VizCode

Et web-baseret, visuelt udviklingsmiljø

Specialeafhandling

Gruppe 8, DAD10

Aalborg University
Cassiopeia - House of Computer Science
Selma Lagerløfs Vej 300, Aalborg 9220

Setup made with LaTeXservice through Overleaf; https://www.overleaf.com.

AALBORG UNIVERSITY
STUDENT REPORT

Cassiopeia - House of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg

Titel:
VizCode

Tema:
Et web-baseret, visuelt udviklingsmiljø

Projektperiode:
Forårssemester 2024

Projektgruppe:
Gruppe 8, DAD10

Forfattere:
Hong Duc Cai
Jesper Thorup Nielsen
Magnus Barslund Pedersen

Vejleder:
Ivan Aaen

Kopier: 1

Sideantal: 99

Afleveringsdato:
31. Maj, 2024

Abstract:

This project explores whether combining graphical and textual representations in an introductory course like CS50 supports the learning of computational thinking and programming concepts. We developed VizCode, a web-based visual integrated development environment using Blockly and C. We employed Sarasvathy's Effectuation strategy and Aaen's Essence: Problem Solution Model Generation to navigate a complex environment. Initial user research, including questionnaires and interviews with CS50 students from Aalborg University, identified challenges related to programming complexity and frustration. Based on Dual Coding Theory and Papert's Constructionism, we designed Visualization-strategies and implemented Two-way synchronization, allowing users to manipulate code graphically and textually with immediate feedback. We evaluated our design using Flowgorithm and Cake-core as MVPs and conducted usability tests and focus group interviews. Findings suggest VizCode alleviates some frustrations by enabling continuous debugging through Blockly. However, further research is needed to confirm its impact on learning computational thinking and programming concepts. Early indicators are promising, but further investigation is necessary to validate these findings.

Forfattere:

Magnus Barslund Pedersen

MAGNUS BARSLUND PEDERSEN

mbpe22@student.aau.dk

Hong Duc Cai

HONG DUC CAI

hcai22@student.aau.dk

Jesper Thorup Nielsen

JESPER THORUP NIELSEN

jtni19@student.aau.dk

Resumé

In this project, we seek to explore the problem statement: "Can a combination of graphical and textual representations, in an introductory course such as CS50, support the learning of Computational thinking and Programming concepts?"We have developed a web-based, visual integrated development environment where the user is presented with a combination of graphical and textual representations. For this, we use the visual programming language *Blockly* and the text-based programming language C, which is used in a large portion of CS50 exercises. Our product has been named VizCode.

In this project, we attempt to employ Sarasvathy's Effectuation strategy in combination with Ivan Aaen's Essence: Problem Solution Model Generation to navigate a complex and unstable environment. Our project begins with an examination of existing resources and how they can be leveraged into creating a design with comparative advantages. Throughout the project, we change course a number of times based on what we discover and learn.

We conduct user-reasearch in the form of a questionnaire of 21 students who recently attended the CS50 course as part of an introductory course to programming at Aalborg University. Likewise, 6 of these students participated in follow up interviews. On behalf of these findings, we identified a number of challenges we argue pose a problem for students attending CS50. These challenges are related to the complexity and frustration experienced as a new student in programming.

To solve some of these problems, we formulate a design based on Dual Coding Theory by Allan Paivio and Paperts Constructionism. We develop the concepts of Visualization-strategies and Two-way synchronization. Visualization-strategies are graphical representations of code in the form of visual programming languages (VPL). We establish a two-way synchronization of textual and graphical representations, so that the user can manipulate in either one and receive immediate feedback. We also propose a number of features related to visual programming and pair programming, that ultimately have not been implemented.

Our design has been evaluated using two existing tools, Flowgorithm and Cake-core, that offer similar functionality, as MVPs by Eric Ries. We have implemented both features and tested them in a usability test, as well as a focus group interview. We analyze the findings using a deductive thematic analysis.

We discuss the findings in relation to our problem statement, where we argue that VizCode appears to solve some of the challenges we defined in our process. Participants explain that they engage in a continuous, less frustrating debugging process since they can use Blockly to mirror their text-based code. However, we argue that further research is required to further establish wheter VizCode introduces a positive change in terms of supporting the learning of computational thinking and programming concepts.

Forord

Denne specialeafhandling er produceret i perioden februar 2024 til juni 2024 på uddannelsen Digitalisering og Applikationsudvikling ved Aalborg Universitet. Projektet udspringer af en fascination af koncepter som LEGO Mindstorms, LittleBits og andre produkter, hvor leg og programmering kombineres til børn og unge. Digitalisering og Applikationsudvikling er en IT-uddannelse for studerende med andre baggrunde end IT. Formålet er at skabe et krydsfelt mellem IT og den studerendes respektive baggrund – et koncept, vi i dette projekt søger at teste. I projektet har vi tilegnet os erfaringer i at anvende vores personlige baggrunde som en *feature* i vores design. Dette har været en ny og spændende udfordring, der har været lærerig for vores videre virke.

Ulig vores tidligere projekter har vi i denne afhandling forsøgt at opnå en løsning, vi kan arbejde videre med efter uddannelsens afslutning. Vi er lykkedes med at skabe et koncept, vi selv finder tiltrækkende, og vi glæder os til at fortsætte arbejdet med VizCode i fremtiden.

Vi vil i den forbindelse gerne rette en tak til vores vejleder Ivan Aaen, der har været en kompetent rådgiver inden for dette formål. Ivan har ydet konstruktiv kritik og vejledning gennem ugentlige møder og sin egen metodologi, Essence: Problem Solution Model Generation. Ivans opfordring til at inkludere vores uddannelsesbaggrunde aktivt, har formet hvordan vi kommer til at tænke på vores rolle som designere og softwareudviklere fremover.

Vi vil ligeledes rette en tak til Ashna Mahmood, der opfordrede os til at undersøge krydsfeltet i mellem læring og teknologi. Uden hende var vi aldrig endt med den kaotiske, læringsrige og udfordrende proces det har været at designe og implementere VizCode.

Til slut vil vi gerne rette en tak til de studerende fra både Aarhus og Aalborg Universitet, der op til eksamensfrister indvilgede i at teste VizCode og give deres ærlige feedback som følge deraf. Vi håber at kunne overlevere et fedt værktøj til dem inden for den nærmeste fremtid.

Indhold

Figurer

Tabeller

Introduktion

I 2006 præsenterede Jeanette M. Wing en vision, hvor alle discipliner, professioner og sektorer vil se sig ændret som følge af brugen af computationelle koncepter, metoder og værktøjer. Hun beskriver en række færdigheder og tankegange, der udspringer af Computer Science, og hvordan disse kan gavne alle mennesker at besidde (Wing 2006). Hun anvender begrebet *Computational Thinking* som en samling af computationelle metoder og modeller, der burde introduceres i almen uddannelse (Wing 2006). I 2010 definerer hun Computational Thinking, fremover forkortet som CT, på følgende måde: *"Computational Thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent"* (Wing 2010). Jeanette M. Wings vision blev grundlaget for en række uddannelsesmæssige initiativer verden over, inklusive politiske tiltag på flere forskellige uddannelsesniveauer (Caspersen et al. 2018, s. 10). I Danmark har det inspireret bl.a. faget *Informatik*, der startede som forsøgfag i 2011 og nu er obligatorisk eller semi-obligatorisk på flere danske ungdomsuddannelser (Caspersen et al. 2018). Ligeledes er *Teknologiforståelse* blevet indført som forsøgsfag i den danske folkeskole i 2018, der forlænges fra 2024 (Grundskolen 2024).

Vores første oplevelser med Computational Thinking og programmering var et fag på Aalborg Universitet, der fulgte Harvards "CS50: Introduction to Computer Science"parallelt med undervisningstimer på Aalborg Universitet. CS50 er et indledende kursus i datalogi tilbudt af Harvard University. Kurset er kendt for sin omfattende introduktion til de grundlæggende koncepter inden for datalogi samt udvikling af software og algoritmisk tænkning. Det tiltrækker studerende fra en bred vifte af discipliner og med forskellige erfaringsniveauer, både på Harvard og globalt via online platforme som edX (Harvard University 2024). På deres online platform har flere millioner studerende fulgt kurset igennem tiden, og det er anerkendt som et af top-kurserne (Harvard University 2024).

1.1 Visuelle og fysiske programmeringssprog

I tilegnelsen af computationelle kompetencer og programmering er en lang række læringsværktøjer blevet udviklet. Blandt de mest kendte er *visuelle programmeringssprog* som Scratch eller Blockly, der ved brug af block-based programming udnytter intuitive grafiske syntakser (Resnick et al. 2009). Disse bruges ligeledes i diverse forløb i Teknologiforståelse, hvor eleverne eksempelvis i 4. klasse kan programmere Micro:Bits, der bruger et letforståeligt blokbaseret sprog til at modtage beskeder eller signaler mellem to enheder. De yngste elever i indskolingen bruger papir-versionen af Scratch Jr. til at lave små algoritmer (Grundskolen 2024). *Tangible Programming* er programmeringssprog, der ikke

er repræsenteret som tekst eller billeder digitalt. Det er i stedet repræsenteret som fysiske objekter, med hvilke et computerprogram kan beskrives igennem flow-of-control-strukturer, kommandoer eller andre programmeringselementer (Horn et al. 2009). Disse bruges ligeledes ofte til indlæring af grundlæggende kompetencer i Computational Thinking og programmering og er særligt anvendelige til helt yngre børn (Horn et al. 2009). CS50 anvender også et visuelt programmeringssprog som en del af forløbet, hvor den første uge bruges i Scratch. Her er formålet, at de studerende tilegner sig basale færdigheder i Computational Thinking, for derefter at påbegynde tekstbaserede programmeringssprog (Harvard University 2024).

Paperts Konstruktionisme, en teori om læring, siger, at individer lærer bedst ved at manipulere fysiske objekter og skabe noget meningsfuldt for individet (Papert 1980). Papert er ligeledes fortaler for inklusionen af teknologi i læring og anser teknologi for værende en mulighed for at udforske, undersøge og afprøve idéer. Flere af både de visuelle og fysiske programmeringssprog bygger på Seymour Paperts konstruktionisme og koncept om "*Things to think with*". Netop *Scratch* har forsøgt at emulere eller genskabe Paperts principper om læring ved at designe både platformen og programmeringssproget ud fra principper indeholdt i Paperts konstruktionisme (Resnick et al. 2009).

1.2 Computational Thinking og programmering

Visionen om at skabe et værktøj til facilitering af Computational Thinking og programmering opstod på baggrund af personlige erfaringer og interesse. Vi er alle studerende ved kandidatuddannelsen Digitalisering og Applikationsudvikling på Aalborg Universitet med bachelorgrader i andre felter. Det er derfor også først de seneste år, at vi har stiftet bekendtskab med Computational Thinking og programmering. På baggrund af vores egne erfaringer i det henseende opstod en interesse for at designe en løsning, der gjorde programmering lettere tilgængelig for nye studerende. Hos os eksisterer også en iboende interesse for spil, leg og elektronik. Her var vi særligt interesserede i muligheden for at inkorporere fysiske elementer i programmering, i tråd med koncepter som LEGO Mindstorm eller Arduino Little:bits.

Dette projekt påbegyndes med henblik på at skabe et fysisk programmeringssprog, der kunne indgå som en del af den danske folkeskole på de yngre klassetrin, situeret i *Teknologiforståelse*. I gennem processen vil fokus skifte, i takt med at vi lærer mere om både problemfeltet og vores egen løsning. Det afsluttes med at foreslå en web-baseret platform, der anvender eksisterende visuelle værktøjer til at støtte nye studerende i CS50. I gennem projektet vil de forskellige *pivots* illustreres med værktøjer fra designmetodologien Essence: Problem-Solution Modeling Generation, der er en række af værktøjer til problemer karakteriseret ved kompleksitet og uvished (Aaen 2024). Vi vil igennem projektet navigere i forskellige kontekster ved at inddrage principper fra Sarasvathys Effectuation - en entreprenøriel tilgang, der lægger vægt på at bruge tilgængelige ressourcer og iterativ læring til at forme

og tilpasse forretningsmål frem for at følge en forudbestemt plan. Her vil vi ligeledes anvende værktøjer fra Essence til at diskutere og anskue omfanget af de problemer, vi søger at håndtere, såvel som det bidrag, vi ønsker at udvikle.

I kapitlet *Teori og metode* redegør vi for baggrunden for dette projekt. Her beskriver vi fagene Teknologiforståelse, Informatik og CS50. Vi vil redegøre for anvendelsen og oprindelsen af visuelle og fysiske programmeringssprog i facilitering af Computational Thinking. Dernæst beskriver vi Essence, Effectuation og abduktion, før disse sammenfattes som vores strategiske grundlag igennem processen. Til slut redegør vi for Seymour Paperts konstruktionisme og "*Things to think with*", da Papert er en central figur igennem hele projektet. I kapitlet *Related work* opsummeres vores litteraturstudie af *Tangible Programming* og Computational Thinking med henblik på at undersøge eksisterende løsninger. Her anvender vi et framework inspireret af Aaen 2024s Prospect Scenarios til at diskutere forskellige løsninger ud fra, hvorvidt de er baserede på et digitalt eller fysisk design. Dernæst forsøger vi at afgrænse os til et problemfelt i kapitlet *Problemfelt*. Her anvender vi Sarasvathys *Bird-in-hand*-princip til at identificere, hvilke ressourcer og midler vi har til rådighed som team. Vi anvender disse ressourcer til at foreslå en række *Leverage points*, i.e. nøgleelementer i vores projekt, der kan udgøre en komparativ fordel for vores bidrag. På baggrund af disse afgrænser vi problemfeltet til kurset CS50: Introduction to Computer Science og de udfordringer, vi oplevede under kurset. I kapitlet *Visuel programmering* beskriver vi vores designproces fra denne problemstilling og frem til vores første udkast til et bidrag. På baggrund af vores *leverage points* foreslår vi en række features på konceptniveau, vi mener kan bringe værdi til nye studerende. Disse er designet ud fra en række teorier om menneskelig læring, hvoraf Paperts *konstruktionisme* er omdrejningspunktet. Der foretages brugerundersøgelser i form af et spørgeskema besvaret af 21 tidligere CS50-studerende, samt 6 interviews af samme studerende. På baggrund af den indsamlede empiri identificeres en række *Manifestationer* som *Tekniske udfordringer*, *Syntaksfejl* og *Feedback og gennemsigtighed*, som vi søger at håndtere med vores bidrag. Dernæst anvendes Problem scenarios til at beskue alternativerne til problemet og foretage en række valg angående omfanget af, hvilke problemer vi ønsker at løse. I kapitlet *Parprogrammering* beskriver vi vores designproces, hvor vi arbejdede på at inkludere samarbejde som en feature i vores løsning. Her anvendes Problem-, Leverage- og Prospect scenarios fra Essence til at undersøge, hvordan samarbejde kan inkluderes i vores design.

I kapitlet *Design* opsummeres og præsenteres vores design. Her beskrives de principper, vi har designet og implementeret vores løsning ud fra. Først præsenteres de manifesteringer af problemet, der blev identificeret på baggrund af brugerundersøgelserne og vores egne erfaringer. Dernæst præsenteres vores bidrag til at håndtere disse manifesteringer. Dernæst præsenteres wireframes til vores samlede løsning, hvor hver feature præsenteres adskilt. I kapitlet *Implementering* opsummeres og præsenteres vores implementerede produkt. Her præsenteres først Herbert Simons *Near decomposability* og samt Objekt-Orienteret Analyse og Design, der har guidet implementeringen af vores produkt.

Dernæst gives et overblik over arkitekturen og det samlede, implementerede produkt samt de anvendte værktøjer i forbindelse med dette.

I kapitlet *Evaluering* fremlægger vi proceduren for vores Usability test og Fokusgruppe-interview. Vi analyserer data, med henblik på at diskutere dem i næste kapitel. I kapitlet *Diskussion* diskuterer vi fundende fra vores fokusgruppeundersøgelse i relation til vores grundlæggende antagelser om VizCode. Vi diskuterer fundenes relevans for problemformuleringen. Vi diskuterer konsekvenserne af de til- og fravalg vi foretog under processen. I kapitlet konklusion konkluderer vi, at VizCode på nuværende tidspunkt viser indikationer på at støtte i indlæringen af computational thinking og programmeringsbegreber, men at yderligere undersøgelser er nødvendige.

Teori og metode

I dette kapitel redegøres der for den anvendte teori og metodiske fremgangsmåde til at behandle problemet. Først redegøres der for Teknologiforståelse og Informatik, efterfulgt af CS50 og derefter de mest kendte visuelle programmeringssprog og deres brug. Herefter redegøres der for Computational Thinking og begrebets rolle i programmering. Dernæst redegøres der for kernebegreberne i Ivan Aaen's Essence: Problem-Solution modeling, som udgør den anvendte designproces i dette projekt. Herefter redegøres der for vores forberedelse og strategi, hvor vi beskriver Sarasvathys Effectuation, den abduktive tilgang sammen med Essence som et strategisk ståsted. Til sidst redegøres der for Eric Ries begreb om MVP, der anvendes i processen til at afprøve forskellige idéer.

2.1 Teknologiforståelse og Informatik

I 2011 blev 'Informatik' indført som forsøgsfag i danske ungdomsuddannelser, der i 2016 blev gjort til obligatorisk eller semi-obligatorisk på HHX, STX og HTX (Caspersen et al. 2018, s. 26). I 2018 med *Strategi for Danmarks Digitale Vækst* igangsattes et 3-årigt forsøgsfag med 'Teknologiforståelse' som fag i folkeskolens 1.-9. klasse (Regeringen 2018). Faget har til formål at udstyre eleverne med kompetencer inden for udvikling og forståelse af digitale artefakter (Caspersen et al. 2018, s. 37). Her beskrives fire kompetenceområder: Digital Myndiggørelse, Digital Design og Designprocesser, Computationel Tankegang og Teknologisk Handleevne (Emu). Som grundlag for forsøgsfaget beskrives i *Strategi for Danmarks Digitale Vækst* en forventning om, at fremtidig vækst og fremdrift i Danmark i høj grad skal bero på kommende generationers kompetencer inden for IT og teknologi (Grundskolen 2024, s. 7). Direktør for IT-Vest, Michael Caspersen, og medforfattere beskriver også motivationen i en rapport, der har til hensigt at give et overblik over status, potentiale og udfordringer for Informatik og Computationel Tankegang i det danske uddannelsessystem (Caspersen et al. 2018). Her beskrives blandt andet, at det ikke blot handler om at uddanne flere specialister, der kan udvikle digitale artefakter. Det drejer sig først og fremmest om at:

> *"... omsætte vores teknologiske fremskridt til bedre og mere meningsfulde liv i et mere sikkert, retfærdigt og demokratisk samfund"(Besenbacher og Caspersen, 2017).*

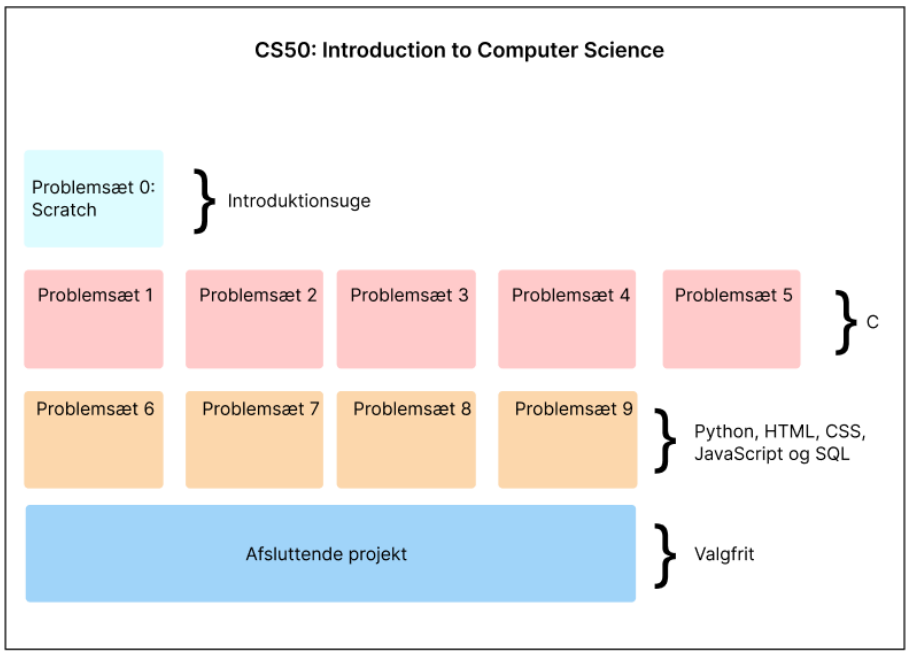
I forsøget har 22 skoler afprøvet Teknologiforståelse som selvstændigt fag, mens 24 skoler har afprøvet Teknologiforståelse som integreret fag (Regeringen februar 2024). I den danske folkeskole har forsøgsfaget også vist sig at være en overvejende succes. Den 8. februar 2024 blev forsøget berammet med yderligere to år (2024-2026) samt 160 mio. kroner (Regeringen februar 2024).

2.2 CS50

CS50, officielt kendt som "CS50: Introduction to Computer Science,"er et indledende kursus i datalogi tilbudt af Harvard University. Kurset er kendt for sin omfattende introduktion til de grundlæggende koncepter inden for datalogi samt udvikling af software og algoritmisk tænkning. Det tiltrækker studerende fra en bred vifte af discipliner og med forskellige erfaringsniveauer, både på Harvard og globalt via onlineplatforme som edX (Harvard University 2024).

2.2.1 Kursusopbygning

CS50 er struktureret til gradvist at introducere studerende til komplekse emner inden for datalogi gennem en serie forelæsninger og praktiske problemløsningssæt. Kurset starter med grundlæggende koncepter som algoritmer, kontrolstrukturer (løkker, betingelser) og datastrukturer. Efterhånden som semestret skrider frem, dækker kurset mere avancerede emner som hukommelsesallokering, datasikkerhed, databasesystemer og udvikling af webapplikationer (Harvard University 2024).



Figur 2.1: CS50 kursusstruktur

2.2.2 Problemsæt

Til hver undervisningsgang er der tildelt et problemsæt. Et problemsæt er en samling af praktiske programmeringsopgaver, der er designet til at give de studerende mulighed for at anvende og fordybe sig i de teorier og koncepter, der er blevet introduceret i forelæsningerne. Disse opgaver varierer i kompleksitet og omfang og tjener til at styrke de studerendes evne til kritisk tænkning og problemløsning.

Hvert problemsæt fokuserer på specifikke tekniske og teoretiske aspekter af datalogi. Disse opgaver kræver, at studerende skriver kode, tester og fejlfinder deres løsninger, hvilket udvikler deres programmerings- og CT-færdigheder (Harvard University 2024).

Mario

Et eksempel på et problemsæt er Mario fra det kendte (Harvard University 2023). Mario er det første problemsæt, der skal skrives i C, som de studerende støder på gennem kurset. I denne opgave skal de studerende skrive et program, der replikerer pyramidens stigende trin, som det ses i det klassiske videospil Super Mario. Opgaven kræver, at de studerende anvender løkker og betingelser for at skabe en pyramidestruktur, hvilket introducerer dem til grundlæggende kontrolstrukturer i programmering.

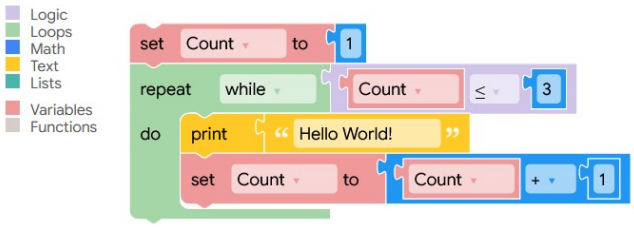
Figur 2.2: Mario-pyramiden (Super Mario Game 2024)

2.3 Visuelle programmeringssprog

Visuel programmering refererer til brugen af grafiske elementer, der tillader en bruger at skabe, modificere eller tilføje til softwareapplikationer (Kuhail et al. 2021). Visuelle programmeringssprog (VPL) repræsenterer forskellige konstruktioner, såsom løkker og kontrolstrukturer, ved brug af visuelle repræsentationer, der ofte også intuitivt viser syntaks og regler (Kuhail et al. 2021). I dette projekt benytter vi os af Kuhail et al. (2021)s taksonomiske opdeling af VPLs i fire kategorier: *Blok-, Diagram-, Ikon- og Formbaserede kodesprog*. Nedenstående afsnit beskriver to af disse fire kategorier, blokbaserede kodesprog og diagrambaserede kodesprog, som har bliver anvendt i løbet af dette projekt.

2.3.1 Blokbaserede sprog

Blokbaserede sprog gør det muligt for brugere at kombinere programmeringselementer, kendt som 'blokke', til forskellige strukturer. Disse strukturer repræsenterer en række foruddefinerede instruktioner (Kuhail et al. 2021). Hver blok repræsenterer en unik funktionalitet eller handling, såsom kontrolstrukturer, løkker, betingelser og variable. Brugere kan sammenkoble disse blokke for at skabe komplekse logiske sekvenser og algoritmer, hvilket gør det muligt at implementere fuldstændige programmeringslogikker uden traditionel kode (Resnick et al. 2009). I nedenstående figur 2.3 ses et eksempel på en repræsentation af et while-loop i *Blockly*.

