast mellem den mørke baggrund og de lyse inputfelter og knappen. Tekst og knapper er tydelige, og layoutet er centreret, hvilket skaber en simpel og overskuelig struktur. Ved at placere felterne for brugernavn og adgangskode tæt sammen inden for en visuel kasse understøttes nærhedsprincippet (Gestalt). Samtidig justerer centreret tekst og elementer brugen af alignment, som giver en ren og organiseret fornemmelse.

Selvom designet er simpelt og overskueligt, mangler der visuelle feedbackelementer, som hover-effekter eller farveskift på knappen. Logoet nederst er visuelt adskilt fra login-processen og kan virke malplaceret. Tilføjelse af farvede fejlmeddelelser og mere engagerende elementer kunne forbedre både brugervenlighed og æstetik. Derudover virker den store negative plads lidt for tom og kan optimeres ved at reducere pladsen mellem sektionerne.

**Menu-siden**

Denne side har tre knapper med funktionerne "Historik," "Admin," og "Beregn," arrangeret horisontalt og centralt på skærmen. Kontrasten mellem de blå knapper og den mørke baggrund er effektiv til at fremhæve interaktive elementer. Gestalt-princippet om ensartethed er opfyldt ved brug af samme størrelse, farve og form på knapperne. Der er også en god justering og balance i designet, som gør siden nem at overskue.

Trods enkelheden fremstår designet lidt for simpelt og kan virke uinspirerende. Den store plads omkring knapperne skaber en visuel adskillelse, men denne afstand kan føles overflødig. Knapperne kunne placeres tættere sammen eller grupperes i en boks for at signalere deres sammenhæng. Tilføjelse af ikoner eller korte beskrivelser ved siden af teksten kunne også forbedre forståelsen og gøre grænsefladen mere interessant.

**Beregningssiden**

Beregningssiden er funktionelt designet med en klar opdeling mellem input-området, grafen og resultatsektionen. Det grå område indkapsler input-elementerne og skaber en naturlig gruppering (Gestalt: nærhed). Dropdown-menuerne, inputfelterne og knappen er konsekvent designet og godt justeret, hvilket giver struktur og ensartethed. Grafens layout er let at forstå, og resultaterne er tydeligt angivet i tabelform.

Grafens hvide baggrund virker lidt malplaceret i forhold til det mørke tema og kunne tilpasses for at harmonisere bedre med resten af designet. Resultattabellen kunne flyttes tættere på grafen for at styrke forbindelsen mellem dem. Input-området kunne drage fordel af visuelle feedbackelementer som hover-effekter eller skygger for at fremhæve brugerens interaktion. Endelig kunne små ikoner eller hjælpeværktøjer ved dropdown-menuerne forbedre forståelsen af inputmulighederne.

**Konklusion på CRAP og Gestalt**

Der er tydelig kontrast, konsistens i designet, god justering og klar nærhed mellem relaterede elementer. De funktionelle områder er nemme at navigere, hvilket gør siderne brugervenlige.

Siderne kunne forbedres ved at optimere deres brug af pladsen, skabe bedre visuel harmoni og tilbyde flere guidende elementer. Login- og menu-siderne har for meget tom plads, hvilket kan give et indtryk af ufuldstændighed og mangel på designmæssige overvejelser. På beregningssiden fungerer funktionaliteten fint, men grafens visuelle stil og layoutet af de tekstbaserede resultater passer ikke helt til det overordnede tema, hvilket skaber et underligt udtryk. Derudover kunne tilføjelse af forklarende elementer, såsom ikoner eller korte beskrivelser, hjælpe brugerne med at forstå funktionerne bedre, især på menu- og beregningssiden.