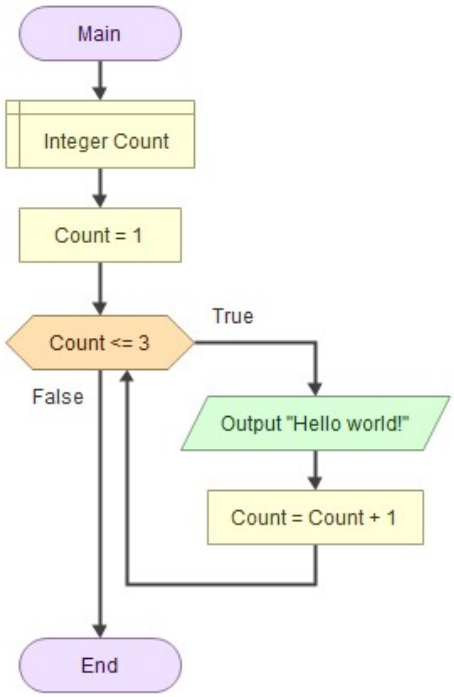


Figur 2.3: Eksempel på et while-loop i Blockly, der printer "Hello World!"tre gange (Google Developers 2024)

Blockly er et gratis web-bibliotek udviklet af Google, der indeholder en blokbaseret kodeeditor. Programmeringskoncepter som variabler, logik og løkker visualiseres som 'puslespilslignende' brikker, der kan sammensættes på forskellige måder. Denne editor kan bruges som den er, men tillader også omfattende muligheder for tilpasning til udvikleres specifikke behov. Udviklere kan tilpasse den genererede kode, ændre forbindelserne mellem blokkene, ændre layoutet af blokkene eller tilføje nye blokke. Blockly er tilpasset til flere læringskontekster og bruges til at fremme computational thinking og motivere studerende til at lære tekstbaserede kodesprog (Google Developers 2024).

Et eksempel på brugen af Blocklys web-bibliotek er det blokbaserede sprog *Scratch*, udviklet af *the Lifelong Kindergarten Group* ved MIT Media Lab (Resnick et al. 2009). I Scratch kan brugeren hurtigt skabe et fungerende program, ofte et mini-spil, og det findes til forskellige aldre i form af eksempelvis Scratch Jr. til børn (Resnick et al. 2009). Scratch bruges både i Harvards CS50 samt de yngre klassetrin i faget Teknologiforståelse (Grundskolen 2024).

Diagrambaserede sprog tillader brugeren at forbinde grafiske elementer (som f.eks. kasser) med pile, linjer eller løkker, der repræsenterer forskellige relationer (Kuhail et al. 2021). På den måde kan et program inspiceres på et højere abstraktionsniveau (Kuhail et al. 2021). Samme eksempel med et while-loop ses nedenfor i figur 2.4, repræsenteret i det diagrambaserede sprog *Flowgorithm*.



Figur 2.4: Eksempel på et while-loop i Flowgorithm, der printer "Hello world!"tre gange (Flowgorithm 2024)

Flowgorithm er et gratis diagrambaseret sprog rettet mod begyndere (Flowgorithm 2024). Brugeren skaber programmer ved brug af flowchart-baserede, grafiske repræsentationer. Flowgorithm er orienteret mod at tillade brugeren at skabe fungerende programmer uden at skulle forholde sig til syntaksfejl som ved tekstbaserede kodesprog. Ligeledes kan Flowgorithm konverteres direkte til en lang række udbredte tekstbaserede kodesprog som C#, Java, Python eller Swift (Flowgorithm 2024).

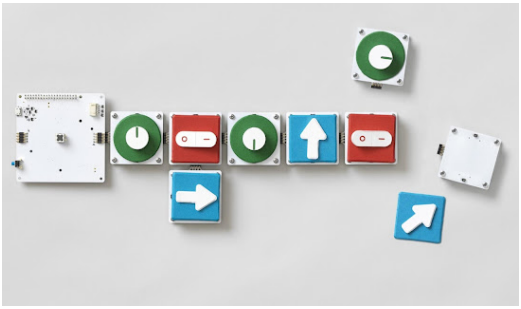
2.4 Fysiske programmeringssprog

Fysiske programmeringssprog, også kaldet 'Tangible programming languages' (TPL), involverer brug af fysiske objekter til at repræsentere og manipulere programmeringsinstruktioner. Disse objekter, som kan være blokke, kort eller andre enheder, tillader brugere, særligt børn, at sammensætte programmeringsinstruktioner ved fysisk at arrangere dem. Dette fysiske aspekt gør programmering mere intuitiv og tilgængelig, da det giver en direkte manipulation og repræsentation af data og kommandoer (Funk et al. 2021).

I kilden Funk et al. 2021 er der lavet et overblik over de mange forskellige TPL, der eksisterer, og forskellige taksonomiske niveauer på dem, hvilket vil blive redegjort og diskuteret yderligere i næste kapitel. Et af de sprog, der fremgår på modellen, er Project Bloks. Project Bloks er et TPL designet til at introducere børn til grundlæggende programmeringskoncepter ved hjælp af fysiske blokke. Sy-

stemet tillader brugere at arrangere blokkene for at danne programmeringssekvenser, hvilket undgår behovet for at forstå tekstbaseret programmeringssyntaks, og af samme grund er det velegnet til børn (Bloks 2016).

De fysiske blokke i *Project Bloks* er designet til at samarbejde både med digitale enheder og fysiske objekter. Dette gør det muligt for brugerne at se umiddelbare virkninger af deres programmeringsforsøg gennem interaktioner i den fysiske verden (Bloks 2016).



Figur 2.5: Project Bloks fra Google (Bloks 2016)

Et andet TPL, som fremgår på modellen af Funk et al. 2021, er Kubo. Kubo er et TPL designet til at indføre grundlæggende programmeringskoncepter for yngre børn ved hjælp af en lille robot og en række fysiske kodefliser kendt som "TagTiles". Hver TagTile repræsenterer en bestemt kommando, såsom 'start', 'stop', 'drej' og 'gå frem' (KUBO Robotics 2024).

Systemet fungerer ved, at brugeren arrangerer TagTiles i en sekvens på et gitter. Når sekvensen er etableret, navigerer Kubo-robotten langs denne sekvens og udfører de kodificerede kommandoer, som fliserne repræsenterer. Denne metode tillader brugerne at se øjeblikkelige fysiske resultater af deres programmeringssekvenser, hvilket giver en direkte forbindelse mellem handling og resultat (KUBO Robotics 2024).

Figur 2.6: Kubo robot og TagTiles (KUBO Robotics 2024)

Kubo er skabt med henblik på at være en indgangsplatform for programmering, hvor komplekse syntaks- og programmeringsstrukturer er forenklet til fysiske handlinger og reaktioner. Denne tilgang gør det muligt for børn at eksperimentere med og lære programmeringens grundlæggende principper uden behovet for forudgående teknisk viden (KUBO Robotics 2024).

2.5 Computational thinking

Jeannette M. Wing lancerede i 2006 begrebet Computational Thinking (CT) som en essentiel færdighed for alle, ikke kun programmører. Wing så CT som en tilgang til problemløsning, hvor datalogiske metoder anvendes til at formulere og løse problemer på en måde, der kan automatiseres og optimeres ved hjælp af computere. Hendes vision var at gøre CT til en integreret del af undervisningen på tværs af alle uddannelsesniveauer og discipliner, da hun anså computational thinking for at være lige

så fundamental som læsefærdigheder i en digital tidsalder (Wing 2006).

Michael Caspersen har bygget videre på Wings ideer og beskriver CT som en ny videnskabelig disciplin parallelt med klassiske områder som naturvidenskab, humaniora og samfundsvidenskab (Caspersen et al. 2018). Han fremhæver, at CT inkluderer evnen til systematisk at løse problemer ved hjælp af teknikker som dekomponering, mønstergenkendelse, algoritmisk tænkning og abstraktion. Denne tilgang er afgørende ikke kun i akademiske sammenhænge, men også i dagligdagens teknologiske udfordringer, som den almindelige dansker møder. Papert 1980 beskriver i sin bog 'Mindstorms', hvorfor det er vigtigt at uddanne børn i at bruge en computer, og at han ser computeren som en "teaching machine". Seymours filosofi genspejler sig i høj grad i konceptualiseringen af CT og CT's videnskabelige fundament. Computeren er ikke blot blevet en "teaching machine", men noget som hele vores samfund hviler på; blandt andet måden hvorpå data behandles, digitale processer og nu også AI, der får en større rolle i analyse- og beslutningsprocesser. Derfor mener Caspersen et al. 2018 også, at vi demokratiserer IT og teknologi ved at gøre det til en del af folkeskoleundervisningen. Ved at sikre demokratiseringen af det, vil flere være i stand til at tænke kritisk og deltage aktivt i forhold til teknologien, som stadig spiller en større rolle.

Læring af Computational Thinking er især værdifuld i konteksten af programmeringsundervisning, da det ikke blot forbedrer evnen til at skrive kode, men også forstærker den overordnede forståelse af, hvordan og hvorfor systemer fungerer. Dette er afgørende, da programmering ikke kun handler om at kode, men om at skabe løsninger på reelle problemer ved at bruge teknologi (Caspersen et al. 2018). I CT er der særligt fire forskellige nøglekoncepter, der er essentielle at lære som beskrevet af Wilensky og Rand 2015.

- **Dekomponering:** Programmering af større såvel som mindre systemer kræver, at komplekse problemer opdeles i håndterbare dele. Ved at lære at dekomponere problemer kan programmører fokusere på enkelte aspekter af problemet ad gangen, hvilket fører til en større overskuelighed og derfor effektiv kode .

- **Mønstergenkendelse:** Dette element af CT hjælper en programmør med at identificere fællestræk ved forskellige problemer og genbruge eksisterende løsninger. For eksempel kan et loop, der bruges til at behandle lister, genbruges med små tilpasninger i forskellige dele af et program.

- **Algoritmisk tænkning:** At kunne tænke i algoritmer er fundamentalt i programmering. Det involverer design af trin-for-trin procedurer for at løse en opgave eller komme frem til en konklusion. Denne tænkning hjælper med at strukturere kode på en logisk og effektiv måde.

- **Abstraktion:** I programmering er abstraktion et værktøj til at kunne skjule kompleksitet og kun fokusere på det relevante. Abstraktion bruges derfor til at definere funktioner og klasser,

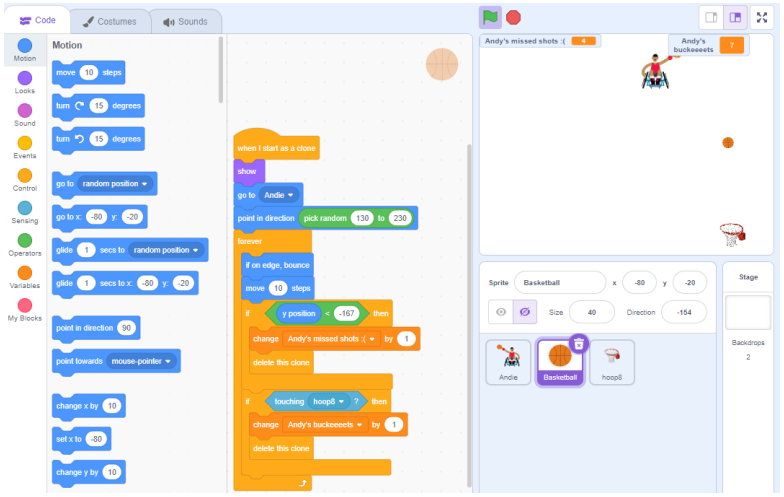
der hjælper med at forenkle komplekse systemer og gøre dem mere håndterbare (Wilensky og Rand 2015).

Når man lærer at programmere, er det ikke nok at kunne syntaksen af et programmeringssprog. Det er lige så vigtigt at forstå, hvordan man anvender CT for at tilgå og løse problemer på en systematisk måde (Caspersen et al. 2018). For eksempel kan et problem som dataanalyse kræve, at data først dekomponeres, derefter analyseres for mønstre, og slutteligt behandles gennem algoritmer, der filtrerer eller sorterer data baseret på definerede kriterier (Wilensky og Rand 2015). Desuden muliggør en stærk grundlæggende forståelse af CT, at programmører kan tilpasse sig nye teknologier og programmeringssprog hurtigere, da de fundamentale problemløsningsmetoder forbliver konstante, selvom værktøjerne og teknologierne ændrer sig (Caspersen et al. 2018).

2.5.1 Visuelle programmeringssprog og Computational Thinking

I CS50 bruges det blokbaserede sprog *Scratch* til at introducere studerende til CT, før de skal kaste sig over tekstbaserede kodesprog. Resnick et al. 2009 understreger, at programmering understøtter CT ved tydeligt at visualisere udviklerens problemløsningsproces gennem det program, de bygger. De var inspirerede af Seymour Paperts argumenter for, at et programmeringssprog skulle have tre kvaliteter (Papert 1980). Han mente, at et programmeringssprog skulle have henholdsvis *"low floor"* (nemt at begynde på), *"high ceiling"* (muligheder for at lave programmer med stigende kompleksitet med tiden) og *"wide walls"* (understøtte projekter af mange forskellige typer). Med den tankegang er *Scratch* et forsøg på at gøre programmering så let som muligt og tilgængelig for alle, med mulighed for at lave mange forskellige typer projekter og med mulighed for at lave store, komplekse projekter (Resnick et al. 2009). Med det argumenterer Resnick et al. 2009, at CT kan faciliteres på mange forskellige niveauer, men særligt at personer uden nogen form for erfaring kan begynde at bygge et CT-fundament med det samme.

I CS50's første uge, også kendt som "week 0,"introduceres de studerende til grundlæggende programmeringskoncepter ved hjælp af *Scratch*.



Figur 2.7: Eksempel af vores *Scratch* "basketball-spil"fra 'week 0' i CS50 kurset lavet af et af gruppemedlemmerne

Opgaven består i, at de studerende bliver bedt om at lave et simpelt spil. Når de studerende går til opgaven, skal eller kan de anvende flere centrale CT-principper, for eksempel på følgende måde:

- **Dekomponering:** De studerende skal opdele projektet i mindre dele såsom karakterbevægelser, interaktioner og mål.
- **Mønstergenkendelse:** Genbruge og tilpasse eksisterende kodeblokke til forskellige funktioner i projektet.
- **Algoritmisk tænkning:** Designe en sekvens af trin, der styrer karakterens handlinger og interaktioner.
- **Abstraktion:** Bruge funktioner og variabler til at forenkle og strukturere koden.

2.6 Essence - Problem-Solution Model Generation

I det følgende afsnit præsenteres Ivan Aaen's Essence: Problem-Solution Model Generation, der anvendes i dette projekt. Først redegøres der for teorien og grundantagelserne bag Essence. Dernæst redegøres der for Problem-Solution Canvas og dets byggeblokke, der bruges som Visual Inquiry Tool igennem projektet. Der redegøres for de forskellige Fulcrums i Essence, der har haft betydning for påbegyndelsen af vores designproces. Til sidst redegøres der for Problem-, Leverage-, og Prospect Scenarios, der er blevet anvendt til at diskutere og navigere i forskellige problemer og løsninger igennem projektet.

'Essence: Problem-Solution Model Generation', fremover benævnt som 'Essence', er en designmetodologi udviklet af Ivan Aaen til problemer karakteriseret ved kompleksitet og uvished (Aaen

2024). Metoden er baseret på 'Dewey's pragmatisme' og karakteriserer design som "...en fortsat samtale mellem problem og løsning, en samtale hvor problemer former løsninger og løsninger former problemer"(Aaen 2024). Aaen gør sig fire observationer, der begrunder, hvorfor design nu finder sted under ustabile forhold.

1. **The Anthropocene epoch: A man-made world**, hvor design er en del af tidligere, nuværende og fremtidige menneskelige bestræbelser og tager del i interaktive økosystemer (Aaen 2024).
2. **Knightian Uncertainties: Fundamental unpredictabilities**, som eksempelvis en global epidemi som COVID-19 og hvordan den skabte og ødelagde markeder og virksomheder (Aaen 2024).
3. **Hypercomplexity**, hvor design, særligt i software-fokuseret udvikling, foregår på et niveau af kompleksitet, der forhindrer eller umuliggør rationel beslutningstagning inden for en rimelig tidsramme (Aaen 2024, s. 4).
4. **Problems and solutions are interdependent**. De problemer, der forsøges løst, vælges, opfattes og afgrænses af de løsningsmuligheder, vi ser. Ligeledes afgrænser vi hvilket problem, vi løser, med valgte løsningsmuligheder. Problemer og løsningsmuligheder er altså gensidigt afhængige af hinanden (Aaen 2024, s. 4).

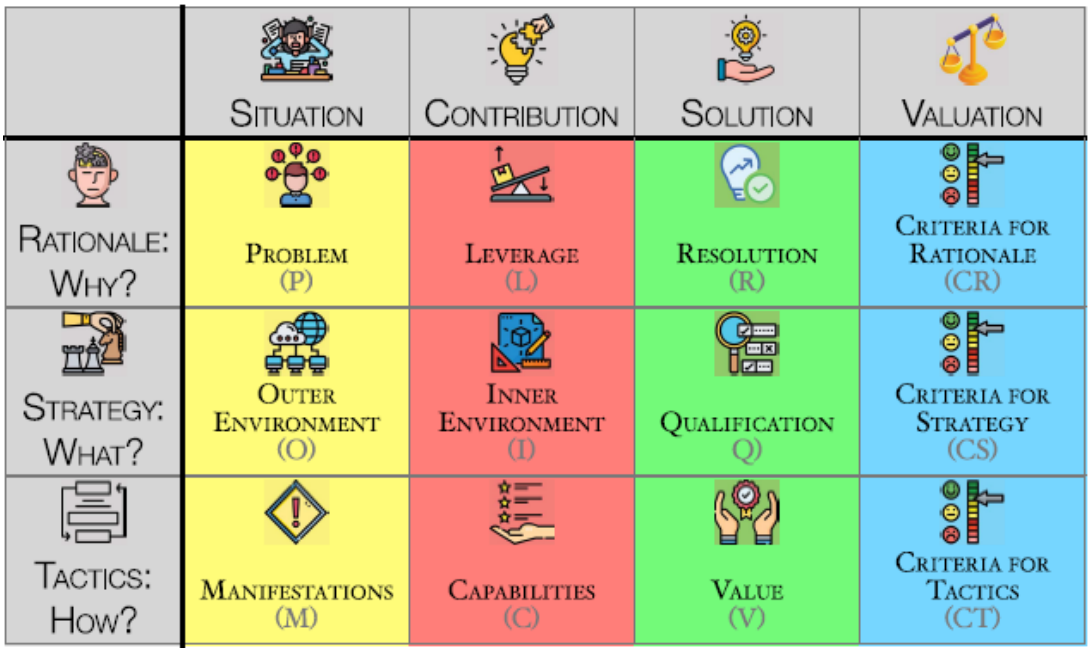
Udfordringerne ved at designe under ustabile forhold kan opsummeres med 'VUCA', beskrevet af Bennis og Nanus 1985. VUCA er et akronym, der dækker over *Volatility*, *Uncertainty*, *Complexity*, *Ambiguity*. Design- og innovationsmetoder bygger på viden fra tidligere lignende løsninger (Aaen 2024, s. 4), men i Essence argumenteres der for, at ustabile betingelser betyder, at ligheder mellem tidligere og nuværende løsninger ikke kan antages (Aaen 2024, s. 4). Med VUCA-udfordringerne kræves der derfor en eksperimentel og fleksibel arbejdsproces, hvor løsningerne ikke er *planlagt* så meget som de løbende *vokser* (Aaen 2024, s. 5) (Bennis og Nanus 1985).

2.7 Problem-Solution Canvas

Den eksperimentelle og fleksible arbejdsproces kan understøttes af Problem-Solution Canvas, fremover benævnt 'PSC'. PSC er et Visual Inquiry Tool, et værktøj der anvender visuelle elementer til at støtte refleksion og fremme samarbejde mellem designere (Aaen 2024, s. 13). I Essence-modellen er det ikke en forudsætning, at man altid skal starte med at identificere et problem for derefter at designe en løsning (Aaen 2024). I stedet kan en løsning og et behov for en løsning opstå sammen eller tilfældigt, men et meningsfuldt design vil til sidst have et forenet perspektiv på problemer og løsninger (Aaen 2024). PSC faciliterer det forenede perspektiv ved løbende at adressere fire kerneområder af et projekt, der afspejler projektets status og retning (Aaen 2024). *Situation* afspejler problemet og dets kontekst. *Contribution* afspejler, hvad der designes som del af en løsning. *Solution* afspejler, hvordan

Contribution bidrager til en løsning, mens *Valuation* afspejler, hvordan projektet bevæger sig i den rigtige retning for at løse problemet.

Alle fire kerneområder diskuteres på tre abstraktionsniveauer; *Rationale, Strategy* og *Tactics*. Det laveste abstraktionsniveau, *Strategy*, omhandler den samlede idé til at løse problemet. Det inkluderer diskussioner vedrørende problemets omfang samt hvilke komponenter, der er kritiske for at bygge løsningen. Det indebærer ligeledes berøring med begrænsninger, der kan påvirke, om løsningen accepteres eller ej. *Tactics* indebærer overvejelser og diskussion af specifikke handlinger, der tages med henblik på at opnå specifikke delmål indeholdt i strategien. *Rationale* udgør det logiske fundament for projektet, der giver handlingerne mening og understøtter ræsonnementet bag *Strategy* og *Tactics* (Aaen 2024).

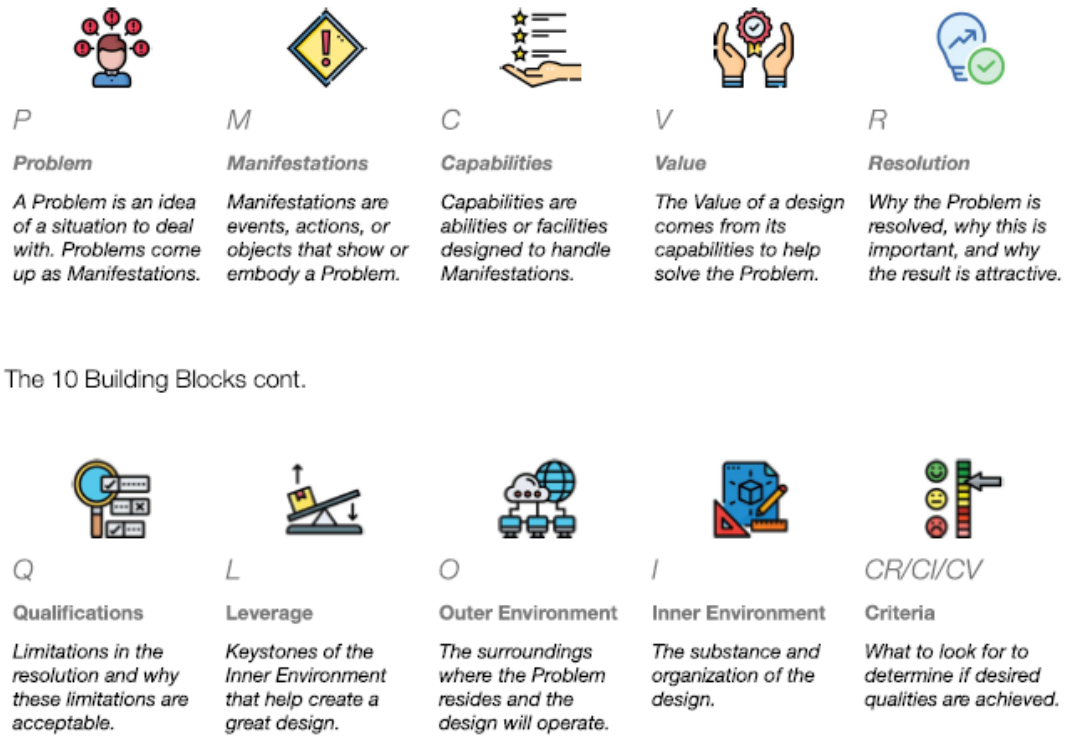


Figur 2.8: Problem-Solution Canvas (Aaen 2024, s. 17)

Hvert kerneområde kan opdeles i ovennævnte abstraktionsniveauer. Med andre ord kan kerneområdet 'Situation' diskuteres på tre niveauer af abstraktion. PSC indeholder 12 byggeblokke, der hver repræsenterer et kerneområde på et specifikt abstraktionsniveau. Ovenstående figur illustrerer, hvordan eksempelvis kerneområdet 'Situation' indeholder byggeblokkene 'Problem' på rationale-niveau, 'Outer environment' på strategy-niveau og 'Manifestations' på tactics-niveau.

2.7.1 Byggeblokkene i PSC

En uddybende forklaring af hver byggeblok vil blive præsenteret løbende i projektet, i takt med at hver byggeblok bliver diskuteret, brugt og udfyldt.



Figur 2.9: De 10 Problem-Solution byggeblokke (Aaen 2024, s. 16)

2.7.2 Kerneområdet 'Situation'

I kerneområdet *Situation* i figur 2.9 eksisterer byggeblokkene *Problem*, *Outer Environment* og *Manifestations*. Den første byggeblok, *Problem*, repræsenterer udfordringen, der skal håndteres med designet. Essence definerer denne som "*A Problem reflects an understanding of a situation and is used as an analytical instrument. It is formulated as part of inquiring into a problematic situation. Problems, therefore, have no objective status.*". Det menes dermed, at problemet ikke blot er en objektiv udfordring; det repræsenterer også andre dimensioner, som er underlagt den kontekst og de perspektiver, hvorfra det betragtes (Aaen 2024, s. 19). Problemet vil ændre sig alt efter, hvem der undersøger det, hvem og hvad der undersøges, og vil derfor også ændre opfattede, mulige løsninger. Problemet afspejler derfor en *forståelse*, der udvikler sig løbende, efterhånden som den problematiske situation bliver undersøgt. I Problem-Solution Canvas (PSC) giver dette problem det rationelle grundlag for at udforske og designe løsninger.

Outer Environment-blokken repræsenterer omgivelserne, hvori løsningen skal eksistere. Essence

definerer det som "*The surroundings in which our Problem resides and where the Contributions we design will operate. External services, implements, repositories, and people are examples of elements in the Outer Environment*". Som tidligere beskrevet, vil en nuværende og fremtidig løsning eksistere og tage del i interaktive økosystemer. Beslutninger om, hvad der inkluderes i Outer Environment, påvirker omfanget af løsningen, og derfor er Outer Environment en del af *Strategy*.

Manifestations-blokken repræsenterer Manifestationer af problemet, altså fremtrædelsesformer af et problem. Essence definerer det som "*An object, event, or action that reflects or gives a tangible or visible form to the Problem. Capabilities might match such objects, events, or actions as part of the project's Contribution.*"

2.7.3 Kerneområdet 'Contribution'

I kerneområdet *Contribution* eksisterer *Leverage, Inner Environment* og *Capabilities*. På tactics-niveau eksisterer *Capabilities*, der defineres i Essence som "*An ability or facility devised to handle Manifestations and provide Value as part of a Solution.*"Capabilities er derfor direkte forbundet til Manifestations og kan ses som de specifikke *Contributions* designet til at håndtere hver manifestation af problemet. Hver Manifestation skal håndteres af en eller flere Capabilities.

Essence definerer *Leverage* som nøgleelementer i det *Indre Miljø*, der kan styrke en løsning ved at tilbyde forbedrede funktioner, lavere omkostninger eller større fleksibilitet. Disse elementer inkluderer teknologier, komponenter, information og interne menneskelige ressourcer, der er unikke og ikke frit tilgængelige, hvilket giver designet en konkurrencemæssig fordel. På det strategiske niveau angiver det *Indre Miljø* substansen og organisationen af det, vi designer og implementerer, hvor leverage spiller en central rolle (Aaen 2024, ss. 30-31).

2.7.4 Fulcrums

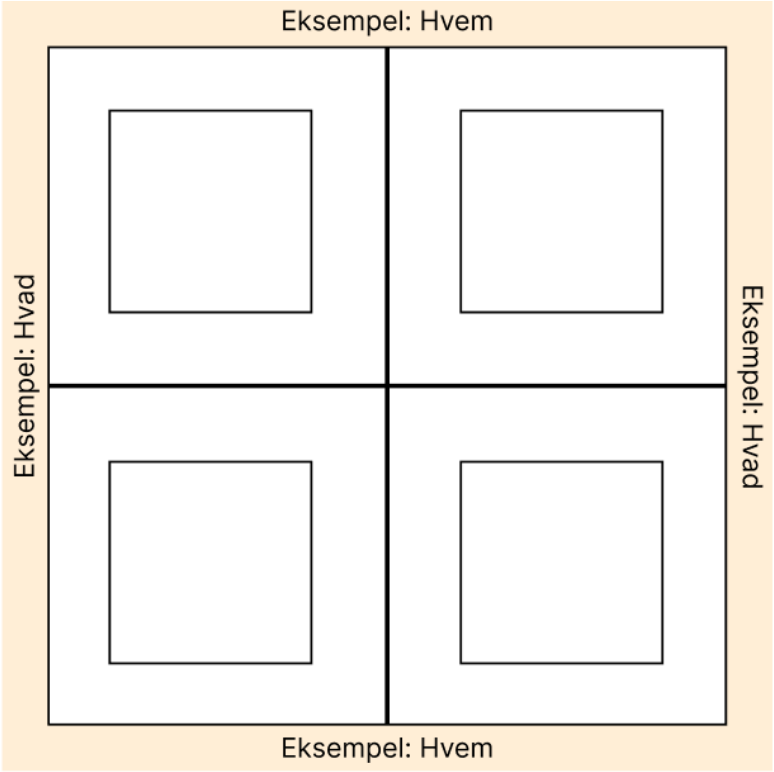
I Essence kan designprocessen begynde ved andre udgangspunkter end selve problemet. I Essence eksisterer der fire *Fulcrums*, der hver repræsenterer et startsted for digital innovation. Disse fire er *Manifestation-Fulcrum*, hvor der fokuseres på kerneområdet *Situation*, *Capability-Fulcrum*, hvor der fokuseres på kerneområdet *Contribution*, *Value-Fulcrum*, hvor der fokuseres på kerneområdet *Solution*, og slutteligt *Criteria-Fulcrum*, hvor projektet fokuserer på kerneområdet *Valuation*. I dette projekt påbegyndes designprocessen ved at undersøge områder, hvori vores uddannelsesmæssige og personlige kompetencer kan udgøre en fordel. I Essence svarer det til at påbegynde PSC ved at undersøge Contribution først, og dermed et *Capability-Fulcrum*.

2.7.5 Problem scenario

I Essence bruges *Problem Scenario* til bedre at analysere alternative forståelser af et problem. Et problem er ofte ikke bare et problem, hvor årsagen og effekten er klar; der kan være flere nuancer til et problem eller forskellig kontekst, som det bliver beskrevet. Der opereres på fire forskellige niveauer, når det kommer til et problems kontekst; *simpel, kompliceret, komplekst* og *kaotisk*. *Simpel* bruges til at beskrive de problemer, hvor der ikke er tvivl om, hvad der forårsager et problem, og der ligeledes er et korrekt svar til. Problem scenarier skal ikke bruges til at håndtere disse problemer, da det er et værktøj til netop at håndtere problemer, der ikke har én korrekt løsning. *Komplicerede* problemer er en kontekst, hvor en årsagssammenhæng ikke er klar for alle, men hvor der er mulighed for at finde det gennem analyse. Konteksten af *komplekse* problemer indebærer endnu større uforudsigelighed, og det kan derfor være vanskeligt præcist at analysere sig frem til en årsagssammenhæng. Der vides ikke, hvad der vil virke, og der er derfor mange konkurrerende idéer. Derfor skal idéer undersøges, opfattes og derefter reageres på. Man vil derfor lede efter mønstre, der kan håndtere den forandring og uforudsigelighed. Den sidste problemkontekst er *Kaotisk*, et område der er præget af de tidligere beskrevne *Knightian Uncertainties*, og der er ingen årsagssammenhænge. Grundet det kaos, der eksisterer på dette niveau, handles der intuitivt; man opfatter situationen og reagerer hurtigt. Målet er her at forsøge at identificere mønstre ved at se på løsninger, der kan hjælpe med at håndtere problemet.

Problem scenarier angribes typisk ved, at man bruger de velkendte "5W2H", på dansk: Hvem, Hvad, Hvor, Hvornår, Hvorfor, Hvordan og Hvor meget. Ved at forsøge at svare på spørgsmålene får man en dybere forståelse af de forskellige aspekter et problem indeholder, og på den måde gør man problemet mere håndgribeligt.

Til at udforske de forskellige aspekter opdeles problemet i dets scenarier i fire forskellige kvadranter via to dikotomiske aksiale spørgsmål, en vertikal og en horisontal. Rammerne for hvilke spørgsmål, der skal lede en, styres frit og tilpasses efter, hvad der passer til problemet.



Figur 2.10: Eksempel på opsætningen af et scenario

Til den vertikale akse vælges der typisk 'hvem' eller 'hvorfor' for at fokusere på, hvem problemet påvirker eller hvad årsagen til problemet er. Til den horisontale bruges der typisk de resterende: hvad, hvor, hvornår, hvordan og hvor meget til at udforske forskellige synspunkter på problemet.

Ved at bruge de fire kvadranter skaber man ligeledes forskellige scenarier til hver kvadrant, som man herefter vil forsøge at sætte ord på. På denne måde udforsker man forskellige scenarier, som kan bidrage til en bedre forståelse af problemet (Aaen 2024, ss. 55-59).

2.7.6 Leverage scenario

Til at hjælpe med at finde Leverage points foreslår Essence, at man gør brug af leverage scenarios. På samme måde som problem scenarios eksisterer der fire typer, her kaldet leverage kategorier. Den første kategori er *teknologi*, som kan bestå af viden, maskiner eller andet udstyr, som kan være et leverage point og forme en løsning. *Komponenter* er en kategori, der karakteriseres ved, at man har for eksempel sensorer eller måleenheder, der bruges til at udforme en løsning. Den tredje kategori *information* karakteriseres ved forskellige måder at håndtere information på; det kan være måder at indsamle eller opbevare data på, eller det kan være adgang til en speciel API, der muliggør en given løsning. Den sidste kategori er *interne menneskelige ressourcer*, som er de forskellige mennesker, man kan have adgang til, når der udvikles en løsning. Det kan for eksempel være eksperter, adgang til

specifikke brugere eller et opbygget netværk, der kan have indflydelse på de løsningsmuligheder, man har (Aaen 2024, ss. 65-69).

Samme princip som fra problem scenarierne med de fire kvadranter opdelt af dikotomiske akser, begge bruger de "5W2H", for bedre at kunne udforske forskellige leverage points. Grunden til, at man ønsker at udforske alternative leverage points, er, at det ikke nødvendigvis er det mest indlysende leverage point, der er det bedste. I Essence bruges det sigende citat fra Maslow: "*It is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail*"(Aaen 2024, s. 65).

2.7.7 Prospect scenario

Ligeledes som de forrige scenarier arbejder vi her i de fire kvadranter opdelt af de dikotomiske akser, men her udforskes de overordnede idéer som givne løsninger på problemet. Dette gøres for at finde det bedste løsningsprospekt til at løse et problem, altså en løsning man erkender har mulighed for at ændre sig i takt med, at man lærer mere, som ordet prospect antyder (Aaen 2024, ss. 76-81).

Da prospect scenarios er en kombination af de to foregående scenarier, problem- og leverage scenarier, bruges her derfor to spørgsmål, et fra begge af de tidligere scenarier, på både den horisontale og vertikale akse. Dette gøres altså for at anskue problemet sammen med de mulige løsninger, for at kunne anskue de forskellige prospects; altså problemet og forskellige løsninger som et samlet design (Aaen 2024, ss. 76-81).

2.8 Forberedelse og strategi

2.8.1 Effectuation

Sarasvathy 2001 har konceptualiseret to strategier til forretningsudvikling, som for os er relevante at forholde os til: Causation og Effectuation. Causation går i sin enkelthed ud på at identificere og forfølge årsager og planlægge handlinger for at opnå en forudbestemt effekt. Sarasvathy 2001 beskriver Effectuation som værende det modsatte af Causation, hvor der fokuseres på at evaluere tilgængelige midler, overveje mulige handlinger og vurdere de potentielle virkninger af disse handlinger. Sarasvathy fortæller, at valget mellem Causation og Effectuation som strategi kan være et spørgsmål om, hvilken kontekst man befinder sig i (Sarasvathy 2001, s. 4). Her placerer hun Effectuation som den optimale strategi, når situationen er karakteriseret af høj kontrollerbarhed og lav forudsigelighed (Kraaijenbrink 2012).

Vi argumenterede for, at vores kontekst var præget af de vilkår, der beskrives som optimale for Effectuation. Teknologiforståelse er præget af usikkerhed, i og med at det er et politisk initiativ, der genovervejes og genforhandles løbende (Grundskolen 2024). Dette indebærer også løbende modifikationer til bekendtgørelser, fagbeskrivelser og regler på området. Vi argumenterede derfor, at vi ville have behov for at kunne tage højde for nye muligheder og udfordringer og manøvrere igennem disse. På den måde anså vi Effectuation-strategien for at være optimal, da den tillader os at forsøge at

kontrollere fremtiden frem for at forudsige den. Ved at *arbejde med, hvad vi har*, og lade målene udvikle sig dynamisk som svar på, hvad vi lærer undervejs, kunne vi sikre os en højere grad af tilpasningsdygtighed og fleksibilitet (Sarasvathy 2001).

2.8.2 Effectuation-principper

Effectuation-strategien bygger på fem forskellige principper, vi har brugt til at hjælpe os med at navigere i den usikre og uforudsigelige situation, vi befandt os i:

- **Bird-in-Hand-princippet:** Start med hvem du er, hvad du ved, og hvem du kender. Dette princip fokuserer på at udnytte de ressourcer, der allerede er tilgængelige. I stedet for at vente på de ideelle betingelser eller ressourcer, begynder man med det, man har ved hånden. Ved at identificere og bruge eksisterende færdigheder, netværk og viden kan man hurtigt tage de første skridt.

- **Affordable Loss-princippet:** Vær villig til at risikere det, du har råd til at tabe. Dette princip handler om at begrænse risikoen ved at fokusere på, hvad der kan overkommes, snarere end at maksimere potentielle gevinster. Det betyder, at vi kun investerer ressourcer, som vi har råd til at tabe, og derved undgår større tab, hvis tingene ikke går som planlagt.

- **Crazy Quilt-princippet:** Skab partnerskaber med andre relevante parter. Dette princip involverer at samarbejde med andre, der har en gensidig interesse i at skabe værdi sammen. Gennem samarbejde med partnere kan man skabe nye muligheder og reducere usikkerheden. Dette hjælper med at bygge et netværk af ressourcer og viden, som styrker projektet.

- **Lemonade-princippet:** Udnyt uforudsigelige hændelser som muligheder. Når uventede begivenheder opstår, skal de betragtes som muligheder for at skabe noget nyt og værdifuldt. Dette princip opfordrer til fleksibilitet og åbenhed over for forandringer, hvilket kan føre til uventede og positive resultater.

- **Pilot-in-the-Plane-princippet:** Princippet understreger, at man bør tage kontrol over projekter ved at udnytte tilgængelige ressourcer og samarbejde med interessenter. I stedet for blot at tilpasse sig fremtidige usikkerheder, skal man aktivt forme fremtiden gennem fælles indsats og strategiske beslutninger med fokus på, hvad man kan kontrollere frem for at stole på forudsigelser (Effectuation.org 2024).

2.8.3 Abduktiv tilgang

I dette projekt tilgik vi problemet *abduktivt*. En abduktiv tilgang er en forskningsmetode, der bruges til at opstille hypoteser og teorier gennem en iterativ proces, hvor observationer og teoretiske rammer veksler mellem at informere og forme hinanden. Denne metode er særlig nyttig i komplekse og usikre situationer, hvor hverken deduktiv (fra teori til observation) eller induktiv (fra observation til teori) tilgang alene ville være tilstrækkelig (Creswell 2014). Aaen 2024 beskriver, at Essence ikke er baseret på antagelser om kausalitet, sekvens eller proces. I Essence antages det ikke, at den bedste fremgangsmåde altid er at starte ved et problem og derefter designe en løsning (Aaen 2024). I stedet kan en løsning og et behov for en løsning opstå sammen eller tilfældigt, men et meningsfuldt design vil til sidst have et forenet perspektiv på problemer og løsninger (Aaen 2024). Sat i relation til en abduktiv tilgang ser vi de to tilgange til rækkefølge som forenelige.

2.8.4 Capability fulcrum

I Essence antages det, at teams har et repertoire af tilgængelige teknologier, komponenter, information eller menneskelige ressourcer. Som tidligere beskrevet, påbegyndte vi design processen ved at undersøge områder hvori vores udannelsesmæssige- og personlige kompetencer kan udgøre en fordel. I Essence svarer det til at påbegynde PSC ved at undersøge Contribution først, og dermed et Capability-fulcrum. Her var vi også inspireret af Effectuations *Bird in Hand Principle*, der understreger vigtigheden af at starte med, hvad man allerede har, som grundlag for handling. Ifølge det princip bør der startes med at vurdere tilgængelige ressourcer, som *Hvem vi er, hvad vi ved og hvem vi kender* (Sarasvathy 2001).

2.9 Seymour Paperts konstruktionisme

I udviklingen af vores løsning er vi inspireret af Seymour Paperts konstruktionisme (Papert 1980). Papert, som var en pioner inden for både uddannelsesteknologi og pædagogisk teori, udviklede konstruktionismen som en videreudvikling af Piagets konstruktivistiske teori. Konstruktionisme bygger på ideen om, at mennesker lærer bedst gennem aktiv deltagelse og ved at skabe konkrete artefakter, som de kan reflektere over og diskutere. Denne tilgang understøtter læring gennem erfaring og interaktion med verden, hvilket gør læring både meningsfuld og engagerende for eleverne (Papert 1980).

Papert anså computere som særligt kraftfulde værktøjer til at fremme konstruktionistisk læring. Han mente, at computere kunne fungere som "teaching machines,"der tillader elever at eksperimentere, bygge og udforske på måder, der tidligere ikke var mulige. Dette sker gennem programmeringssprog som LOGO, som Papert udviklede for at give børn mulighed for at styre og programmere små robotter, kaldet "turtles", for at se direkte resultater af deres kode. På denne måde bliver læring en

dynamisk proces, hvor eleverne lærer ved at skabe, teste og iterere på deres egne projekter (Papert 1980).

2.9.1 Things to Think With

Et centralt begreb i Paperts konstruktionisme er "things to think with."Papert brugte dette udtryk til at beskrive de konkrete objekter og værktøjer, som eleverne kan manipulere og eksperimentere med for at udvikle deres forståelse af komplekse koncepter. Disse "things to think with"kan være fysiske objekter, såsom LEGO-klodser eller robotter, eller virtuelle objekter, såsom software og programmeringssprog. Vi er inspirerede af Paperts idé om "things to think with"og integrerer denne tankegang i vores løsning ved at tilbyde de studerende forskellige værktøjer, som de kan bruge til at udforske og lære. For eksempel, når de studerende arbejder med visuelle programmeringssprog som Blockly, skaber de mentale modeller, der kan hjælpe dem med at forstå abstrakte koncepter. Dette skift fra konkret manipulation til abstrakt tænkning er kernen i konstruktionistisk læring (Papert 1980).

2.10 Lean Startup og Minimal Viable Product

Lean Startup, en metodologi udviklet af Eric Ries, er designet til at hjælpe startups med at innovere hurtigt og effektivt ved at reducere spild og maksimere læring gennem kontinuerlige iterationer og validering af deres ideer. Denne tilgang er baseret på feedback-løkker, hvor man konstant tester hypoteser, lærer af resultaterne og tilpasser sig derefter (Ries 2011).

2.10.1 Minimum Viable Product (MVP)

Et centralt koncept i Lean Startup er Minimum Viable Product (MVP). En MVP er den simpleste version af et produkt, der kan bygges og frigives, samtidig med at den stadig giver mulighed for at indsamle den maksimale mængde valideret læring om kunderne med mindst mulig indsats. Formålet med en MVP er at teste grundlæggende antagelser og hypoteser om produktets værdi og funktionalitet tidligt i udviklingsprocessen, så ressourcer ikke spildes på at udvikle funktioner, som kunderne ikke ønsker eller har brug for (Ries 2011).

MVP er et praktisk supplement til Effectuation, fordi det muliggør hurtig eksperimentering og læring i usikre og uforudsigelige miljøer. Mens Effectuation fokuserer på at arbejde med de midler, man har, og lade målene udvikle sig dynamisk (Sarasvathy 2001), tillader MVP-strategien, at disse midler anvendes på en måde, der maksimerer læringen og tilpasningsevnen. Ved at lancere en MVP kan man hurtigt få feedback fra brugere, hvilket hjælper med hurtigt at justere retningen (Ries 2011).

2.11 ChatGPT

I dette projekt har vi anvendt AI i form af ChatGPT til en række formål. Flere medlemmer i vores gruppe har læsevanskeligheder, og har anvendt ChatGPT til følgende:

- At rette grammatik og tegnsætning.
- Som stavekontrol på hele afsnit
- At oversætte engelske begreber til dansk
- At omformulere tekststykker

Vi har ikke anvendt AI til generative formål, såsom at producere originalt indhold i dette projekt.

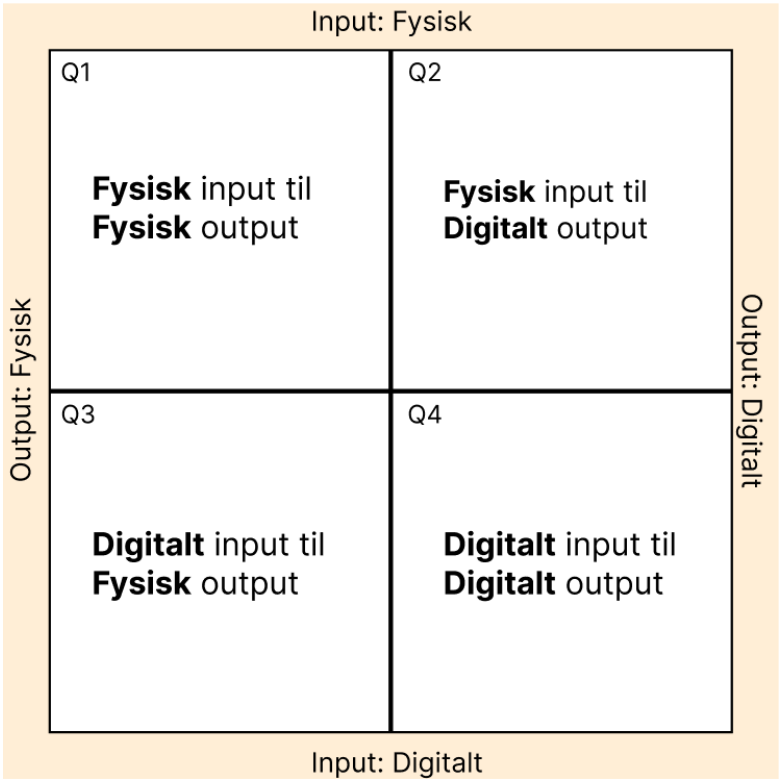
Related work

I dette kapitel præsenteres fundende fra vores indledende litteraturstudie. For at danne os et overblik over eksisterende løsninger, foretog vi et litteraturstudie med søgestrengen "Tangible programming AND Computational Thinking". Denne søgestreng var baseret på en indledende problemformulering *"Hvordan kan et fysisk design facilitere Computational Thinking?"*. Fundende herfra er præsenteret i nedenstående afsnit som følgende. Først præsenteres Funk et al. 2021s klassifikation af fysiske løsninger, hvorefter vi præsenterer et framework inspireret af Essences Prospect Scenarios, der inkluderer Visuelle programmeringssprog. Dernæst indgår vi i en beslutningsproces, hvor vi fremlægger en række *Leverage points* udarbejdet på baggrund af vores egne personlige baggrunde.

3.1 Klassifikation af fysiske- og digitale løsninger

Funk et al. 2021 har klassificeret en lang række fysiske programmeringssprog og løsninger i forhold til input, output og kompleksitet. Funk et al. 2021 behandler konceptet om *Tangible solutions* ud fra en række ligheder ved hvert design. Funk et al. 2021 klassificerer de løsninger ud fra enten digitalt eller fysisk input og output. Med det menes input som værende den grænseflade brugeren interagerer med, hvor output repræsenterer den *ydre repræsentation* der skabes, som beskrevet af Resnick et al. 2009. Et fysisk input indebærer at brugeren skal programmere ved brug af håndgribelige elementer, i tråd med *tangible programming*. Det kunne være fysiske brikker, ledninger eller lignende, hvor brugeren manipulerer håndgribelige objekter for at skabe et input. Et digitalt input indebærer at brugeren programmerer digitalt, i tråd med *visuel programmering*. Det kunne eksempelvis være digitale puslespilslignende brikker der repræsenterer kodestykker som i Scratch og Blockly. Et fysisk- eller digitalt output adskiller sig ved, om den ydre repræsentation af koden, i form af eksempelvis en Robot der bevæger sig, er af fysisk eller virtuel karakter.

Vi opstillede en ny model i figur 3.1 på baggrund af deres klassificering, der blev brugt som genstand for diskussion om mulige beslutninger. Vi var inspireret af Aaens *Prospect Scenarios*, der kan bruges til at udvikle og kontrastere alternative idéer for samtidigt at kunne definere problemer og løsninger (Aaen 2024). Med den tankegang forsøgte vi at lære, hvilke forskellige problemer de forskellige designs var effektive til at løse.



Figur 3.1: Løsnings model: Fysisk vs. digitalt

De fire kvadranter repræsenterer hver en samlet række af eksisterende løsninger. I figuren er der abstraheret fra konceptet om *konsoliderede* eller *separerede* løsninger beskrevet af Funk et al. 2021. Med det menes, at en række af løsninger er et separat input og/eller output som i det fysiske programmeringssprog *Tern*, der kan anvendes til en lang række outputs. En konsolideret løsning, derimod, er løsninger hvor input og output er designet specifikt til hinanden, som i *Thymio*, hvor det fysiske input er rettet specifikt mod en fysisk robot (Funk et al. 2021).

- ***Q1*:** I kvadrant 1 (Q1) baseres designet på et fysisk input og output. Inklusion af det fysiske input, opfordrer naturligt til samarbejde og aktiv deltagelse (Horn et al. 2009), og kan være velegnet til at facilitere computational thinking gennem leg og problem-løsning, sociale interaktioner og kollaborative øvelser (Funk et al. 2021). Flere studier pointerer ligeledes, at det kollaborative og participatoriske aspekt af fysiske løsninger har effektivt bidraget til at reducere kønsforskelle (mange kilder fra funk et al). Både Horn et al. 2009 og Sapounidis og Demetriadis december 2013 fandt at særligt yngre børn i alderen 5-8 år fandt de fysiske grænseflader nemmere at bruge, sjovere at bruge og mere tiltrækkende end den virtuelle tilsvarende løsning.
- ***Q2*:** I kvadrant 2 (Q2) baseres designet på et fysisk input som i ovenstående, men et digitalt out-

put. Designet ville således kunne bygge på Paperts konstruktionisme som i ovenstående, men uden den øjeblikkelige, fysiske feedback. Et eksempel på sådan en løsning er T-maze, hvor fysiske kodeblokke kan bruges til at gennemføre en digitalt repræsenteret labyrint (Wang, Wang og Liu februar 2014). Michael S. Horn 2012 argumenterer for, at en *hybrid tilgang* kan være effektivt, i situationer hvor det fysiske aspekt skaber begrænsninger. Fælles for de løsninger der anvender fysisk input i Funk et al. 2021s model, og vores Q1 og Q2, er at det anvendte sprog har lav grad af kompleksitet.

- ***Q3*:** I kvadrant 3 (Q3) baseres designet på et digitalt input og et fysisk output. Løsninger af denne slags er karakteriseret ved oftest at anvende sprog med høj grad af kompleksitet i Funk et al. 2021. I LEGO Mindstorm programmerer brugeren en fysisk robot i et digitalt, visuelt programmeringssprog. Således inkluderes en håndgribelig oplevelse i form at et manipulerbart, fysisk objekt, samtidigt med at det visuelle programmeringssprog tillader en høj grad af fleksibilitet og kompleksitet (Scharf, Winkler og Herczeg 2008). Således argumenterer Funk et al. 2021 også for, at disse løsninger i højere grad kan anvendes til indlæring af computational thinking på et mere komplekst niveau, og således egner sig til en ældre aldersgruppe end løsninger med fysiske inputs og lav kompleksitet. Horn et al. 2009 og Sapounidis og Demetriadis december 2013 fandt at de yngste børn ligeledes havde sværere ved at anvende de digitale grænseflader, noget Scharf, Winkler og Herczeg 2008 forklarer ved at de yngste børn endnu ikke har udviklet tilstrækkelig finmotorik til effektivt at anvende eksempelvis en computermus. De kollaborative og participatoriske aspekter af løsningen mindskes ligeledes (Funk et al. 2021), da samarbejde omkring en computer eller tablet er mere udfordrende (Scharf, Winkler og Herczeg 2008).

- ***Q4*:** I kvadrant 4 (Q4) baseres designet på et digitalt input og et digitalt output. Funk et al. 2021 inkluderede ikke disse løsninger i deres model, da løsningerne ikke længere har fysiske elementer. Det visuelle programmeringssprog *Scratch*, som tidligere beskrevet, anvendes til samme formål som ovenstående (Resnick et al. 2009). Her har teamet bag forsøgt at inkludere nye programmører ved at skabe et sprog der var *low-floor* i.e nemt at gå i gang med, *high-ceiling* i.e mulighed for at lave mere og mere komplekse programmer og *wide-walls* i.e mulighed for at lave en lang række forskellige typer af programmer.

3.1.1 Beslutningsproces

Vores første beslutningsproces tog udgangspunkt i Paperts konstruktionisme og vores litteraturstudie, Sarasvathys effectuation og Aaens Essence. Vi tog udgangspunkt i de ressourcer vi havde til rådighed ud fra Sarasvathy's *Bird in hand principle* og *Affordable loss*. Vi diskuterede de fire ovenstående kvadranter, med henblik på at identificere forskellige *Leverage points*, hvorefter vi besluttede os for en retning at tage på baggrund af de ressourcer vi identificerede.

Fysisk design

I litteraturstudiet blev vi opmærksomme på, hvor stor en del *samarbejde* fylder i læring. Ved at gå i retning af et fysisk design, kunne vi gøre brug af de kollaborative og participatoriske effekter beskrevet af Horn et al. 2009. Derved kunne vi orientere os mod problematikker som social inklusion og inkludering af begge køn (Funk et al. 2021), samtidigt med at vi kan designe ud fra Paperts konstruktionisme som den er beskrevet i afsnit 2.9, hvor brugeren direkte interagerer med fysiske blokke. Det vil sandsynligvis ville være en større udfordring at skabe programmeringssprog med *low-floor, wide-walls* og *high-ceiling* i forhold til et digitalt design (Resnick et al. 2009), Da fysiske programmeringssprog generelt præsenterer en større udfordring i at inkludere højere kompleksitet (Funk et al. 2021). Til gengæld ville vi kunne henvende os til de helt yngste, i eksempelvis faget teknologiforståelse, på baggrund af børns præferencer for fysiske grænseflader beskrevet af Horn et al. 2009.

En udfordring for os ville blive tid. Vi arbejder i en akademisk kontekst med begrænsede ressourcer, særligt i forhold til tid og penge. Vi har sparsom erfaring med fysiske prototyper, hardware og indlejret elektronik - ting vi anså som værende essentielle i et helt eller delvist fysisk design og som går igen i løsningerne inkluderet i (Funk et al. 2021). Vi har mere erfaring med digitale programmeringssprog og digitale grænseflader, hvilket vi anså som værende *Hvad vi ved* i forhold til Sarasvathys *Bird in hand* princip. I forhold til Sarasvathys *Affordable losses* anslog vi et *digitalt* design som værende en mindre, mere håndterbar risiko end et fysisk design. Derudover forsøgte vi at kigge på *Hvad vi har*, og konkluderede at vi i et fysisk design ikke har adgang til væsentlige, eksisterende værktøjer eller ressourcer. I et digitalt design har vi adgang til en stor mængde open-source værktøjer som *Flowgorithm, Blockly, Scratch, Pycparser* mm. Sat i relation til vores begrænsede tid og akademiske kontekst sammen med vores vision om at skabe et konkret bidrag, bevægede vi os væk fra *fysiske* designs, og besluttede os for et helt eller overvejende *digitalt design*.

Digitalt design

Hvor Papert understreger det fysiske aspekt, kunne vi forsøge at genskabe det digitalt på samme måde som *Scratch* eller *Blockly*. Ved at fokusere på et helt eller overvejende digitalt design på trods af den indledende filosofi om *Tangible programming*, kunne vi vinde tid i forhold til at få skabt et konkret bidrag. Vi kunne stadig orientere os mod de helt yngste, om end der ville være udfordringer forbundet med de *digitale* grænseflader sammenlignet med de *fysiske* grænseflader. Vi anså stadig muligheden for at inkorporere fysiske elementer i vores design, men var opmærksomme med at tillæge den *fysiske* del af løsningen for stor betydninger, eftersom vi var bevidste om at det var udenfor vores kompetenceområde.

3.1.2 Leverage points

Som tidligere beskrevet havde vi ikke foretaget brugerundersøgelser i form af interview eller lignende, og var udelukkende inspireret af det materiale vi havde fremfundet i vores litteraturstudie. På baggrund af dette, opstillede vi en række *Leverage points* vores design kunne baseres på eller bygges ud fra. Herefter tog vi udgangspunkt i vores egen erfaring som nye programmører, og forsøgte at identificere forskellige capabilities vi *troede* kunne have en værdi, for bagefter at diskutere hvorvidt vi havde ressourcer og kompetencer til at bygge dem. Nedenstående er vores bud på hvilke *Leverage points* fra Essence vi som team besidder.

1. **Egne erfaringer:** Vi kunne bruge vores egne, nylige erfaringer med at lære et tekst-baseret kodesprog fra bunden som *leverage* til at designe en løsning hurtigere. Ved at tage udgangspunkt i hvad vi selv havde brug for, kan vi forsøge at spejle de erfaringer i andre studerende, og undgå at skulle investere tid i at undersøge alt inden vi handler på det.

2. **Netværk:** Vi kunne benytte os af vores netværk af studerende (igennem et studiejob som Teaching Assistant i CS50 for yngre studerende på DAD) til at teste hyppigt igennem processen, eller be- eller afkræfte antagelser løbende i processen igennem små interviews eller lignende. På den måde kunne vi spare tid og ressourcer i forhold til brugerundersøgelser- og tests, og investere den tid andetsteds.

3. **Færdigheder i specifikke teknologier:** Vi har baggrunde indenfor spil- og webudvikling, særligt i teknologier som Csharp og .Net, React, Typescript og Docker. Ved at centrere vores løsning omkring de teknologier vi kender og har erfaring i, frem for nødvendigvis de teknologier der eventuelt ville være mest egnet, kunne vi hurtigere designe og implementere en løsning. På den måde skulle vi ikke investere tid og ressourcer i at lære nye teknologier.

4. **Bachelorgrader:** Vi har forskellige uddannelser indenfor webudvikling, teknoantropologi og psykologi. Ved at trække på de kompetencer, og den viden vi besidder indenfor de respektive

felter, kan vi skabe et design i krydsfeltet deraf hvor vores teams komparative viden udgør en fordel. Vi kan eksempelvis benytte os af baggrunden indenfor psykologi, til at guide designet ud fra relevante læringsteorier som *konstruktionisme* eller *Cognitive Load Theory*.

CS50 eller Teknologiforståelse

Vi argumenterede for at ovenstående *Leverage points* ville gøre os i stand til at implementere mere samt give os mulighed for at afprøve flere ting i forhold til de kompetencer vi besidder.

Vi anså dette som værende det primære, store gode, i forhold til eksempelvis at investere vores tid i større og mere omfattende brugerundersøgelser- eller tests. På baggrund af vores egne erfaringer med introduktion til tekst-baseret programmering, gik vi derfor i gang med at undersøge hvilke værktøjer vi selv kunne have haft brug for. Givet vores begrænsede mængder af tid og ressourcer, begyndte vi også at overveje hvorvidt vores baggrund udgjorde en fordel i design af en løsning situeret i Teknologiforståelse og folkeskolen. Her tænkes særligt på *Hvem vi kender* i Sarasvathy 2001s *Bird in hand* princip, hvor vi reelt set ikke havde umiddelbar adgang til folkeskoleelever. Efter at have kontaktet enkelte skoler i de indledende uger, blev det tydeligt for os, at den involverede planlægning ville være omfattende. Vores netværk kunne til gengæld være en fordel eftersom vi havde adgang til studerende der netop havde gennemført CS50 kurset. Her begyndte vi derfor også at orientere os mere imod at designe en løsning rettet mod CS50 frem for Teknologiforståelse.

Problem felt

I dette kapitel afgrænser vi vores problemfelt. Først redegøres der for vores egne oplevelser med CS50, hvor vi anvender John Swellers *Cognitive Load Theory* til at beskrive vores oplevelser. Dernæst præsenterer vi vores problemformulering.

4.1 Vores erfaring med CS50

Vi oplevede introduktionen til Computational Thinking og programmering som udfordrende og krævende, omend spændende og stimulerende. Øvelserne i CS50 krævede en stor indsats for at kunne gennemføres til tiden, såfremt man ikke ønskede at falde bagud i forhold til forelæsningsrækken.

4.1.1 Cognitive Load Theory

Vi diskuterede en lang række situationer, for derefter at anvende John Swellers *Cognitive Load Theory* (CLT), som værktøj til at bedre at italesætte vores udfordringer. CLT er et framework der søger at forklare hvordan den menneskelige hjerne processerer information, og hvordan læring kan opnås ved at kontrollere kognitiv belastning (Sweller, Ayres og Kalyuga 2011). Vi anvendte specifikt CLTs opdeling af forskellige typer af belastninger præsenterer sig selv i undervisnings- og læringssituationer. *Intrinsic load* (på dansk: indbygget load) sker af den iboende kompleksitet i det materiale, der skal læres (Sweller, Ayres og Kalyuga 2011). *Extraneous load* (på dansk: støjrelateret load) omhandler den ekstra kognitive byrde, opstår af måden hvorpå informationen præsenteres. *Germane load* (på dansk: læringsbaseret load) beskriver den indsats, der kræves for at opbygge kognitive skemaer, bearbejde information og flytte viden til langtidshukommelsen. Sweller beskriver hvordan mennesker har en begrænset mængde *arbejdshukommelse*, og for meget belastning vil høre til dårlig indlæring og negative oplevelser af materialet (Sweller, Ayres og Kalyuga 2011). Teorien er særligt relevant indenfor design af undervisning, da den har til hensigt at undersøtte formuleringen af effektivt undervsiningsmateriale (Sweller, Ayres og Kalyuga 2011).

Vores CS50 forløb kan ligeledes beskrives ud fra CLT. Her diskuterede vi den indbyggede load i.e kompleksitet af materialet. Vi fandt programmering som en kompleks opgave. Vi skulle tilegne os færdigheder i syntaks, semantik, en lang række teknikker og ikke mindst en masse nye begreber indenfor Computer science. I vores diskussioner af støjrelaterede load argumenterede vi dog for, at CS50 er et velstruktureret forløb hvor materialet præsenteres på en veltænkt måde. Vi oplevede at CS50 kurset var renset for mange unødvendige belastninger, og koncepterne blev introduceret løbende i gennem øvelserne. Til sidst diskuterede vi den læringsbaserede load, hvor der var rig mulighed

for at engagere sig i hands-on kodning og projekter der aktivt opfordrede til tilegnelsen af nye koncepter. Her referer vi tilbage til beskrivelsen af CS50-øvelserne, hvor de studerende får mulighed for at løse eksempler på problemer i den virkelige verden. Ifølge Sweller er dette en effektiv måde at optimere den *Germane belastning* (Sweller, Ayres og Kalyuga 2011).

4.2 Problemformulering

Kurset CS50: Introduction to Computer Science er kendt for sin omfattende introduktion til de grundlæggende koncepter inden for datalogi samt udvikling af software og algoritmisk tænkning. Vi oplevede en række udfordringer forklaret af begrebet *støjrelateret belastning* i Swellers Kognitive Belastningsteori. Vi vil med udgangspunkt i vores egne oplevelser søge at håndtere de udfordringer vi selv har oplevet under forløbet.

I gennem projektet opstiller vi en række features der søger at emulere principper indeholdt i Paperts konstruktionisme. Her vil vi forsøge at anvende en kombination af tekstuelle og grafiske repræsentationer af kode til at støtte den studerende i gennem et CS50 forløb. Vores indledende problemformulering lød således *Hvordan kan et digitalt design støtte indlæringen af Computational thinking i et CS50 kursus?*

I gennem processen afgrænser vi yderligere problemfeltet til at fokusere på indlæringen af fundamentale kompetencer i computational thinking og programmeringsbegreberne indeholdt i specifikke øvelser. Derudover definerer vi to grundlæggende features, som vi mener er bærende i vores løsning. Disse omhandler forskellige visualiseringsstrategier, men er grundlæggende en kombination af grafiske og tekstuelle repræsentation af kode. På baggrund af dette har vi formuleret problemstillingen som følgende:

"*Kan en kombination af grafiske og tekstuelle repræsentationer, i et introduktionskursus som CS50, støtte indlæringen på to måder: Computational thinking og Programmeringsbegreber?*"

Visuel programmering

Dette kapitel påbegyndes med undersøgelsen af problemet "Hvordan kan et digitalt design støtte indlæringen af Computational Thinking". Vi foreslår igennem kapitlet en række Capabilities på koncept niveau, designet til at løse en række Manifestationer vi havde identificeret på baggrund af vores egne opleveler. Her anvendte vi en række teorier om læring til at underbygge og guide. Ved brug af Eric Ries MVP, testede vi to varianter af disse capabilities på yngre studerende. Vi foretog ligeledes brugerundersøgelser i form af et spørgeskema af 21 studerende, og interviews af 6 studerende. På baggrund af den indsamlede empiri, opstillede vi en række Manifestationer vi mente dækkede over de udfordringer de studerende oplevede i forbindelse med et CS50-forløb. Herefter diskuterede vi vores foreslåede bidrag, og kom med nye idéer til hvordan de kunne ændres til at håndtere de studerendes udfordringer. Til slut i kapitlet anvendte vi Essences Problem-Scenarios til at belyse alternativer til problemet. Det guidede os til beslutninger vedrørende omfanget og konteksten af vores bidrag. Kapitlet med vores endelige problemformulering i dette projekt.

5.1 Design aktiviteter

På baggrund af de *Leverage points* vi har præsenteret i kapitel *Problemfelt* og det nylige skift til CS50-forløb begyndte vi at foreslå *Capabilities* med udgangspunkt i vores egne erfaringer. I vores projekt betød det i praksis, at vi benyttede os selv som *personaer* hvorefter vi indledte en række design aktiviteter med henblik på at generere forskellige løsninger. Som noget nyt for os, blev vores tidligere beskrevne *leverage points* inddraget i rangering og sortering af disse idéer, med henblik på at finde en løsning vi havde ressourcerne og kompetencerne til at udvikle. Dette afsnit beskriver kort hvilke design aktiviteter vi engagerede os i. Vi anvendte en række designaktiviteter til at generere forskellige idéer til at foreslå *Capabilities*. Nedenstående afsnit opsummerer de anvendte teknikker. Aktiviterne er præsenteret ud fra typiske faser i User-centered design (Nielsen 1994), men her skal det nævnes at de forskellige aktiviteter foregik i forskellige rækkefølger igennem processen. Alle disse teknikker er anvendt forskellige tidspunkter i processen, og flere af aktiviteterne er gentagne aktiviteter.

5.1.1 Idégenerering

I idégenereringsfasen anvendte vi brainstorming og Crazy8 til at generere så mange idéer som muligt. Vores brainstorming sessioner var 10-15 minutters varighed, hvor vi i fællesskab skrev idéer på et whiteboard og blev inspireret af dem til at skrive nye. Crazy8 er en hurtig skitseøvelse til idégenerering. Hvert teammedlem får 8 minutter til at tegne 8 hurtige sketches, hvorefter runden gentages. Vi gentog de ovenstående øvelser indtil vi ikke længere kunne producere nyt (Design Sprints 2023).

5.1.2 Sortering

For at skabe et overblik over mange usammenhængende idéer anvendte vi *Octopus sorting*. En af gangen fik hver deltager lov til at rykke rundt på post-it notes, for at sætte idéer der mindede om hinanden sammen. Efter 30 sekunder roterede vi, hvorefter en ny deltager gjorde det samme. Da ingen deltagere til sidst kunne rykke på idéerne, stoppede vi aktiviteten og begyndte at diskutere sammenhænge imellem idéerne (This is service design doing 2022).

5.1.3 Sketching

Vi anvendte ikke specifikke sketching teknikker, men tillod i stedet alle deltagerne 20 minutter til at tegne og sammensætte idéer som de ville. Efter hver runde blev sketches præsenteret, hvorefter runderne blev gentaget (Buxton 2010).

5.1.4 Wireframing og prototype

I dette projekt anvende vi de eksisterende produkter *Cake-Core* (cra16 2016) og *Flowgorithm* som prototyper. Dette beskrives i senere afsnit (Flowgorithm 2024). Til wireframing af vores design anvendte vi Figma, et digitalt designværktøj, der gør det muligt at lave både wireframes og interaktive prototyper effektivt (Figma 2024).

5.2 Capabilities

Som følge af ovenstående fokuserede vi på at foreslå indledende løsninger til at håndtere det vi mente var karakteriseret som *indbygget load* ud fra Swellers teori om *Cognitive Load*. De nedenstående afsnit beskriver vores idéer til indledende *capabilities* på koncept niveau. Med dette menes, at en feature som *Visuel trinvis eksekvering* reelt set dækker over en lang række af små idéer, der kan beskrives i ét koncept.

5.2.1 Visuel trinvis eksekvering

Et tekst-baseret output i en terminal er ikke altid meningsgivende. Særligt i øvelser hvor nye koncepter som *Structs*, *Pointers* eller *Recursion* introduceres, kan det være svært at følge med. Vi oplevede flere gange at gå i stå i vores øvelser på grund af semantiske fejl der var svære at lokalisere. Det kunne eksempelvis være sorteringsalgoritmer der ikke synes at sortere på den korrekte måde, men *hvilken* måde det gik galt på kunne være udfordrende at finde. Vores løsning var ofte at plastre vores program til med *print*-statements, der printer en besked i terminalen som den eksekveres. CS50-Sandbox, og andre IDEs, har ofte en *debugger* hvor koden mere eller mindre trinvist kan inspiceres. Vores erfaring med denne er dog, at den er udfordrende at bruge som nybegynder. Vi så en mulighed for i stedet at forsøge at give den nye studerende bedre indsigt i hvad der foregik inde i computere, mens programmet eksekveres. Ved at visualisere processen på en anden måde, så vi mulighed for at støtte den studerende i indlæringen af nye begreber og koncepter der ligger langt fra hvordan de tænker.

Paperts konstruktionisme understreger vigtigheden af at understøtte feedback og iteration (Papert 1980). Her argumenterede vi for, at højere gennemsigtighed i eksekveringen af et program ville tillade brugeren at opnå reel feedback på semantiske fejl, der ellers kunne være svære at identificere ved almindelig eksekvering.

Dual Coding Theory af Allan Paivio, der deler kognitiv behandling op i to separate kanaler: en visuel kanal til billedinformation og en verbal kanal til tekst og lyd (Clark og Paivio 1991). I hver kanal *kodes* information forskelligt, og mennesker lærer bedst når materialet således præsenteres på begge måder (Clark og Paivio 1991). Et tekst-baseret output ville svare til den verbale kanal. Her argumenterede vi for, at tilføjelsen af den visuelle kanal i form af illustrationer, diagrammer eller animationer, ville øge den studerendes indlæring af koncepter.

5.2.2 Visualiseringsstrategier

I diskussionen af vores egne oplevelser bemærkede vi, at vores ønsker til en løsning var forskellige. Her diskuterede vi blandt andet hvordan vi hver især havde fundet vores egne strategier til at gennemføre kurset og eksamen. Med dette argumenterede vi for, at der i vores tilfælde ikke ville være en "one-size fits all"løsning, da vi tilgik læring på forskellige måder. Vi havde dog alle et ønske om at kunne tilgå vores program på et højere abstraktionsniveau. Dette ønske stammer fra den indledende uge af CS50, hvor *Scratch* anvendes som en blød overgang til programmering. Vores egne oplevelser af dette var positivt, og en stor del af de studerende på vores årgang havde ligeledes lavet velfungerende, små programmer. Her undrede vi os over, at denne mulighed for at tilgå programmet på et højere abstraktionsniveau ikke var en mulighed i CS50-øvelserne. Vores tanke var, at flere studerende på vores årgang havde positive oplevelser den første uge af forløbet, men begyndte at møde udfordringer da øvelserne blev udelukkende tekst-baseret. Her undrede vi os over om det ikke ville være fordelagtigt, at visuelle programmeringssprog, eller lignende redskaber, kunne anvendes i en mere

direkte forbindelse med indlæring af et tekst-baseret kodesprog.

Visualiseringsstrategier tilbyder den studerende forskellige muligheder at tilgå deres program på et højere abstraktionsniveau. Internt i teamet havde vi forskellige ønsker til disse strategier. Eksempler på det kunne være, at programmets overordnede *design* bedre kunne præsenteres for at inspicere den overordnede struktur. Dette kunne eksempelvis være ved brug af Flowchart eller diagrambaserede sprog. Andre i teamet ønskede at fokus skulle være tættere på design af funktioner og rutiner, ved brug af eksempelvis visuelle programmeringssprog som *Blockly*.

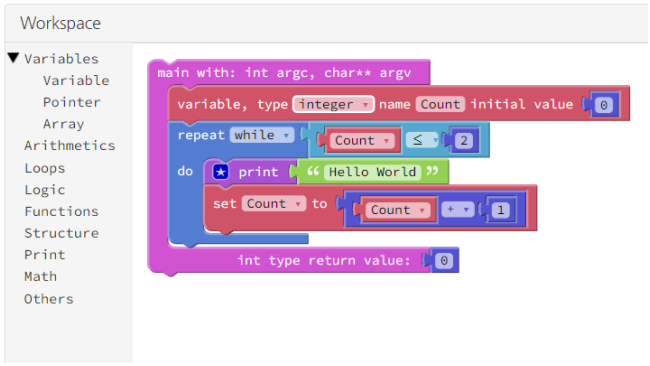
Papert beskriver, at mennesker lærer bedst når de manipulerer fysiske objekter og modtager øjeblikkelig, fysisk feedback (Papert 1980). *Scratch* er designet ud fra Paperts konstruktionisme (Resnick et al. 2009). Det er selvfølgelig ikke ideelt at erstatte det tekst-baserede kodesprog med et visuelt programmeringssprog, når formålet med kurset til dels er at tilegne sig færdigheder i tekst-baserede kodesprog. Vi argumenterede dog for, at værktøjer som *Scratch* også kunne benyttes til at tilegne sig færdigheder i eksempelvis syntaks ved at designe det på den rigtige måde. På den måde anså vi *Scratch* eller andre visuelle programmeringssprog, som værende værktøjer til at lette den *intrinsiske belastning* beskrevet i CLT. Med det menes, at formålet stadig skal være at lære tekst-baserede kodesprog, men det kunne give en værdi at de studerende kunne benytte sig af værktøjer på et højere abstraktionsniveau til at lære de færdigheder.

Scratch kunne dog også gå hen og blive unødvendigt, når de studerende havde tilegnet sig gode færdigheder i syntaks. Et diagrambaseret sprog kunne være særligt anvendeligt i senere øvelser, hvor øvelserne eksempelvis bliver mere omfattende og strukturen deraf mere kompliceret. Vi anså derfor de forskellige visualiseringsstrategier som *midlertidige løsninger*, der skulle bruges i situationer hvor den studerende havde behovet. Her var vi inspirerede af konceptet om *Scaffolding*, hvor støtten gradvist fjernes i takt med at studerende tilegner sig de nødvendige kompetencer (Wood, Bruner og Ross 1976). Denne tankegang er baseret på Vygotskys *Zone for nærmeste udvikling*, der i en undervisningskontekst karakteriserer handlinger som et menneske ikke endnu kan klare på egen hånd, men kan udføre ved brug af støtte fra værktøjer, underviser eller lignende (Wood, Bruner og Ross 1976). Her anså vi altså forskellige visualiseringsstrategier som støtte under forskellige steder i CS50-forløbet, hvor nogle af disse strategier måske ville blive unødvendige efter kort tid, og andre først blive anvendelige senere i forløbet.

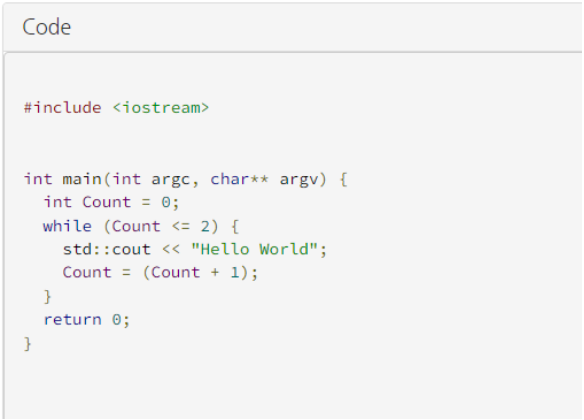
Det *visuelle* aspekt bygger igen på *Dual Coding Theory* af Paivio. Her anser vi eksempelvis *Scratch* som værende et veldesignet værktøj ud fra Paperts konstruktionisme, der ligeledes præsenterer information i både verbale- og visuelle kanal ud fra Paivios princip om dobbeltkodning.

5.3 Cake-Core og Flowgorithm som MVP

Vi testede to forskellige visualiseringsstrategier, der allerede eksisterer ved brug af et Minimum Viable Product som beskrevet af Ries 2011. Det handler om at udvikle en version af et nyt produkt, som indeholder den mindst mulige mængde af funktioner, der stadig gør det muligt at indsamle ny læring om brugeren baseret på deres adfærd og feedback (Ries 2011). Til dette formål var vi inspirerede af *Competetive Usability Evaluations* som beskrevet af Nielsen 1994. Her testes en konkurrents produkt for specifikke features, effektivitet eller lignende. Vi anså vores primære bidrag på dette tidspunkt som værende muligheden for at tilgå og manipulere sit program på et højere abstraktionsniveau. Vi undersøgte i hvilket omfang det eksisterede, og fandt de populære løsninger som *Scratch* og *Blockly*. Derudover fandt vi en række eksisterende, Open source værktøjer der løser meget specifikke problemer. Et af dem var Cake-Core, et *High-Level programming environment* der tillader at skabe C++ kode ved brug af *Blockly* (cra16 2016). Eftersom vi nu var orienteret mod at studerende kunne tilgå deres CS50 øvelser, som primært er i C, så vi muligheden for at benytte cake-core til at afprøve. Velvidende at C og C++ ikke er det samme, var funktionaliteten vi ønskede at teste, altså direkte konvertering fra Blockly til kode, stadig til stede.



Figur 5.1: Cake-cores program skrevet i Blockly, der printer "Hello World"tre gange (cra16 2016)



Figur 5.2: Cake-cores konvetering fra Blockly to C++ kode (cra16 2016)

I dette deltog 4 studerende fra DAD7, årgangen under os. De fik til opgave at løse den første øvelse i CS50 (Mario.c), hvorefter de deltog i et opfølgende interview. De blev ligeledes informeret om, at koden ville blive i C++, og ikke C som de kender fra Mario opgaven. Vi forklarede dem kort om de forskelle de vil støde på, så de var forberedte på, at outputtet så en smule anderledes ud. Vi havde ikke forinden opstillet hypoteser, og målte kun på tiden brugt i alt. I de efterfølgende interviews spurgte vi ind til konceptet, og om hvorvidt de så det som et potentielt værktøj. De opfølgende interviews var korte, og vi så ikke anledning til at foretage yderligere analyse på baggrund af den indsamlede data. Alle 4 studerende fandt konceptet tiltalende og brugbart. Dog mente alle 4 studerende ligeledes, at

Blockly blokkene var blevet mere forvirrende efter Cake-core havde forsøgt at lave deres egne. Vi tog det med videre, at *Blockly* allerede har et velafprøvet og gennemtænkt design af deres blokke.

På samme vis som ved Cake-core, afprøvede vi *Flowgorithm*, et diagram-baseret sprog (Se figur 2.4 for eksempel på Flowgorithm). *Flowgorithm* tillader brugeren at programmere i flowchart-lignende strukturer, og konvertere outputtet til en lang række tekst-baserede sprog. Formålet med denne test var således det samme som Cake-core bare med en anden type af visuelt programmeringssprog. Vi søgte ikke at sammenligne de to, men blot at få en fornemmelse for om de studerende kunne være interesserede i et sådan værktøj. På samme vis som ved Cake-core fik brugerne til opgave at løse den første CS50 opgave (Mario.c).

Alle de fire studerende fandt *Flowgorithm* meget brugbart. Vi observerede dog en ændring i de studerendes måde at tilgå deres program på. Hvor de studerende i Cake-core skrev kode både som tekst og som *blockly* blokke, så brugte de udelukkende *Flowgorithm* i den anden test. Her abstraherede de studerende altså overvejende fra det tekst-baserede kodesprog.

5.4 Manifestations

Efter at have påbegyndt forskellige udkast til forskellige *capabilities*, igangsatte vi en brugerundersøgelse for at lære mere om hvilke af vores bidrag der var særligt (eller overhovedet) relevante. Det primære formål var her at spejle vores egne erfaringer i de nye studerendes, for derefter at påbegynde implementeringen. I det følgende afsnit redegøres der for henholdsvis vores spørgeskema og vores interviews.

5.4.1 Spørgeskema af 21 studerende på DAD7

Vi formulerede et simpelt spørgeskema hvor vi bad de studerende om at beskrive de fem største udfordringer de havde oplevet på CS50 i forbindelse med øvelserne. De studerende skulle ikke rangere deres svar, men blev bedt om kun at angive udfordringer de mente havde været en væsentlig begrænsning for deres læring. Såfremt de studerende ikke kunne finde 5, blev de instrueret i at lade felterne være blanke. I alt modtog vi 21 besvarelser der kan inspiceres i nedenstående tabel 5.1. De studerendes svar er blevet opsummeret i kategorier ved brug af Octopus sorting - kategorierne eksisterer altså ikke på forhånd, men er genereret i den efterfølgende analyse. Hvis ikke et svar har passet meningsfuldt ind i en større kategori, har det fået sin egen. De studerende har ligeledes navngivet deres udfordringer selv i besvarelserne.

Alle 21 deltagere er studerende ved årgangen under os, på Digitalisering og Applikationsudvikling. På det tidspunkt spørgeskemaet blev udfyldt, var det omtrent to måneder siden de havde afsluttet CS50. De havde ikke påbegyndt ny undervisning i programmering.

Formålet med dette spørgeskema var ikke at afdække alle de studerendes behov og frustrationer

Kategori efter Octopus sorting	**Frekvens (antal gange nævnt)**	**Forklaring**
Syntaksfejl	16	Fejl der opstår som følge af forkert syntaks, og ikke tillader brugeren at eksekvere programmet. Fejlsøgningsprocesser i forbindelse med dette.
Feedback og gennemsigtighed	15	Forvirring og fejl der opstår når programmet virker, men ikke virker efter hensigten. Udfordringer med at lokalisere disse fejl.
Tekniske udfordringer	14	Fejl og udfordringer relateret til den tekniske del af deres anvendte platform.
Komplekse begreber	18	En lang række af begreber de studerende havde udfordringer med. Størstedelen var *Pointers og memory management*.
Tidspres	9	Oplevelser af manglende tid i forbindelse med øvelserne. Særligt som følge af sideløbende kurser.
Kodepraksis	3	Udfordringer med basale kodepraksisser og deres anvendelse.

Tabel 5.1: Oversigt over hyppige fejl og udfordringer blandt studerende

til punkt og prikke. Formålet var i stedet at forsøge at identificere de mest kritiske udfordringer, da vi er begrænsede i vores ressourcer. Vi interviewede herefter 6 studerende med udgangspunkt i ovenstående kategorier for at uddybe ovenstående kategorier.

5.4.2 Interviews af 6 studerende på DAD7

vi foretog i alt 6 interviews af de studerende, hvor vi brugte spørgeskema-kategorierne til at guide os igennem interviewet. Det var de samme studerende der havde udfyldt spørgeskemaet. Undervejs i interviewet spurgte vi særligt ind til de studerendes største udfordringer og modtog uddybede svar. Til alle interviews sad et medlem af vores team og udfærdigede noter, der løbende kodede og organiserede de forskellige temaer fra interview til interview. Da vi genbesøgte datamaterialet, var vi tilfredse med den læring vi havde fået i disse sammensatte noter, og valgte derfor at rykke videre. I analysen er de nedenstående kategorier (Manifestationer) altså prædefineret, hvor kategorierne stammer fra spørgeskema-undersøgelsen. Her er det ligeledes 6 studerende der deltager i interviews, der

også har udfyldt spørgeskema.

Manifestation: Syntaksfejl

De studerende forklarede flere situationer med syntaksfejl og fejlsøgning, hvor det begrænsede deres læring. Alle deltagerne i interviews forklarede, at de kunne bruge store mængder tid på at identificere småfejl, der fjernede deres fokus fra programmets overordnede design. Som uddybning forklarede 4 af de studerende, at det oftest var fejlbeskederne var de forskellige anvendte udviklingsmiljøer der var forvirrende. Da der blev spurgt hvorvidt denne fejlsøgningsprocess ikke også kunne være lærerig, svarede 4 af de studerende at de fandt mængden af syntaksfejl, samtidigt med andre udfordringer, som værende for "...voldsomme og frustrerende". De studerende forklarede ligeledes, at de alle havde anvendt teknologier som ChatGPT til automatisk at færdiggøre eller rette deres kode. Her fandt 3 af de studerende at det kunne føles som et nederlag, særligt med tanke på at øvelserne stiger i sværhedsgrad. Alle de interviewede studerende havde haft både gode og dårlige oplevelser med fejlsøgning. Her forklarede en studerende at det var en "succesoplevelse når det lykkes i første hug, men meget demotiverende når man ingen anelse havde, om hvad man skulle gøre".

På baggrund af vores egne erfaringer kan vi spejle os selv i de studerendes oplevelser. En nævneværdig forskel var dog, at ChatGPT kun lige var udkommet da vi tog CS50, og derfor ikke var ligeså udbredt. De studerende mente at næsten alle på den ene eller anden vis havde benyttet sig af ChatGPT til at skrive færdige stykker kode og/eller finde fejlene.

Manifestation: Tekniske udfordringer

De studerende forklarede om en lang række brugerfejl, problemer med deres anvendte udviklingsmiljø eller på anden vis problemer med at kompilere og eksekvere programmet. Eksempler på dette var at glemme at importere de nødvendige biblioteker, navigere og give de korrekte instrukser i terminalen eller få opsat projekterne korrekt ved brug af eksempelvis Flask og Python, som der oftest anvendes i det afsluttende projekt. Flere af de interviewede studerende forklarede ligeledes, at de i løbet af kurset ikke modtog introduktion til teknologier som Git (versionsstyring), Visual Studio (udviklingsmiljø) eller Flask (Web-framework i Python). Her oplevede næsten halvdelen af de interviewede studerende, at deres læringsudbytte blev væsentligt forringet som følge af tekniske udfordringer. Dog fandt de studerende først dette som værende en væsentlig udfordring, når de skulle skifte fra de isolerede CS50 øvelser til at udvikle hele applikationer. Alle de interviewede studerende forklarede, at det mest frustrerende havde været at samarbejde om kode i de samme programmer. Her frembragte de studerende en lang række eksempler på problemer der opstod, når de forsøgte at anvende Git til versionsstyring. Kun 1 ud af de interviewede studerende havde oplevet samarbejde som værende givende og gnidningsfrit.

Vi kunne nikke genkendende til en lang række af de studerendes oplevelser. Hvor vi ikke havde

oplevet tekniske udfordringer som en væsentlig udfordring i de indledende øvelser, så oplevede vi skiftet fra øvelser til projekt som værende meget frustrerende. Vi forklarede det ved at CS50 øvelserne laves i et integreret udviklingsmiljø, hvor en stor del af de tekniske udfordringer er klaret på forhånd. Vi diskuterede et eksempel hvor man i CS50 sandbox kunne begynde Python kodning ved bare at oprette en fil kaldet test.py, men man i en anden IDE som Visual Code skulle downloade og installere en packagehandler, downloade og installere en SDK, oprette et python-miljø og tænde det i sit projekt ved brug af terminalen, og have det hele til at spille sammen.

Manifestation: Feedback og gennemsigtighed

Feedback og gennemsigtighed er hos de studerende relateret til den semantiske del af deres programmering. De studerende forklarede, at elementer som datastrukturer og typer, sorteringsalgoritmer, hukommelseshåndtering eller kontrolstrukturer i programflowet var svære at forstå og frustrerende at arbejde med. Her uddybede tre af de studerende, at de ikke havde færdiggjort flere af øvelserne i CS50 på grund af semantiske fejl. De studerende gav eksempler på, at de ofte skulle "lære, hvad et hashtable eller en linked list er, mens de prøvede at få det til at fungere i koden."Dette betyder, at de studerendes forståelse for eksempelvis en linked list ofte var afhængig af, om de havde fået programmet til at virke efter hensigten. De studerende udtrykte, at de ofte manglede en form for gennemsigtighed i forhold til, hvad der skete trinvist i deres program. Det betød, at det var frustrerende for dem at identificere semantiske fejl i eksempelvis deres sorteringsalgoritme, når de ikke kunne følge med trin for trin. To af de interviewede studerende forklarede ligeledes, at ofte havde svært ved at finde ud af hvordan de skulle fejlsøge semantikken i deres program. Med dette menes, at de studerende ikke havde kendskab til specifikke teknikker til hvordan man eksempelvis "sikrede sig at en variabel blev opdateret med en ny værdi som den skulle". Da de studerende blev adspurgt om de anvendte teknikker som at printe forskellige trin ud i terminalen eller benytte sig af debuggeren, svarede 4 ud af de studerende at det havde de ikke rigtig benyttet sig af på kurset, men havde samlet det op efterfølgende.

Vores egne erfaringer minder meget om de studerendes. Særligt i de senere øvelser som DNA, hvor der skal designes et program til at matche DNA strenge, eller Speller, hvor der skal designes et program der håndterer stavekontrol. Da koncepter som *Pointers* og *Memory management* blev introduceret, begyndte mange af de studerende på vores årgang at falde bagefter. De studerende oplevede det samme, hvor de forklarede at de i påbegyndelsen af øvelsen stadig ikke havde forstået hvad *Pointers* var, og mange ventede med øvelsen til kurset var overstået og der skulle læses op til eksamen.

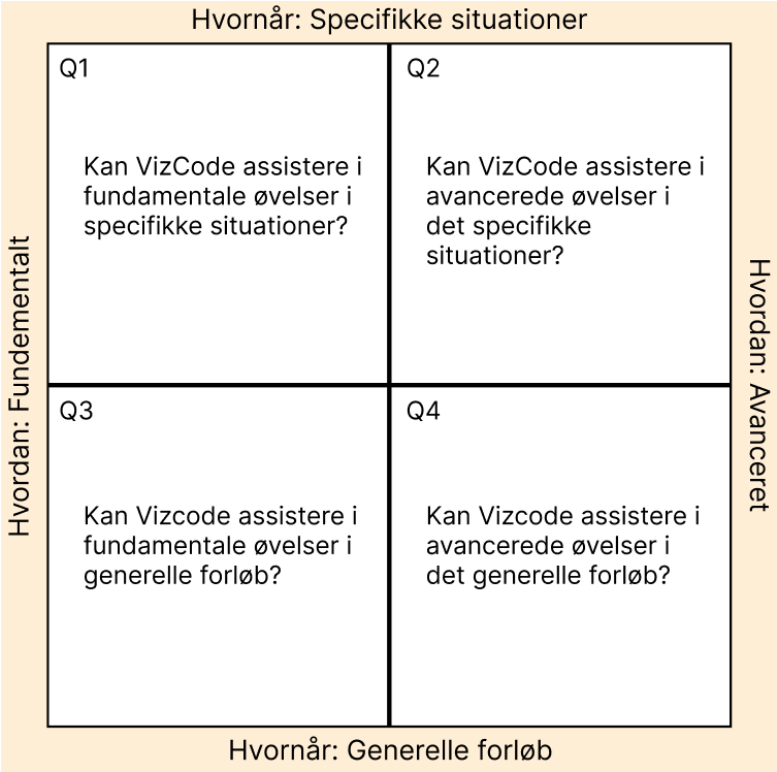
5.4.3 Problem Scenario

Efter at have foretaget tests af MVPs og brugerundersøgelser, påbegyndte vi en beslutningsprocess hvori vi besluttede os for hvilke Manifestationer vi ville håndtere, og hvilke af vores capabilities vi ville bruge til at håndtere dem. Til dette undersøgte vi muligheden for at anvende de forskellige scenarios fra Essence. På dette tidspunkt havde vi identificeret en lang række små løsninger vi mente kunne bringe værdi på den ene eller anden måde. Disse er tidligere blevet opsummeret i de to koncepter *Visualiseringsstrategier* og *Visuel, trinvis eksekvering*. Med begrænset tid og ressourcer, og en vision om at implementere et konkret bidrag, fandt vi det vigtigt at være skarpe i vores valg om hvilke Manifestationer vi ønskede at håndtere. For at kunne gøre dette, anvendte vi Problem Scenarios i Essence til bedre at skabe en forståelse for de forskellige varianter af problemet.

Som tidligere beskrevet bygger Problem Scenarios på 5W2H. Vi beskrev den vertikale akse med *Hvornår*. Her søgte vi særligt en afklaring i *hvornår* problemet manifesterer sig eller ændrer karakter. Den horisontale akse brugte vi *Hvordan* skaber problemet udfordringer i.e påvirker det indirekte, direkte, som en netværkseffekt eller en side-effekt. Her fortolkede vi dog lidt på det. Vi var inspireret af den tankegang der beskrives af Aaen 2024, hvor de to spørgsmål var dem vi fandt bedst stimulerede vores fantasi og hjalp os til at udforske alternativer til problemet. Her var særligt *Hvordan* vigtig for os, da vi følte vi manglede at undersøge hvad konsekvenserne ved problemet var for den studerende.

På *Hvornår* aksen valgte vi at fokusere på forskellene i hvad vi kaldte for her-og-nu situationer, eller generelle *forløb*. Vi anså dette som værende forskellen i om vi forsøgte at skabe en løsning på specifikke situationer *indenfor* CS50-forløbet i.e en løsning rettet direkte mod indlæring af specifikke begreber eller situationer. Det kunne være øvelsen 'Speller', hvor *Pointers og memory management* første gang introduceres. Eller om vores løsning skulle søge at håndtere CS50 som et helt forløb, hvor den studerende måske skal bruge andre værktøjer til at understøtte læring over tid.

På *Hvordan* aksen valgte vi at fokusere på forskellene på *Fundamentalt* og *Avanceret*. Her forsøgte vi at tackle forskellene på hvorvidt vi ville tackle problemet på et introduktions- eller videregående niveau. Vi anså dette som værende en væsentlig overvejelse på baggrund af vores egne erfaringer og brugerundersøgelserne. Begrundelsen for dette var det store spring de studerende oplevede fra CS50-øvelser, der forsøgte at facilitere nogle *fundamentale* forståelser versus det afsluttende projekt hvor en lang række nye teknologier blev introduceret.



Figur 5.3: Visual programming - Problem scenario

Vi diskuterede scenarierne i de fire kvadranter efterfølgende, særligt ud fra hvilke kompetencer og ressourcer vi besad. I hver kvadrant formulerede vi et spørgsmål på samme vis som eksemplet i *Essence* (Aaen 2024), således vi diskuterede nedenstående spørgsmål. Her navngav vi i øvrigt også vores løsning *VizCode*.

- **Q1**: Kan VizCode assistere i fundamentale øvelser i specifikke situationer?
- **Q2**: Kan VizCode assistere i avancerede øvelser i det specifikke situationer?
- **Q3**: Kan Vizcode assistere i fundamentale øvelser i generelle forløb?
- **Q4**: Kan Vizcode assistere i avancerede øvelser i det generelle forløb?

Vi diskuterede forskellene i mellem specifikke situationer versus generelle forløb. I specifikke situationer kunne problemet manifestere sig ved at en øvelse er svær at gennemføre, og indlæringen af det tilhørende koncept ikke sker. På en tidslinje kunne dette også fortolkes som forskellige uger i forløbet, hvor den studerende havde brug for særlig assistance. Med *Pointers og Memory management* som eksempel, har den studerende en konkret her-og-nu udfordring i at skulle forstå og implementere et nyt, udfordrende koncept som mange af de studerende (inklusive os) havde svært ved. Her

kunne vores *Visuel, trinvis-eksekvering* og *Visualiseringsstrategier* være relevante, da de har til hensigt blandt andet at mindske den *intrinsiske belastning* der opstår i den situation.

I et generelt forløb kunne problemet i stedet manifestere sig ved, at den studerende havde behov for værktøjer til at understøtte læring over tid. Her kunne der eksempelvis opstå behov for at genbesøge tidligere øvelser, eller anden vis blive støttet i at 'opretholde' færdighederne fra øvelserne.

Det primære resultat af disse to overvejelser var, at vi indså vores nuværende fokus havde været orienteret mod 'her-og-nu' situationer. Vores nuværende capabilities kan være relevante i hele CS50-forløbet. Her kunne vores idé om forskellige *Visualiseringsstrategier* eksempelvis være relevant, da de er baseret på princippet om *Scaffolding* hvor hver strategi skal understøtte eleven ud fra deres nuværende kompetencer. Her kunne de forskellige visualiseringsstrategier således være designet med henblik på at tackle forskellige udfordringer, eftersom den studerende tilegnede sig flere og flere færdigheder. Ved at anskue problemet som et *forløb* frem for en *situation*, ved brug af Problem-Scenario, indså vi forskellen i mellem de to. Med dette kom også en forståelse af, at vores design var i større risiko for at blive overflødigt, såfremt det ikke understøttede en vis form for *forløb*.

Vi diskuterede ligeledes forskellene i at assistere med fundamentale øvelser versus avancerede øvelser. Den primære forskel heri var for os, at de isolerede øvelser i CS50 er kendt på forhånd og abstraherer fra en lang række teknologier, mens de avancerede øvelser er med valgfrie teknologier. Med andre ord skulle vores design sandsynligvis inkorporere en lang række håndteringer relateret til forskellige frameworks, versionstyring og/eller programmeringssprog på samme vis som hvad der forventes af en standard IDE som Visual Studio Code. Med udgangspunkt i de ressourcer vi har til rådighed, vurderede vi at det var kompleks en opgave. At designe et værktøj der assisterer med CS50 øvelser er et realistisk mål med de kompetencer vi besidder. At inkludere øvelser af højere kompleksitet som Objekt-orientede sprog eller en lang række teknologier, ville kræve mange flere ressourcer og ikke være noget vi kunne realisere i dette projekt.

Her ender vi med at revidere vores problemformulering der tager afsæt i materialet præsenteret i de foregående afsnit, samt vores afgrænsning i dette afsnit. VizCode skal søge at hjælpe de studerende med specifikke øvelser i CS50. De specifikke øvelser repræsenterer hver et nyt begreb eller koncept indenfor programmering grundet CS50s struktur.

VizCode skal ligeledes søge at støtte indlæringen af grundlæggende færdigheder i Computational Thinking. Vi formulerede problemet på samme vis som i vores Problem Scenarie, og det er som følgende:

"Kan VizCode, i et introduktionskursus som CS50, støtte indlæringen på to måder: (1) Computational thinking og (2) Programmeringsbegreber?"

5.4.4 Beslutningsprocess

Vi anvendte PSC til at konkretisere hvilke Manifestationer vi anså som værende hjemmehørende i de forskellige kvadranter. Nedenstående figur afspejler derfor vores forståelse af problemet på dette tidspunkt i processen.

Vi besluttede os for at vores design skulle henvende sig til de fundamentale, eller indledende øvelser i CS50. Vi besluttede os ligeledes for at fokusere på capabilities til specifikke situationer, mens vi anerkendte vigtigheden af at anskue problemet som et *forløb* over tid.

5.5 Løsningsforslag

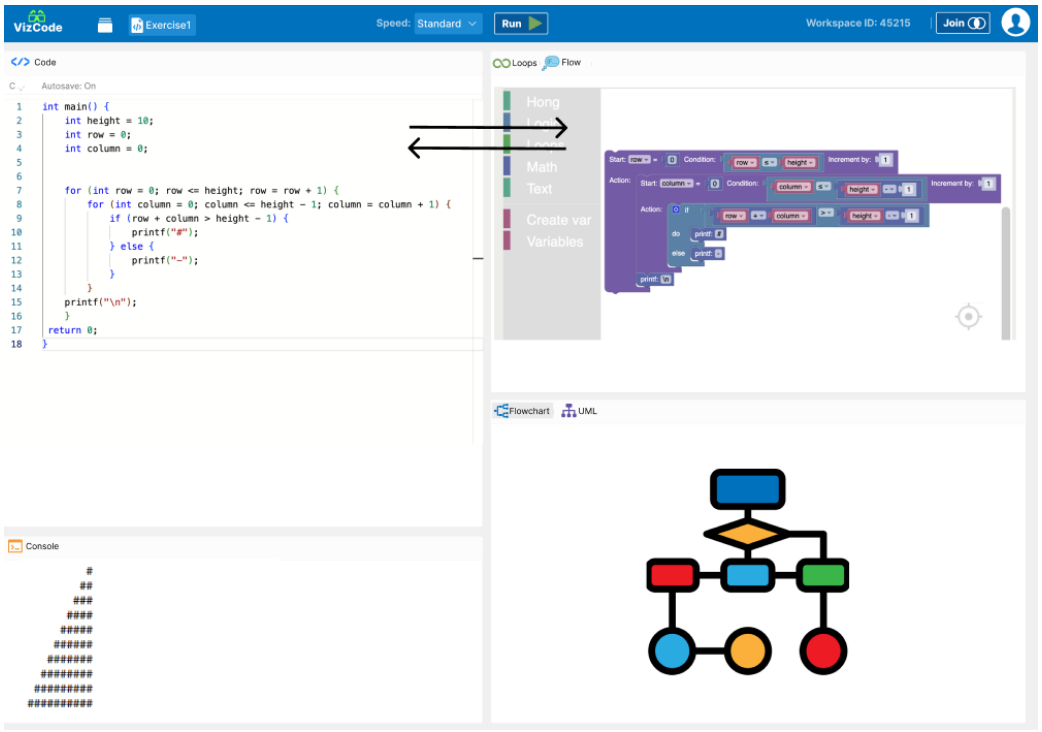
I dette afsnit præsenteres de konkrete Capabilities vi besluttede os for samt begrundelsen for dette. Alle disse figurerer ligeledes i kapitel *Design*, hvor de uddybes og sættes i en kontekst af vores generelle design.

5.5.1 Visualiseringsstrategier

De tidligere beskrevne *Visualiseringsstrategier* skal i vores design tilbydes som håndteringer af *Syntaksfejl* hos de studerende. Ved at benytte visuelle programmeringssprog som *Flowgorithm* eller *Blockly*, kan den studerende tilgå sit program på et højere abstraktionsniveau - i et sprog der ikke tillader brugeren at lave syntaksfejl. Den studerende kan inspicere det visuelle programmeringssprog på en meningsfuld måde i fejlsøgningsprocessen, da syntaksfejl vil være repræsenteret som blokke der mangler, eller ikke er sat sammen, i et sprog som Blockly.

5.5.2 To-vejs synkronisering af visuelle- og tekst-baserede programmeringssprog

Alle de eksisterende værktøjer vi kunne finde, tilbød ligeledes kun en-vejs konvertering i.e Visualisering på baggrund af tekst-baseret kode, eller tekst-baseret kode på baggrund af visuelt programmeringssprog. Vi introducerer derfor en simultan, tovejs konvertering mellem *Blockly* og C for at gøre vores design unikt. Denne teknologi gør det muligt for studerende at udnytte *Blockly*s visuelle syntaksfejlindikatorer, hvor inkompatible kodeblokke ikke kan sammenkobles, og giver dem mulighed for at foretage nødvendige rettelser i koden efter behov. På samme vis kan de studerende foretage ændringer i *Blockly* og se den samme ændring i den tekst-baserede kode. Nedestående figur er vores wireframe. På wireframet fremgår to pile, som blot er til for at visualisere to-vejs synkroniseringen.



Figur 5.4: Wireframe og illustration af to-vejs synkronisering

For at de studerende modtager øjeblikkelig feedback som beskrevet af Papert, skal konverteringen ske med det samme. Vi finder det ligeledes vigtigt at konverteringen sker uden fejl, således vores system ikke introducerer lignende udfordringer eller ændrer på de studerendes originale kode. Således opstillede vi en række betingelser til denne feature, som vi videreudviklede på senere i implementeringsprocessen. I kapitel *Design* er disse repræsenteret i tabel 7.1.

1. Brugeren skal kunne tilgå deres på et højere abstraktionsniveau i form af visuelle programmeringssprog
2. Brugeren skal kunne foretage ændringer i alle repræsentationer af deres program, hvis meningsfuldt
3. To-vejs synkroniseringen skal indeholde så få fejl som muligt
4. Brugeren skal tilbydes visuelle repræsentationer, der matcher brugerens nuværende kompetencer i programmering
5. Synkroniseringen af forskellige repræsentationer skal foregå øjeblikkeligt
6. Synkroniseringen af forskellige repræsentationer skal foregå uden videre handling fra brugeren, således effekten af deres ændringer sker i realtid

5.5.3 Trinvis eksekvering og Visuelt output

Den tidligere beskrevne Visuelle, trinvise eksekvering skal håndtere de studerendes Manifestationer *Feedback og Gennemsigtighed*. I vores design skal der tilbydes en anden form for eksekvering af brugerens program. Denne trinvise eksekvering, hvor hver linje kode eksekveres langsomt efter hinanden, er typisk indeholdt i det der traditionelt kaldes en *Debugger*. I vores løsning skal dette dog foregå på en meget simplere måde, da en debugger typisk indeholder en lang række funktionaliteter. De studerende udtrykte, at de ikke anvendte CS50s debugger (eller andre) da de fandt denne meget kompliceret. Med tanke på at de studerende på samme tid er ved at tilegne sig en lang række af andre færdigheder, vil vi simplificere dette koncept.

Her foreslår vi en simpel valgmulighed om at eksekvere programmet som normalt i.e så hurtigt som muligt, eller langsomt i små trin der kan pauses og genstartes. På den måde argumenterer vi for, at den studerende lettere kan identificere semantiske fejl i eksempelvis en sorteringsalgoritme, ved at følge med i processen. Her skal det være muligt for den studerende at se sit program i forskellige øjebliksbilleder.

En anden håndtering af *Feedback og Gennemsigtighed* er et Visuelt output. Vi vil forsøge at engagere de studerende i at konstruere deres kode og se resultaterne af deres arbejde visuelt, hvilket vi argumenterer for kan hjælpe med at fastslå deres forståelse for abstrakte koncepter. Vi forsøgte at betragte en sorteringsalgoritme som et objekt i tråd med Paperts *Things to think with*, hvor eksempelvis en håndgribelig visualisering af Mario, der placerer mursten, ville være lettere at forstå end et øjeblikkeligt output af en forkert bygget pyramide i øvelse 1. Her lod vi os også inspirere af teamet bag *Scratch*, som benytter sig af lignende principper til at visualisere output (Resnick et al. 2009).

5.5.4 Sandbox og standardløsninger

Vi vil håndtere de studerendes *Tekniske udfordringer*, ved at lade dem arbejde i et kontrolleret Sandboxmiljø, hvor en stor del af de tekniske aspekter er klaret på forhånd. Her vælger vi, som tidligere beskrevet, ikke at fokusere på inklusionen af en lang række teknologier som er krævet i de senere øvelser.

Tekniske udfordringer dækker blandt andet over en række brugerfejl som manglende importering af biblioteker eller at navigere i terminalen. I vores løsning søger vi derfor, ved at fokusere på brugervenlighed, at minimere risikoen for at foretage brugerfejl der kunne skabe frustrerende situationer. Vi kender en del af disse på forhånd, men forventer at disse vil præsentere sig selv løbende i gennem tests af vores design.

5.5.5 Endelige problemformulering

På dette tidspunkt anså vi konceptet med Visualiseringsstrategier og To-vejs synkronisering som vores bærende features. Senere i projektet vil vi danne hypoteser ud fra disse features. På dette tidspunkt i processen formulerede vi vores endelige problemformulering, der tager afsæt i beslutningen om Visualiseringsstrategier og To-vejs synkronisering. Den er formuleret som følgende:

Kan en kombination af grafiske og tekstuelle repræsentationer, i et introduktionskursus som CS50, støtte indlæringen på to måder: Computational thinking og Programmeringsbegreber?"

Samarbejde og pair programming

Efter vi undersøgte både tangible og visual programming, blev det tydeligt, at samarbejde og specifikt pair programming er effektive metoder til at facilitere læring af computational thinking.

I dette kapitel vil vi undersøge to centrale præmisser for pair programming, som har vækket vores interesse for samarbejde og læring inden for programmering og computational thinking (CT).

Den første præmis er remote undervisning. Her er vores undren centreret omkring, hvordan man kan understøtte indlæring af programmering og CT, når deltagerne er adskilte i rum, men sammen i tid. Vi ønsker at undersøge, hvilke værktøjer og metoder der kan muliggøre synkron kodning, realtidsskærmdeling og effektiv kommunikation, der kan fremme kontinuerlig interaktion og feedback mellem studerende.

Den anden præmis er kolokeret undervisning. Vores interesse her ligger i at forstå, hvordan man kan understøtte indlæring af programmering og CT, når deltagerne er i samme rum og tid. Vi vil undersøge strukturerede guider og frameworks for pair programming, der sikrer, at studerende aktivt deltager og skifter roller.

6.1 Samarbejde

Studier viser, at studerende, der engagerer sig i pair programming, ofte udviser højere engagement, dybere forståelse af programmeringskoncepter og en stærkere evne til at anvende disse koncepter i praktiske sammenhænge (McDowell et al. 2006). McDowell et al. 2006 fandt, at studerende, der arbejdede i par, ikke alene opnåede højere karakterer i programmeringskurser, men også rapporterede en større tilfredshed med kursusoplevelsen sammenlignet med dem, der arbejdede alene. Dette understøttes af Funk et al. 2021 og Horn et al. 2009, der begge dokumenterer, hvordan fysiske og visuelle programmeringssprog kan fremme samarbejde. Funk et al. påpeger, at tangible interfaces, såsom fysiske programmeringsblokke, naturligt indbyder til samarbejde mellem elever, idet de skal samarbejde om at manipulere og organisere disse blokke for at løse programmeringsopgaver (Funk et al. 2021). Dette kræver dialog og udveksling af idéer, som er centrale aspekter af pair programming.

Disse indsigter om samarbejde understøtter yderligere, at samarbejde vil være en fordelagtig feature, når det kommer til at lære programmering og computational thinking. I et digitalt læringsmiljø, som vi forsøger at skabe, bliver pair programming især værdifuldt, fordi det muliggør realtidsfeedback og kontinuerlig interaktion mellem de studerende. Dette er ikke kun begrænset til fysisk nærhed; man kan derfor facilitere remote pair programming, hvor elever kan samarbejde og kode sammen fra forskellige geografiske lokationer, som når man laver CS50-opgaverne til næste forelæsning og ikke

nødvendigvis længere befinder sig samme sted. Værktøjer, der understøtter deling af kode, synkron redigering og kommunikation, kan forstærke læringsprocessen og samtidig øge samarbejdet (Resnick et al. 2009). Maloney et al. 2010 forklarer ligeledes, at ved at gøre det muligt at dele projekter, fremmer det et åbent læringsmiljø, hvor feedback og konstruktiv kritik er velkomne fra både medstuderende og lærere.

6.1.1 Brugerundersøgelse

Da vi selv sad med CS50-opgaverne, kommunikerede vi gennem sociale medier som Facebook og Discord, hvor vi ligeledes kunne sparre med hinanden. Det var gavnligt, hvis man følte, man sad fast med en opgave. Netop dette perspektiv kunne de studerende i vores brugerundersøgelse også genkende. Flere af de studerende kunne ligeledes få en følelse af at sidde fast, hvis man sad alene med en opgave og ikke vidste, hvordan man skulle løse den og derved mistede motivationen. Nogle af de studerende fortalte dog, at samarbejdet også kunne have sine ulemper; når der skal arbejdes på semesterprojektet, som er obligatorisk hvert semester hos Aalborg Universitet, kan der være gruppemedlemmer, der "free-rider", altså indgår i projektgruppen uden at lave noget. Dette er dog noget, stort set alle de studerende og os selv havde personlig erfaring med i vores tid som studerende. Der blev dog nævnt, at der specifikt med koden, som den fælles nye færdighed alle skal lære, var en øget tendens til, at folk ikke deltog. Der var dog bred enighed om, at gruppearbejdet i langt de fleste tilfælde har været gavnligt for læringen af programmeringsfærdigheder og CT.

6.1.2 Pair programming som capability og leverage point

Efter vores litteraturstudie og brugerundersøgelse var det derfor tydeligt, at vi ønskede at facilitere samarbejde og mere specifikt 'pair programming' som en feature i vores løsning. Pair programming, oprindeligt udviklet som en del af Extreme Programming (XP) softwareudviklingsframeworket (Larman 2004), er en teknik, hvor to programmører arbejder sammen ved én computer. Én skriver koden ('driver'), mens den anden ('navigator') gennemgår hver linje kode, der skrives, tænker på strategiske forbedringer og mulige fejl. Dette kan dog også faciliteres remote. Denne samarbejdsmetode er blevet bredt adopteret i uddannelses- og oplæringskontekster for dens evne til at forbedre programmerings- og computational thinking-færdigheder (Williams og Kessler 2000).

For at 'pair programming' kan fungere effektivt, er det vigtigt, at undervisere klart definerer rollerne inden for parret, således at opgaver og ansvar er balanceret (Qentelli 2024). En velstruktureret guide til pair programming bør omfatte klare regler for turtagning. En simplificeret redegørelse af, hvordan pair programming foregår: de nye programmører skifter mellem at være 'driver', den person der skriver koden, og 'navigator', den der gennemgår koden og foreslår forbedringer (Qentelli 2024). Dette fordrer, at alle involverede er aktivt engagerede og lærer af processen (Williams og Kessler 2002). Williams og Kessler beskriver, hvordan denne rolleskiftning bidrager til dybere engagement

og forståelse, da det tillader eleverne konstant at være involveret i læringsprocessen, mens de aktivt reflekterer over kode og problemstillinger fra forskellige perspektiver (Williams og Kessler 2002).

Dette fokus på aktiv deltagelse og engagement i læringsprocessen understøttes yderligere af forskningen af Dybå og Arisholm (Dybå og Arisholm 2005). I deres studie "Are Two Heads Better than One? On the Effectiveness of Pair Programming,"konkluderer de, at pair programming ikke kun forbedrer kvaliteten af softwareudvikling, men også øger programmeringseffektiviteten og tilfredsheden blandt udviklere (Dybå og Arisholm 2005). De fremhæver, at den direkte samarbejdsdynamik i pair programming fører til en dybere forståelsesproces og mere kritisk tænkning, da programmørerne kontinuerligt er udfordret til at forklare og argumentere for deres kodevalg over for hinanden.

Studiet viser, at denne metode er særligt effektiv i uddannelsesmæssige sammenhænge, hvor eleverne udvikler en stærkere forståelse for programmeringskoncepter og forbedrer deres problemløsningsevner gennem gensidig læring og feedback (Dybå og Arisholm 2005). Denne indsigt er afgørende for at sikre, at pair programming ikke kun ses som en teknik til kodeproduktion, men som en undervisningsstrategi, der fremmer både faglige og sociale kompetencer.

Udfordringer ved remote pair programming

Som tidligere nævnt er pair programming en samarbejdsmetode, der kan forbedre programmørers færdigheder og effektivitet (Williams og Kessler 2002). Dog opstår der flere udfordringer, når denne metode implementeres remote. Ifølge Robe et al. (Robe et al. 2020) kan den fysiske adskillelse mellem programmører skabe kommunikationsbarrierer, hvilket gør det sværere at tolke nonverbale signaler og sikre effektiv interaktion. Teknologiske problemer som ustabil internetforbindelse og softwarekompatibilitet kan yderligere hindre arbejdsprocessen.

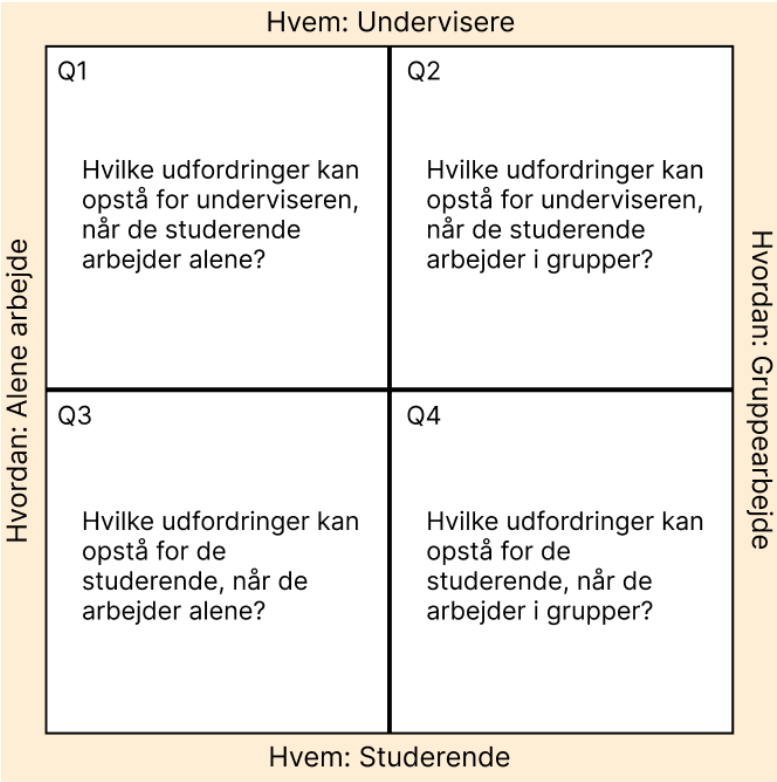
På den anden side viser forskning af Dybå og Arisholm (Dybå og Arisholm 2005), at pair programming, selv når det udføres remote, stadig kan øge kvaliteten af softwareudvikling samt programmørers tilfredshed og effektivitet. De fremhæver, at den direkte samarbejdsdynamik i pair programming fører til en dybere forståelsesproces og mere kritisk tænkning, selvom det kræver en større indsats for at opretholde denne dynamik remote.

For at 'pair programming' skal fungere effektivt i en remote kontekst, anbefales det at anvende robuste samarbejdsværktøjer, der muliggør realtidsskærmdeling og kontinuerlig kommunikation via video og chat (Robe et al. 2020). Derudover bør klare retningslinjer og strukturer etableres for at sikre, at begge parter er klar over deres roller og ansvar, hvilket kan medvirke til at minimere forvirring og maksimere produktiviteten (Williams og Kessler 2002).

Selvom der er betydelige udfordringer ved remote pair programming, viser forskningen, at med de rette værktøjer og tilgange kan mange af disse udfordringer overvindes, hvilket understøtter de samme principper om aktiv deltagelse og engagement i læringsprocessen (Dybå og Arisholm 2005).

6.1.3 Problem Scenario

For bedre at forstå problemer med samarbejde og dets manifestationer, valgte vi at bruge problem scenarios til at belyse kompleksiteten med hjælp af vores viden fra litteratur og vores brugerundersøgelse. Det er her interessant at se på undervisere og studerende som *hvem*, på x-aksen. På y-aksen vil vi undersøge Samarbejde mod alene som et *hvordan*. Akserne er valgt, fordi vi ønsker at undersøge, hvilke problemer der kan manifestere sig ved forskellige arbejdsformer. Eftersom vores løsning er tiltænkt studerende, kan det ligeledes være interessant at belyse problemer, der måtte være fra underviserens side. Ved at opdele på ovenstående måde har vi diskuteret de forskellige manifestationer:



Figur 6.1: Samarbejde - Problem scenario

- **Q1: Undervisere - Arbejder alene:**
  Manifestation: Mangel på indsigt i studerendes forståelsesniveau og individuelle udfordringer.
  Beskrivelse: Når de studerende arbejder alene, kan det for underviseren være vanskeligt at tilpasse undervisningsmaterialet til de studerendes specifikke behov eller identificere, hvem der kæmper med materialet. Dette kan føre til mindre effektiv undervisning og mindre engagerede studerende (McDowell et al. 2006).

- **Q2: Undervisere - Gruppearbejde:**

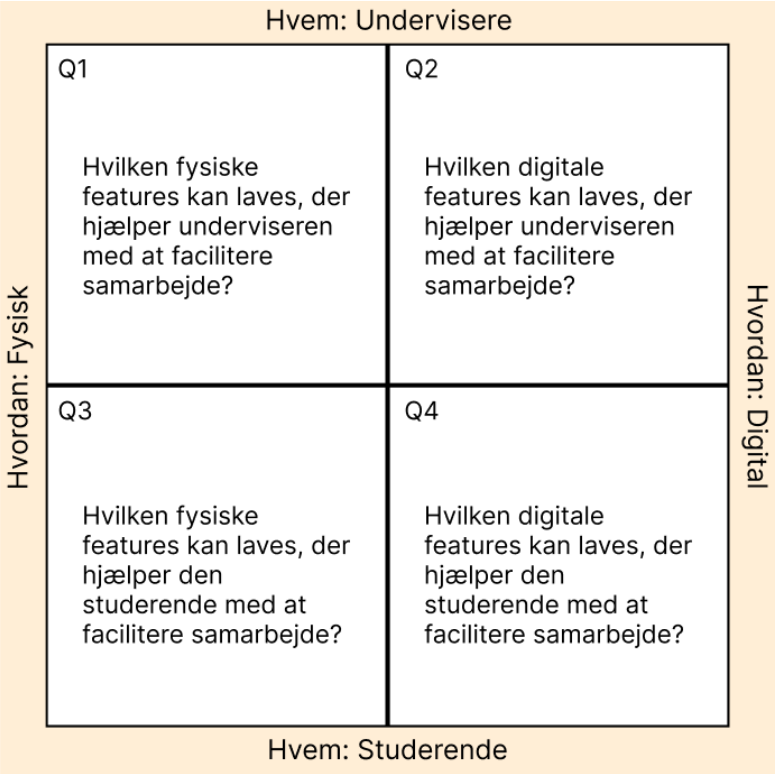
Manifestation: Koordinering og styring af gruppedynamikker.
Beskrivelse: Når undervisere laver gruppearbejde, kan de stå over for udfordringer med at sikre, at alle medlemmer deltager lige meget og derfor kan falde bagud.

- **Q3: Studerende - Arbejder alene:**
  Manifestation: Isolation og manglende støtte.
  Beskrivelse: Studerende, der arbejder alene, kan opleve problemer med motivation og kan føle sig isolerede, især når de støder på udfordrende materiale. Uden medstuderende til at diskutere og reflektere over læringsmaterialet med kan de kæmpe mere med at forstå komplekse koncepter og føle sig mindre engagerede i læreprocessen (McDowell et al. 2006).

- **Q4: Studerende - Gruppearbejde:**
  Manifestation: Ujævn arbejdsfordeling.
  Beskrivelse: Mens gruppearbejde kan fremme samarbejde og dybere læring, kan det også medføre problemer som "free-riding", hvor nogle medlemmer lader andre gøre det meste af arbejdet.

6.1.4 Leverage scenario

For at finde frem til leverage points, valgte vi at gøre brug af leverage scenarios. Ligesom vores problem scenario valgte vi igen at bruge undervisere og studerende, da det er de primære aktører. De udgør altså derfor x-aksen. På y-aksen vil vi her undersøge, hvilke mulige features der kan laves, enten som en fysisk eller digital løsning. Det vil altså sige, hvilke muligheder der vil være ved eksterne features som for eksempel en manual. Ligeledes menes der med digital, inkorporerende features i løsningen der faciliterer samarbejde.



Figur 6.2: Samarbejde - Leverage scenario

- **Q1: Undervisere og Fysisk:**
  Leverage point: Struktureret guide/manual for samarbejde
  Beskrivelse: Udviklingen af en detaljeret guide eller manual, der beskriver hvordan effektivt pair programming og gruppearbejde faciliteres, kan hjælpe undervisere med at forstå og implementere praksissen i undervisningen (Qentelli 2024).

- **Q2: Undervisere og Digital:**
  Leverage point: Feature til at facilitere fjernsamarbejde
  Beskrivelse: I lyset af forskningen af Resnick (Resnick et al. 2009) og Maloney (Maloney et al. 2010), som understreger digital platforms evne til at støtte læringsmiljøer ved at tilbyde realtidsfeedback og kontinuerlig interaktion, kan en dedikeret digital platform hjælpe undervisere med at følge med i og guide fjernsamarbejde mere effektivt. Platformen kan inkludere funktioner designet til at understøtte kommunikation og samarbejde.

- **Q3: Studerende og Fysisk:**
  Leverage point: Samarbejdsfaciliterende læringsrum
  Beskrivelse: På baggrund af brugerundersøgelser, hvor studerende udtrykte, at fysisk samar-

bejde var gavnligt for deres læring og hjalp dem med at lære at programmere, kan en specificeret guide i 'pair programming' give dem en effektiv læring af programmering og computational thinking (Dybå og Arisholm 2005).

- **Q4: Studerende og Digital:**
  Leverage point: Integrerede værktøjer til online samarbejde
  Beskrivelse: Med henblik på Dybå og Arisholms (Dybå og Arisholm 2005) konklusioner om pair programmings effektivitet i at fremme dybere forståelse for programmering og CT, kan integrerede digitale værktøjer støtte disse processer i et remote læringsmiljø. Disse værktøjer kan omfatte kode-delingsfunktionaliteter og synkron redigering for at gøre samarbejde gnidningsfrit.

6.1.5 Opsamling og strategisk valg

Gennem arbejdet med de forskellige scenarier er det blevet klart, at et digitalt læringsmiljø tilbyder væsentlige fordele, især i kontekster, hvor fysisk samvær ikke er muligt eller praktisk. Vores valg af at fokusere på digitale løsninger i både Q2 (Undervisere og Digital) og Q4 (Studerende og Digital) understøttes af både den aktuelle forskning og den feedback, vi har indsamlet (Resnick et al. 2009) (Maloney et al. 2010).

Q2: Undervisere og Digital og **Q4: Studerende og Digital** tilbyder muligheder for at udvide og forbedre læringsoplevelsen gennem remote pair programming. Ved at have muligheden for at kunne arbejde sammen remote, så man ikke føler, man sidder fast, kan forhåbentligt bidrage med at opretholde engagement og motivation hos den studerende (Dybå og Arisholm 2005).

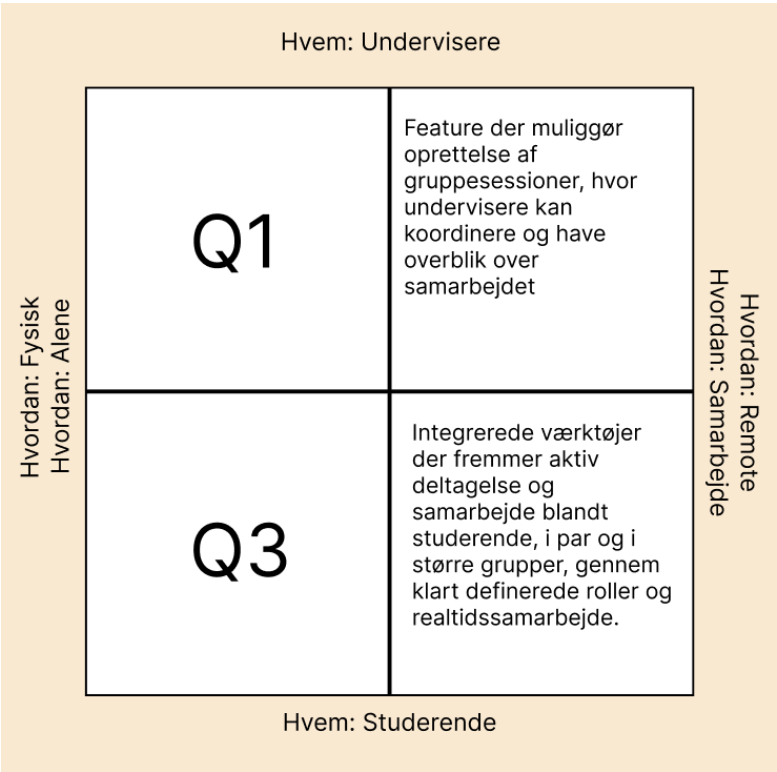
Valget om at nedprioritere manuelle, fysiske løsninger (Q1 og Q3) til fordel for digitale løsninger afspejler også en strategisk beslutning om, at fysiske løsninger kan bruges uagtet, hvordan vores løsning ender, og det vil derfor ikke give vores software en speciel capability. Dog er det vigtigt at anerkende, at Q1 (Undervisere og Fysisk) og Q3 (Studerende og Fysisk) stadig repræsenterer effektive metoder til læring (Funk et al. 2021).

6.1.6 Prospect scenario

Gennem vores undersøgelse af samarbejde har vi identificeret, at effektivt samarbejde spiller en vigtig rolle i læring og udvikling af programmering og computational thinking. For at opsummere ovenstående: vi ved nu, at samarbejdsbaseret læring ikke kun fremmer dybere forståelse af domænet, men også udvikler sociale færdigheder, såsom kommunikation og teamarbejde (McDowell et al. 2006) (Dybå og Arisholm 2005). Dette understøttes af studier som McDowell et al., der viser, at studerende, der engagerer sig i pair programming, ofte opnår bedre resultater og oplever større tilfredshed med læringsprocessen, hvilket er til fordel for både undervisere og studerende (McDowell et al. 2006).

Et leverage point for os er, at vi har mulighed for at implementere en løsning direkte i vores platform. Der findes guides til, hvordan pair programming faciliteres fysisk, og intet forhindrer brugerne i at gøre brug af klassisk fysisk pair programming (Qentelli 2024). Derfor vælger vi at have fokus på Q2 og Q4 fra 'leverage scenario'-modellen 6.2, altså remote samarbejde.

Vi gør derfor brug af et prospect scenario for at udforske og kombinere forskellige manifestationer af problemet med identificerede leverage points og capabilities.



Figur 6.3: Prospect scenario til samarbejde

- **Q2: Undervisere og Remote**
  Løsning: For at imødekomme udfordringen med at sikre effektiv gruppedynamik og koordinering i remote settings, kan platformen tillade undervisere at oprette og administrere interaktive gruppesessioner. Funktioner som realtidsfeedback og muligheden for at opdele studerende i breakout-rooms vil gøre det muligt for underviseren at følge med og deltage direkte i de studerendes arbejdsprocesser.

- **Q4: Studerende og Remote løsninger**
  Løsning: Remote pair programming og gruppearbejde vil adressere nogle af de problemer, som studerende oplever med isolation og ujævn arbejdsfordeling i remote læringsmiljøer. Ved at give

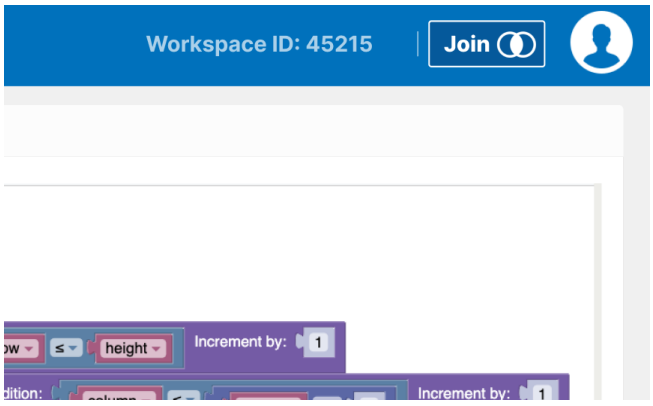
studerende mulighed for at deltage i programmeringssessioner via en "Join"knap og en session-kode, skabes mulighed for remote pair programming. Muligheden for at vælge og skifte roller som 'driver' og 'navigator' samt gruppearbejde med flere deltagere sikrer, at alle studerende er aktivt engagerede og lærer fra hinanden. For at undgå "free-riding", eller at der er gruppemedlemmer, der ikke får lov at være 'driver', kan man gøre rollerne tidsbestemt.

6.1.7 Løsningsforslaget

Løsningsforslaget nedenfor skal ikke antages for et endeligt design. Vi har prioriteret at få implementeret to-vejs synkronisering og Blockly som visualiseringsstrategi, da dette er vores vigtigste capabilities. Samarbejde og pair programming vil derfor være det næste skridt at implementere i vores løsning. Vi har dog gjort os nogle tanker om, hvordan featuren kunne designes, og hvilken rolle den ville have på baggrund af vores scenarios.

Vi ønsker at gøre remote-samarbejde nemt og overskueligt og samtidig fremme brugen af det. Vi vil derfor placere en "Join"knap i toppen af menuen, som ses på nedenstående billede 6.4. Når man trykker på knappen, vil det være muligt at indtaste et session ID, som man kender det fra apps som 'Kahoot' (Kahoot 2024). Udover de synlige elementer på 'Workspace'-siden er der ikke blevet lavet yderligere wireframes til, hvordan de specifikke elementer til pair programming skal se ud, da vi ikke vil få implementeret det inden projektets afslutning.

Efter ens makker til pair programming er tilsluttet ens session, skal der uddelegeres roller og eventuel tid i den givne rolle. Parret vælger en opgave og går i gang med at løse den. En 'pair programming-guide' vil eventuelt blive anbefalet at læse inden man starter, men som nævnt tidligere har vi abstraheret væk fra en konkret guide til pair programming, og det vil derfor antages, at underviseren har formidlet de studerende om processen.



Figur 6.4: Wireframe til samarbejde

En underviser vil ligeledes kunne vælge at lave en gruppesession, hvor alle elever kan tilslutte på samme tid. Til sådan en feature vil man kunne bruge inspiration fra, hvordan Zoom håndterer grupperum (Communications 2024). Her har session-afholderen (underviseren i dette tilfælde) mulighed for at styre gruppefordelingen, følge med i og tilslutte undersessionerne for at hjælpe de studerende.

Design

I de følgende to afsnit præsenteres VizCode, vores web-baserede udviklingsmiljø. I den første del vil hver feature præsenteres som en capability. De vil blive præsenteret sammen med de manifestationer fra vores egne erfaringer og brugerundersøgelser, som de er designet til at håndtere. Her forklares hvilket teoretisk grundlag idéerne er designet ud fra i form af forskellige teorier om menneskelig læring. Dernæst opsummerer vi de features, vi mener udgør vores bidrag til besvarelse af problemformuleringen 7.1. Til slut i dette kapitel vil vores design blive præsenteret. I næste kapitel beskrives de dele, vi nåede at implementere, og hvilke valg vi foretog angående dette.

7.1 Visuel programmering

Vi løser problemstillingen med vores design ved at tillade brugeren at visualisere tekstbaseret kode på forskellige måder, alt efter hvad der giver mening for dem. Vores løsning giver brugere mulighed for at tilgå deres tekstbaserede kode via en række forskellige visualiseringsstrategier. Dette sker på et højere abstraktionsniveau og tilbyder dermed adskillige tilgange til programmering. Gennem disse strategier adresserer vi de centrale udfordringer, som vi har identificeret hos de nye programmører. I dette afsnit vil vi argumentere for vores valg samt beskrive den indsamlede empiri.

7.1.1 Håndtering af Syntaksfejl, Tekniske Udfordringer, og Manglende Feedback

De udfordringer, vi selv oplevede som nye programmører, drejede sig særligt om kompleksiteten af programmering. Her argumenterede vi for, at vores forløb med CS50 bar præg af høj, *indbygget load* ud fra Swellers teori om *Cognitive Load*. Vi undersøgte disse udfordringer blandt 21 studerende ved brug af et kvantitativt spørgeskema med 21 deltagere, samt opfølgende interviews af 6 af deltagerne, der er studerende ved Digitalisering og Applikationsudvikling på Aalborg Universitet. Vi identificerede i vores designprocess tre indledende manifestationer, baseret på de studerendes udfordringer med at gennemføre CS50. Disse manifestationer er *Syntaksfejl*, *Tekniske udfordringer* og *Feedback og gennemsigtighed*. I de følgende afsnit vil hver manifestation blive uddybet, samt præsentation af vores capabilities designet til at håndtere dem.

7.1.2 Overordnede koncept: Visualiseringsstrategier

Dual Coding Theory af Allan Paivio, der deler kognitiv behandling op i to separate kanaler: en visuel kanal til billedinformation og en verbal kanal til tekst og lyd (Clark og Paivio 1991). I hver kanal *kodes* information forskelligt, og mennesker lærer bedst, når materialet således præsenteres på begge måder (Clark og Paivio 1991). Et tekstbaseret output ville svare til den verbale kanal. Her argumenterede vi for, at tilføjelsen af den visuelle kanal i form af illustrationer, diagrammer eller animationer, ville øge den studerendes indlæring af koncepter.

7.1.3 Manifestation: Syntaksfejl

Manifestationen *Syntaksfejl* er hos de studerende en væsentlig årsag til frustration. I programmering refererer syntaks til de regler, der definerer strukturen af kode, herunder hvordan kodeelementer som nøgleord, operatorer og symboler skal arrangeres for at danne gyldige instruktioner, som en computer kan eksekvere. Syntaks varierer mellem programmeringssprog, og præcis overholdelse af disse regler er nødvendig for, at programmet kan kompileres og køre korrekt. Fejl i syntaks skal rettes, før programmet kan eksekvere. 16 af de i alt 21 adspurgte studerende rangerede syntaksfejl som værende den største begrænsning for deres læring. Denne udfordring skyldtes ofte, at de studerende, især i starten, brugte meget tid på at rette mindre fejl, hvilket afledte deres opmærksomhed fra programmets overordnede struktur. Ligeledes oplevede de studerende denne fejlsøgningsproces som frustrerende, da fejlmeddelelserne fra de forskellige anvendte udviklingsmiljøer ikke altid var behjælpelige.

Syntaksfejl er et problem flere visuelle programmeringssprog allerede har fundet en løsning på (Resnick et al. 2009). Ved at designe eksempelvis puslespilslignende brikker, der intuitivt kan forbindes på bestemte måder, tillades brugeren af *Blockly* eller *Scratch* ikke at sætte kodeblokke sammen, der ville give syntaksfejl (Resnick et al. 2009). Fordelen ved dette er, at brugeren hurtigt kan skabe et fungerende program og anskue sit program på et højere abstraktionsniveau (Resnick et al. 2009). Med dette tilegner brugeren sig dog heller ikke den viden om, eller færdigheder i, syntaksen i et tekstbaseret programmeringssprog.

Capability: To-vejs konvertering

Vores design tilbyder brugeren at tilgå sin kode gennem visuelle programmeringssprog. Den tekstbaserede kode og det visuelle programmeringssprog præsenteres samtidigt. Her kan brugeren manipulere i begge repræsentationer, der er synkroniseret. Til at demonstrere dette koncept har vi anvendt det eksisterende værktøj Blockly.

Konvertering fra et visuelt programmeringssprog som *Blockly* til et tekstbaseret sprog som JavaScript eller Dart eksisterer allerede i små løsninger. Vi har anvendt to af disse løsninger som prototyper; Cake-Core, der tillader *Blockly* til C konvertering for simple kodestykker, og *Blocklys* egen

konvertering til tekstbaseret kode. Her fandt vi, at de studerende var tiltrukket af muligheden for at visualisere deres program på et højere abstraktionsniveau. Alle de eksisterende værktøjer, vi kunne finde, tilbød ligeledes kun en-vejs konvertering, i.e. visualisering på baggrund af tekstbaseret kode eller tekstbaseret kode på baggrund af visuelt programmeringssprog. Vi introducerede derfor en simultan, tovejs konvertering mellem *Blockly* og C for at gøre vores design unikt. Denne teknologi gør det muligt for studerende at udnytte *Blocklys* visuelle syntaksfejlindikatorer, hvor inkompatible kodeblokke ikke kan sammenkobles, og giver dem mulighed for at foretage nødvendige rettelser i koden efter behov.

Idéen til denne to-vejs konvertering bygger på Paperts konstruktionisme (Papert 1980). Papert beskriver, at mennesker lærer bedst, når de manipulerer fysiske objekter og modtager øjeblikkelig, fysisk feedback. I vores digitale design emulerede vi det ved, at brugeren kan iterere og sammensætte blokke for at se en øjeblikkelig effekt i C - og omvendt. I en programmeringskontekst anså vi ligeledes fejlsøgningsprocessen ved syntaksfejl som værende til hindring for den feedback, den studerende kunne modtage ved at eksekvere programmet. Dette bundede i, at syntaksfejl ikke tillader brugeren at eksekvere programmet overhovedet. Ved at nedbringe tiden det tog at finde fejl, tillader vi brugeren at modtage feedback hyppigere og hurtigere. På den måde søgte vi at genskabe Paperts principper digitalt.

7.1.4 Manifestation: Tekniske udfordringer

Manifestationen *Tekniske udfordringer* er hos de studerende en overordnet betegnelse for en række brugerfejl, problemer med udviklingsmiljøet eller på anden vis problemer med at kompilere og eksekvere programmet. Det indebærer eksempelvis at importere de korrekte biblioteker på korrekt vis, navigere og give de korrekte instrukser i terminalen eller få opsat projekterne korrekt ved brug af Flask og Python eller lignende. Flere af de adspurgte studerende forklarede ligeledes, at de i løbet af kurset ikke modtog introduktion til teknologier som Git (versionsstyring), Visual Studio (udviklingsmiljø) eller Flask (web-framework i Python). Her oplevede næsten halvdelen af de adspurgte studerende, at deres læringsudbytte blev væsentligt forringet som følge af tekniske udfordringer. Dog fandt de studerende først dette som værende en væsentlig udfordring, når de skulle skifte fra de isolerede CS50 øvelser til at udvikle hele applikationer.

Vores design er primært orienteret mod at kunne løse isolerede øvelser i CS50, og ikke på at bygge hele applikationer ved brug af en længere række teknologier. I vores design valgte vi også derfor at fokusere på de udfordringer, der opstod primært i de isolerede øvelser. Dette betød, at vi i vores design begrænsede brugerens muligheder for at udføre handlinger, der potentielt kunne føre til unødvendig frustration. Ligeledes skulle tekniske anliggender som at importere biblioteker eller gå tilbage til tidligere versioner være let tilgængelige og intuitive.

Capability: Sandbox

I vores design løste vi udfordringerne ved at fokusere på brugervenlighed. Under processen brainstormede vi situationer og handlinger, hvor uønskede udfald opstod. Vi baserede dette på Normans heuristikker for brugervenlighed (Nielsen 1994) og planlagde at foretage tests af brugervenligheden i vores design. Dette indebar forskellige beslutninger i design- og systemudvikling, som eksempelvis at installere en lang række kendte biblioteker i vores miljø, hvor koden kunne behandles, selvom den ikke var kompilerbar. På denne måde kunne en manglende import af et bibliotek kommunikeres til brugeren, men programmet kunne stadig eksekveres. På denne måde anså vi vores løsning som et sandbox-miljø, hvor den studerende blev tilbudt en række forskellige værktøjer og quality of life-funktioner, der kunne hjælpe med deres programmering uden forudgående tekniske færdigheder.

7.1.5 Manifestation: Feedback og gennemsigtighed

Manifestationen *Feedback og gennemsigtighed* er hos de studerende relateret til den semantiske del af deres programmering. I programmering henviser semantikken til betydningen af konstruktioner inden for et sprog, i modsætning til syntaksen, som beskriver struktur og form. Semantik fokuserer på, hvad koden gør – hvordan handlingerne udføres, snarere end hvordan de udtrykkes. 15 af de adspurgte studerende angav mangel på feedback og gennemsigtighed som en væsentlig udfordring i deres forløb. De forklarede, at elementer som datastrukturer og typer, sorteringsalgoritmer, hukommelseshåndtering eller kontrolstrukturer i programflowet var svære at forstå og frustrerende at arbejde med. I interviews forklarede tre ud af seks studerende, at de ikke havde færdiggjort flere af øvelserne i CS50 på grund af semantiske fejl. De studerende gav eksempler på, at de ofte skulle "lære, hvad et hashtable eller en linked list er, mens de prøvede at få det til at fungere i koden."Dette betyder, at de studerendes forståelse for eksempelvis en linked list ofte var afhængig af, om de havde fået programmet til at virke efter hensigten. De studerende udtrykte, at de ofte manglede en form for gennemsigtighed i forhold til, hvad der skete trinvist i deres program. Det betød, at det var frustrerende for dem at identificere semantiske fejl i eksempelvis deres sorteringsalgoritme, når de ikke kunne følge med trin for trin.

Capability: Trinvis eksekvering og visuelt output

I vores design tilbydes der to former for programudførelse: trinvis eksekvering, hvor hver linje kode eksekveres med en lille pause imellem, og almindelig eksekvering, hvor programmet udføres så hurtigt som muligt og uden pauser. Valget mellem disse to er valgfrit. Desuden vil de forskellige visualiseringsstrategier, som *Blockly* – der allerede understøtter dette – kunne give forskellige visuelle indikatorer på lignende vis. Formålet var at tilbyde brugeren en mere simpel debugger, hvor særligt nye studerende let kunne pause og genstarte deres kode uden forudgående kendskab til *Debuggers*.

I vores proces arbejdede vi også med at visualisere output på andre måder end som tekst i en

terminal. Under designprocessen havde vi idégenereringsaktiviteter, hvor vi foreslog alternative måder at integrere trinvis eksekvering med eksempelvis AI-genererede visualiseringer. Vi inkorporerede ikke dette i vores endelige design, men har dsikuteret AI yderligere i kapitlet *Diskussion*.

Paperts konstruktionisme understreger vigtigheden af at understøtte feedback og iteration (Papert 1980). Her argumenterede vi for, at højere gennemsigtighed i eksekveringen af et program ville tillade brugeren at opnå reel feedback på semantiske fejl, der ellers kunne være svære at identificere ved almindelig eksekvering. Her argumenterede vi ligeledes for, at tilføjelsen af den visuelle kanal i form af illustrationer, diagrammer eller animationer, når programmet blev eksekveret, ville øge den studerendes indlæring af koncepter.

7.2 Pair programming og samarbejde

Vores løsning adresserer udfordringerne ved at facilitere computational thinking gennem en designstrategi, der integrerer pair programming som en kernefunktion. Pair programming, en teknik udviklet inden for Extreme Programming (Larman 2004), involverer to programmører, der samarbejder om en enkelt computer, eller i vores løsning; et fælles view, hvor én person ('driver') skriver koden, mens den anden ('navigator') foreslår forbedringer og identificerer fejl (Williams og Kessler 2000). Denne metode har vist sig at være særlig effektiv i uddannelsesmæssige sammenhænge ved at fremme dybere forståelse og engagement blandt studerende (Dybå og Arisholm 2005).

Fra vores litteraturstudie fremgik det, at studerende, der engagerer sig i pair programming, typisk opnår bedre resultater og oplever en større tilfredshed med læringsprocessen (McDowell et al. 2006). Dette understøttes yderligere af vores egen erfaring og brugerundersøgelser, hvor det blev tydeligt, at samarbejde kan mildne følelsen af isolation og forvirring, som kan opleves, når man arbejder alene. Remote pair programming er en løsning til at overkomme fysiske begrænsninger, idet det tillader studerende at deltage i realtidssamarbejde uanset deres geografiske placering. Denne mulighed er særligt relevant i situationer, hvor fysisk samarbejde ikke er muligt, såsom under vores egne erfaringer med at løse CS50 problemsæt mellem undervisningsgangene. Ved at implementere remote pair programming sikres det, at studerende stadig kan opnå de læringsmæssige fordele af pararbejde, selv når de er fysisk adskilt (Resnick et al. 2009).

7.2.1 Tidsbegrænsning

Vores løsningsforslag anvender rollefordeling med tidsbegrænsning for at understøtte samarbejde, hvor man giver de studerende et realtidsbillede af den opgave, man som par sidder med. Denne opdeling med tidsbegrænsning håber vi bidrager positivt til samarbejdsdynamikken, men også sikrer, at begge deltagere — driver og navigator — er aktivt engagerede og bidrager ligeligt til opgaveløsningen, hvilket adresserer nogle af de traditionelle udfordringer som ujævn arbejdsfordeling og "free-riding"(Dybå og Arisholm 2005). Dette kan selvfølgelig aldrig garanteres, da der lægger et ansvar ved de studerende, om de vælger at deltage aktivt. I vores løsning kan vi derfor blot sørge for, at der lægges op til ligelig tid i hver rolle.

7.2.2 Gruppesessions med central styring

En capability i vores løsning er muligheden for, at en underviser kan oprette og administrere gruppesessions, hvor alle studerende kan deltage samtidigt. Inspireret af, hvordan en platform som Zoom håndterer grupperum, kan denne funktion tillade undervisere at følge med i og hjælpe under gruppearbejde (Communications 2024). Undervisere vil have mulighed for at fordele studerende i forskellige undersessioner. Dette giver undervisere en styrende rolle, hvis behovet skulle være det, til at styre gruppedynamikken og sikre, at samarbejdet foregår på en struktureret og produktiv måde.

7.3 Opsummering

Vores række af features er mere eller mindre ikke blevet valideret. De bygger alle overvejende på eksisterende teorier om læring eller samarbejde. Om end vores oprindelige plan var at afprøve de forskellige features, nåede vi det ikke. Her anser vi stadig disse idéer for at være på konceptniveau, uden konkrete forslag til designet af disse features. Da vores arbejde med VizCode vil fortsætte udover rammerne for dette speciale, vil de sandsynligvis blive genbesøgt i takt med produktet udvikler sig.

I nedenstående tabel har vi opsummeret vores to features *Visualiseringsstrategier* og *To-vejs konvertering*, der begge er blevet testet og implementeret. Vi anser disse to for at være vores bidrag til at løse problemformuleringen "*Kan en kombination af grafiske og tekstuelle repræsentationer, i et introduktionskursus som CS50, støtte indlæringen på to måder: Computational thinking og Programmeringsbegreber?*"

Kan en kombination af grafiske og tekstuelle repræsentationer, i et introduktionskursus som CS50, støtte indlæringen på to måder: Computational thinking og Programmeringsbegreber?				
Feature	Beskrivelse	Grundlag	Kilder	Vores implementering
Visualiserings-strategier	VizCode benytter sig af visualiseringsstrategier, der er visuelle repræsentationer af koden, til at præsentere en kombination af grafiske og tekstuelle elementer under programmeringen.	Visualiseringsstrategier er baseret på Paivios teori om dobbeltkodning, der fortæller at læring bedst faciliteres når information præsenteres i både den verbale og den visuelle kanal.	(Clark og Paivio 1991)	Vi anvender Blockly, et velkendt Visuelt programmeringssprog for at udnytte en lang række fordele associeret med visuelle programmeringssprog såsom intuitiv, grafisk syntaks.
To-vejs synkronisering	VizCode benytter sig af to-vejs synkronisering, hvor brugeren kan manipulere alle repræsentationer af koden, hvorefter den synkroniseres.	To-vejs synkronisering er baseret på Paperts konstruktionisme der fortæller at mennesker lærer bedst når du manipulerer fysiske objekter og får en øjeblikkelig, fysisk feedback. Vi forsøger at emulere, at mennesker i Paperts konstruktionisme skaber en forståelse for abstrakte koncepter, ved at konstruere noget håndgribeligt.	(Papert 1980), (Resnick et al. 2009)	To-vejs synkroniseringen i VizCode er øjeblikkelig og uden væsentlige fejl. På nuværende tidspunkt understøtter den en lang række almindelige strukturer som løkker, variabler, kontrolstrukturer mm.

Tabel 7.1: Capabilities der eksisterer i VizCode på nuværende tidspunkt

7.4 Præsentation af design

I det følgende afsnit præsenteres nogle af de designprincipper, vi har anvendt i udfærdigelsen af VizCodes brugergrænseflade.

7.4.1 Designprincipper

Vi har i vores design forsøgt at følge nogle kerne designprincipper for at opnå den optimale brugeroplevelse og sikre en intuitiv brugergrænseflade.

- **Affordance:** Dette princip fokuserer på at designe elementer, så de naturligt indikerer, hvordan de kan og skal bruges. Målet er at gøre interaktionerne selvfølgelige for brugeren, så de kan navigere i applikationen uden besvær.
- **Signifiers:** Signifiers hjælper med at klargøre, hvor og hvordan interaktion skal finde sted. Ved at bruge visuelle eller tekstuelle hints sikrer vi, at brugerne intuitivt forstår, hvordan de skal interagere med forskellige dele af applikationen.
- **Feedback:** Feedback er afgørende for at informere brugerne om, at deres handlinger har haft en effekt. Det kan være auditivt eller visuelt, som begge bidrager til at forstærke brugerens forståelse og tillid til, at deres handlinger resulterer i ønskede ændringer.
- **Konceptuelle modeller:** Vi anvender enkle, intuitive modeller for at repræsentere komplekse systemer eller data, hvilket hjælper brugerne med hurtigt at forstå underliggende mekanismer uden at behøve at kende til de tekniske detaljer. Disse modeller er designet til at være lettilgængelige og forståelige (Norman 2002).

Ved at følge disse principper stræber vi efter at skabe en løsning, hvor brugervenlighed og funktionalitet går hånd i hånd, og hvor hver bruger kan føle sig sikker og kompetent i deres interaktion med vores produkt.

7.4.2 Wireframe

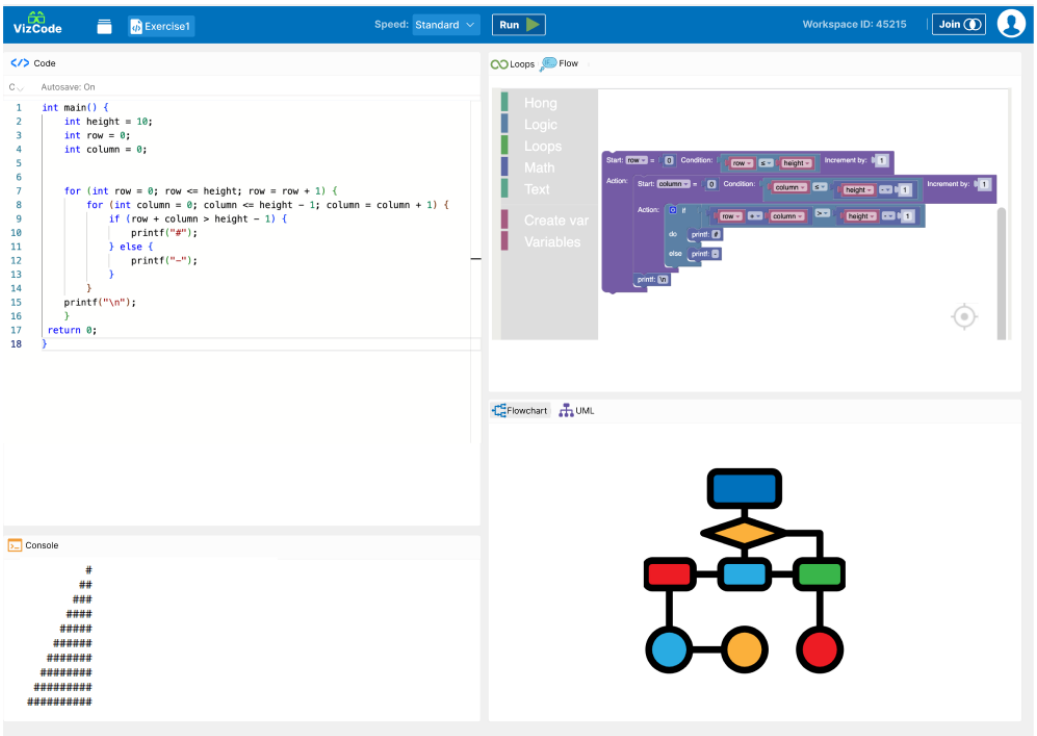
I vores wireframe har vi designet en brugerflade, der inspireres af funktionaliteten på Leetcode, især i hvordan den håndterer forskellige vinduer. Vores wireframe har fokus på at møde brugerens specifikke behov med forskellige visuelle strategier, med både en mulighed for at få vist kode i Blockly og/eller repræsenteret i UML eller flowchart-diagrammer.

Layout og navigationsbar

På figur 7.1 "Workspace-vinduet, hvor opgaveløsningen vil finde sted. Workspace-vinduet er opdelt i fire mindre vinduer (deres indhold bliver forklaret senere). Vinduerne kan justeres i størrelsen, så brugeren selv har frihed over, hvor meget hvert vindue skal fylde.

Navigationsbaren indeholder en række forskellige elementer: Til højre fra logoet er der et ikon, der repræsenterer de forskellige *opgaver*. Man kan trykke på den og derefter vælge, hvilken opgave man ønsker at løse. Den specifikke wireframe til det vindue er ikke lavet. Til højre for den ser man den *nuværende opgave*. Her forestiller vi os, at man kan trykke på opgaven for at få beskrivelsen vist. Endelig wireframe på det er heller ikke lavet. Til at håndtere debugging fremgår "*Speed*"i midten af navigationsbaren og en dertilhørende dropdown-menu. Speed refererer til hastigheden på eksekveringen af ens program. Sænker man hastigheden, vil man derfor kunne følge med i, hvor præcist ens program fejler. Ved siden af "Speed"er der en *Run-knap*, som compiler ens program.

I højre side af navigationsbaren fremgår det, hvilket workspace-ID man er tilknyttet, og ligeledes en *Join-knap* for at kunne tilslutte sig andres workspaces. Til slut i baren fremgår ens profil, hvilket heller ikke har fået en endelig wireframe.



Figur 7.1: Wireframe af 'workspace'-side

Implementering

8.1 Kanban

Vi benyttede os af Kanban til at strukturere vores udviklingsproces. I softwareudvikling er Kanban en metode, der anvendes til at styre og optimere arbejdsprocesser, især i forbindelse med Agile-udvikling (Atlassian 2024). Vi anvendte ligeledes enkelte ceremonier og artefakter fra Scrum, der er beskrevet nedenfor.

8.1.1 Backlog

Vores Kanban board fungerede som en *Backlog*, der er en prioriteret liste af opgaver i Scrum (SCRUMstudy 2014). I vores projekt dekomponerede vi de forskellige features i figur 7.1 til række mindre opgaver. Herefter var hvert teammedlem ansvarlig for selv at påtage sig opgaver i det omfang, de var i stand til. I vores backlog havde vi kategorierne "To do", "Doing", "In Review"og "Done".

8.1.2 Daily Scrum

Vi anvendte et koncept, der minder om *Daily Scrum*, hvor hver morgen blev startet med et kort morgenmøde. Her blev gårsdagens arbejde præsenteret, såvel som resten af dagen prioriteret. Daily Scrum stammer fra Scrum og anvendes blandt andet til kommunikation og koordinering (SCRUMstudy 2014).

8.1.3 Definition of done

Vi havde et fælles dokument med kriterier for opfyldelsen af en opgave. Dette indebar blandt andet en gennemgang af kodens struktur, således at den fulgte principper som SOLID. I review af opgaver blev de sammenlignet med kriterierne i vores Definition of done, før koden kunne blive integreret i det kørende produkt.

Vi anvendte ikke yderligere ceremonier fra Scrum eller andre frameworks. Som team har vi programmeret sammen i alle semestre på uddannelsen og har erfaret, at vores specifikke team udvikler bedst i mere frie rammer.

8.1.4 Resultat

Vi nåede at implementere *Visualiseringsstrategier* og *To-vejs konvertering* fra foregående afsnit. Vi arbejdede ud fra at implementere én feature ad gangen, hvor vi anså disse to for at være de bærende for vores koncept.

Vi nåede ligeledes at implementere en række standard features som *Login*, *Data persistence* og *Sikkerhed*. Vi kunne have abstraheret fra disse, for eksempelvis at have flere ressourcer til at afprøve forskellige elementer af vores løsning. Vores formål med denne implementering er at skabe funktionaliteten, der tillader os at generere proof of concept. Vi har dog behandlet vores løsning som et produkt, der skal lanceres, frem for at være en prototype. Med dette menes, at alle komponenter i vores løsning er udviklet med det mål at kunne blive lanceret efter dette projekt afsluttes. Denne beslutning diskuteres i kapitel *Diskussion*, da dette gjorde implementeringsprocessen mere tidskrævende, end hvis vi havde arbejdet ud fra eksempelvis MVPs af Ries 2011.

8.2 Objekt-orienteret analyse og design

Vores design af systemet bygger på Herbert Simons begreb om *Near-decomposability*, som beskrevet af Sarasvathy 2003. Formålet var at designe og implementere vores systemer som værende *near-decomposable* subsystemer, der kunne operere næsten selvstændigt fra andre subsystemer. Sarasvathy argumenterer for, at det passer godt med Effectuation, da man nemt kan fjerne et subsystem uden nødvendigvis at skulle lave hele softwaren om, hvilket giver en *kontrol over fremtiden* (Sarasvathy 2003). Vi så et potentiale i at designe og implementere vores løsning således, at nye Manifestationer kunne løses ved eksempelvis at samle nogle specifikke, selvstændige subsystemer til en ny tjeneste. Vi har kompetencer i *Objektorienteret analyse og design* (OOAD) fra tidligere fag og projekter. Her anså vi Simons *Near-decomposability* som værende et gennemgående mål og en filosofi for vores software, og OOAD som metodologien til at opnå dette.

8.2.1 Visualiseringsstrategier som aggregat af forskellige tjenester

Vores primære formål med løsningen er, at brugeren kan tilgå sit program på et højere abstraktionsniveau. Vores sekundære formål med løsningen er at tilbyde flere forskellige måder at illustrere dette højere abstraktionsniveau på, således at disse visualiseringsstrategier kan agere *scaffolds*, når de er relevante for den studerende, med tanke på Vygotskys *Zone for nærmeste udvikling* (Wood, Bruner og Ross 1976). Vi har dog ingen viden om, hvilke visualiseringsstrategier der er mest anvendelige eller effektive, ej heller hvornår i et CS50-forløb de er mest relevante. Vi er ligeledes bevidste om, at udvikle en visualiseringsstrategi fra bunden er en ressource- og tidskrævende proces. Vi lærte i vores test af både *Cake-core* og *Flowgorithm*, at andre allerede har lavet gode værktøjer til dette. Derfor valgte vi en fremgangsmåde, hvor vi ville anvende eksisterende værktøjer, som *Blockly*s egen engine, og tilpassede dem vores platform. For hver visualiseringsstrategi kan vi således udvikle de 'mangler', der

måtte være, for at skabe blandt andet to-vejs interaktion, hvis dette findes meningsfuldt. Mange af disse værktøjer udnytter forskellige dataformater og programmeringssprog, men bygger på de samme grundlæggende mekanismer. Vi så udfordringen, og deraf muligheden for at udnytte moduler, der løser én visualiseringsstrategi, som *Blockly*, til at løse en anden visualiseringsstrategi som *Flowgorithm*. På samme måde kunne vi minimere risikoen for, at moduler blev irrelevante. Hvis eksempelvis *Flowgorithm* viste sig at være en uhensigtsmæssig løsning til at støtte indlæring, kunne vi fjerne den del af løsningen uden at påvirke andre dele, altså et *affordable-loss*.

8.2.2 Microservices som arkitektur

Microservices refererer til en arkitektonisk tilgang, hvor en applikation er opdelt i en samling af mindre, uafhængige tjenester. Hver mikroservice kører i sin egen proces og kommunikerer med andre tjenester via et veldefineret interface, som ofte bruger protokoller som HTTP. Denne metode står i kontrast til den traditionelle monolitiske arkitektur, hvor alle dele af en applikation er tæt integreret og kører som en enkelt proces (Newman 2015).

I vores løsning betyder dette, at systemet er opdelt i tjenester. Et eksempel er, hvordan det at kompilere koden sker i en selvstændig applikation, mens det at eksekvere programmet sker i en anden, selvstændig applikation. Vi anså dette for at være et naturligt valg i vores kontekst, da Microservices effektivt kan skaleres, ændres, fjernes eller tilføjes (Newman 2015). Det giver os flere fordele i vores design, blandt andet dem som er beskrevet i nedenstående liste.

1. **Visualiseringsstrategier:** De forskellige visualiseringsstrategier kan *pakkes* af forskellige tjenester. En Microservice, der specifikt konverterer tekst-baseret kode til et dataformat som AST (Abstract Syntax Tree), kan være relevant for flere typer af visualiseringsstrategier. Tilføjelsen af en ny visualiseringsstrategi kan således påbegyndes med at undersøge, hvorvidt vores tjenester blot kan sammensættes på en ny måde for at opnå den ønskede funktionalitet.

2. **Skalering:** Såfremt vi i fremtiden bevægede os væk fra Aalborg Universitet mod et større marked, er Microservices effektive at skalere. Da hver Microservice fungerer som en selvstændig enhed, kan vores system skalere specifikke funktionaliteter som *Login* eller *Kompilering*, såfremt disse anvendes ofte og er mere belastede.

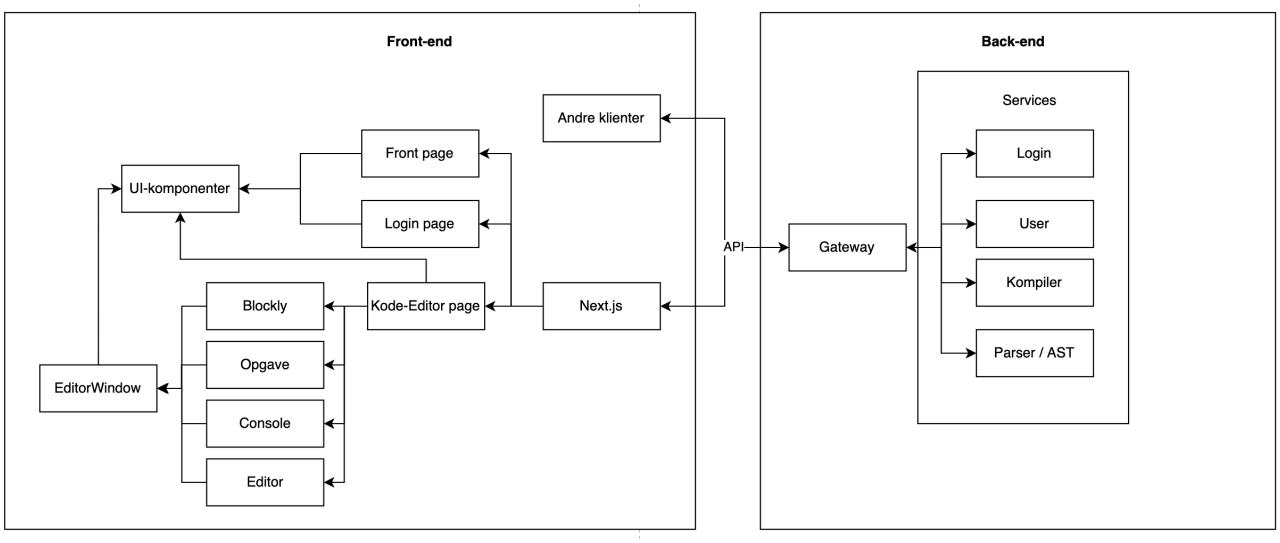
3. **Udskiftning:** Når tjenesterne eksisterer for sig og er mere eller mindre uafhængige, kan vi lettere fjerne eller udskifte enkelte tjenester, såfremt det videre arbejde med VizCode ændrer på vores bidrag til problemet.

8.3 Systemudvikling og arkitektur

I figur 8.10 illustreres den overordnede arkitektur af den samlede løsning. Diagrammet præsenterer den overordnede arkitektur opdelt i to dele: front-end og back-end, som kommunikerer gennem et *Application programming interface* (API). En API er et sæt af regler og protokoller, der gør det muligt for forskellige softwarekomponenter at kommunikere med hinanden (IBM 2024). Med udgangspunkt i princippet om Near-decomposability og OOAD, som redegjort i 8.2, valgte vi at adskille front-end fra back-end, hvilket skaber en lav kobling mellem systemerne, og med brug af API beholder vi en høj samhørighed. Dette muliggør, at udskiftning og ændringer i et af systemerne kan implementeres uden at påvirke det andet system.

I front-end delen på figur 8.10 illustreres to front-end systemer: "andre klienter"og Next.js. "Andre klienter"refererer til muligheden for, at andre applikationer, såsom mobil- eller computerapplikationer, kan interagere med vores back-end uden at påvirke de eksisterende klienter.

Next.js udgør vores applikation, der er struktureret omkring komponentmoduler, med hver deres ansvarsområde. Hvordan vi har struktureret vores system, kommer i 8.3.2 afsnit.



Figur 8.1: Oversigt over arkitektur

Back-end delen består af en Gateway og en række mikroservices. Gatewayen modtager og router requests fra front-ends via API til de relevante tjenester. Gatewayen håndterer også sikkerhed, hvor requests autoriseres ved brug af JWT-tokens. Dette betyder, at det kun er valide requests, der når forbi Gatewayen og får adgang til de forskellige tjenester. Bag Gatewayen har vi en række forskellige tjenester. Som beskrevet i 8.2.2 om microservice er koblingen mellem hver tjeneste lav, hvilket betyder, at hver enkelt kan fungere uafhængigt af hinanden. Dette gør det nemt at udskifte, fjerne og tilføje nye tjenester og funktionalitet, hvor det kræver minimale eller ingen ændringer i de andre tjenester. En

væsentlig fordel ved denne arkitektur er, at hver tjeneste ikke er bundet til en specifik teknologi. For eksempel anvender vi .Net Core i vores Kompiler-service, mens vores Parser-service bruger Node.js.

8.3.1 Værktøjer

Docker og Docker compose

Da hver tjeneste kører uafhængigt af hinanden, er der behov for en metode til at holde styr på hver individuel tjeneste. Til dette formål benytter vi Docker-containere og Docker Compose. En Docker-container er et selvstændigt kørende miljø, hvor alle nødvendige afhængigheder er tilpasset og installeret, så en tjeneste kan køre inden i containeren (Docker, Inc. 2024). Her er hver tjeneste en applikation for sig selv, der kører i et isoleret miljø, der kan have sin egen database.

For at have en mere strømlinet håndtering af tjeneste-versioner og deployment-processer anvender vi Docker Compose. Docker Compose er et værktøj til at administrere flere containere som en enkelt tjeneste. Dette betyder, at vi har et samlet sted, hvor vi deployer og vedligeholder forskellige tjenester.

React, next.js, typescript

React er et komponentbaseret JavaScript UI-bibliotek, der bruges til at bygge interaktive brugergrænseflader og klient-funktionalitet. Reacts komponenter virker på den måde, at hver komponent er en lille isoleret del af systemet med egen logik og UI, som kan bruges på kryds og tværs af komponenterne, der samles for at udgøre systemet (React Contributors 2024). I 8.10 front-end Next.js ses, hvordan Kode-Editor page kan have flere sub-komponenter og hvor alle komponenterne bruger genbrugelige UI-komponenter.

React er et bibliotek, der udelukkende bruges til UI og behandling af klientdata, derfor bruger vi Next.js, som er en samling af værktøjer bygget omkring React. Next.js gør det muligt at udvikle højtydende og skalerbare web-applikationer. Disse værktøjer omfatter blandt andet navigering, optimering og deployment, og frameworket danner derfor grundlaget for vores web-klient (Vercel 2024).

Derudover bruger vi TypeScript, som er en overbygning eller "supersæt"af JavaScript, der tillader os at definere de typer af data, vi forventer at arbejde med (Microsoft 2024b). Dette giver os IntelliSense, som tydeliggør hvilke datatyper og parametre vi arbejder med. Det sikrer en mere sikker kodebase, hvor vi undgår typiske kompileringsfejl og får hurtig valideringsfeedback, som gør det nemmere at videreudvikle og vedligeholde kodebasen (Microsoft 2024c).

C# og .Net Core

Flere af vores tjenester anvender C# og .NET Core. C# er et objektorienteret programmeringssprog, vi anvender sammen med .NET Core frameworket. NET Core er et fleksibelt, cross-platform framework (Microsoft 2024a), der også inkluderer en række out-of-the-box features som brugeradministrering og login.

8.3.2 SOLID, KISS og DRY

For at konkretisere hvornår og hvordan vi bygger vores komponent system, benytter vi principperne SOLID, Don't Repeat Yourself (DRY) og Keep It Simple, Stupid (KISS) som udviklingsmetode. Særligt SOLID er principper, vi har forsøgt at implementere efter, og disse principper redegøres i nedenstående afsnit.

SOLID repræsenterer fem grundlæggende principper for objektorienteret design i softwareudvikling. Disse principper er designet til at gøre software mere forståelig, fleksibel og vedligeholdelig (Martin 2024). Disse har været en del af vores Definition of Done og review-process, om end der altid vil være afvigelser.

1. **S - Single Responsibility:** Hvert komponent skal kun have én grund til at ændre sig. Dette betyder, at hvert komponent ligeledes skal have kun ét ansvarsområde.
2. **O - Open/Closed:** Komponenter skal være åbne for udvidelser, men lukkede for modifikationer. Dette betyder, at man skal kunne tilføje ny funktionalitet uden at ændre eksisterende kode. Når et komponent er skrevet og testet, bør koden ikke ændres. Dette hjælper med at undgå utilsigtede fejl og bevare systemets stabilitet.
3. **L - Liskov Substitution:** Objekter af en klasse skal kunne erstatte objekter af basisklassen uden at påvirke programmets korrekthed.
4. **I - Interface Segregation:** Flere, specifikke interfaces er bedre end et enkelt, generelt interface.
5. **D - Dependency Inversion:** Moduler på et højere niveau skal ikke afhænge af moduler på et lavere niveau. Begge bør afhænge af abstraktioner. Abstraktioner skal ikke afhænge af detaljer; detaljer skal afhænge af abstraktioner.

SOLID-principperne var særligt aktuelle i de tjenester, hvor vi anvendte det objektorienterede C# og .Net Core. Dette skyldes, at SOLID er specifikt udviklet til objektorienteret software. I vores front-end, hvor vi anvendte React og Next.js, anvendte vi mere KISS og DRY.

Hunt og Thomas beskriver DRY som *"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"(Hunt og Thomas 1999).* DRY-princippet betyder i vores sammenhæng, at man skal undgå at skrive gentagende kode; hvis det sker, skal det laves om til en selvstændig komponent, så komponenten kan genbruges og skabe en lav kobling.

KISS-princippet refererer til, at hvis en komponent bliver kompleks, skal man overveje, hvordan man kan abstrahere noget af logikken til en ny og mindre komponent, så komponenten bliver simplificeret (Baeldung 2024). Derudover giver KISS anledning til, at man vælger de simple kodeløsninger, når det er muligt.

I kombination med Reacts komponent system betyder dette, at hvert komponent skal have et klart ansvarsområde. Når en komponent bliver for kompleks, skal den opdeles i mindre komponenter. Hvert komponent skal desuden være modulært, så dens funktionalitet kan udvides uden at påvirke andre komponenter, der benytter den. Dette resulterer i, at hele web-klienten består af en samling fleksible komponentmoduler, der kan ændres afhængigt af de parametre, de modtager. Dette gør det nemmere at bevare kode, selv når der foretages store ændringer, da systemet består af komponenter med lav kobling og høj samhørighed.

På figur 8.10 illustreres en genanvendelig ImageButton-komponent, hvis ansvar er at vise en knap med et billede. Dette er et klassisk eksempel på anvendelsen af KISS og DRY. Komponenten følger KISS- og DRY-principperne, da den er meget simpel og kan anvendes flere steder i systemet. Fleksibiliteten opnås ved brug af parametre, som det ses på linje 11, hvor man giver billedeinformation, størrelse og funktionalitet, som overføres til komponentinstansen.

```
11  export default function ImageButton(text: string, src: string, alt: string, onClick: () => void, dimensions?: number) {
12      return (
13          <>
14              <button
15                  type="button"
16                  className="inline-flex items-center h-6 p-1 text-sm text-black border border-transparent rounded-lg hover:bg
17                  onClick={onClick}
18              >
19                  <Image src={`/images/${src}`} alt={alt} width={dimensions || 25} height={dimensions || 25} />
20                  <div className="pl-1">{text}</div>
21              </button>
22          </>
23      );
24  }
```

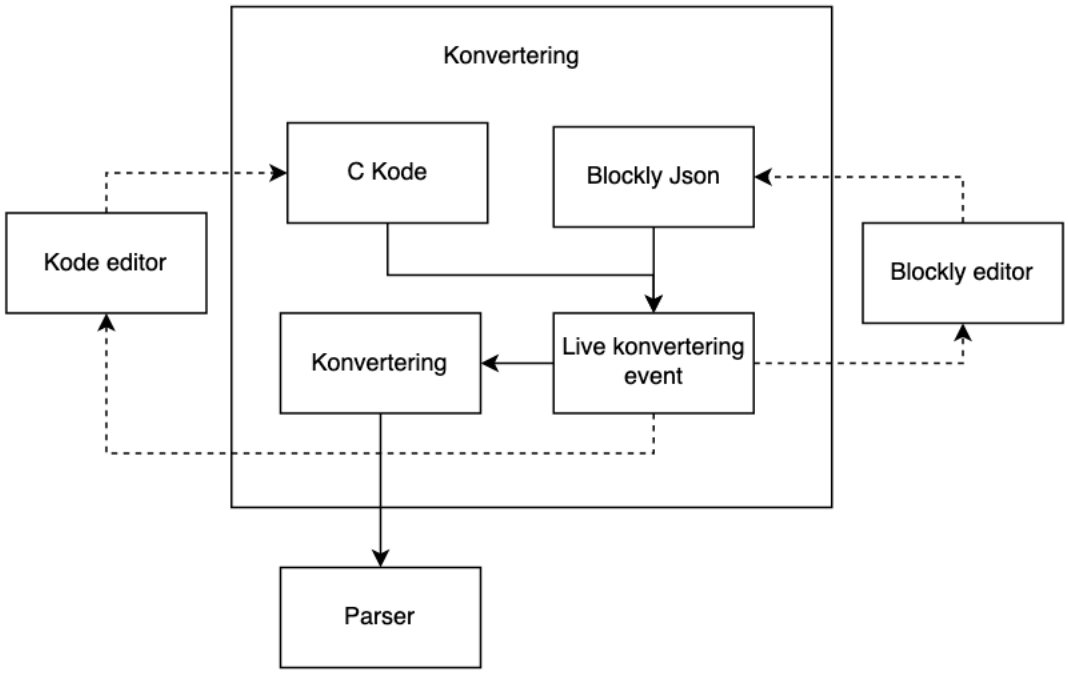
Figur 8.2: Dekomponeret og genbrugt button-komponent

8.4 Visualiseringsstrategien Blockly

8.4.1 To-vejs konvertering

To-vejs konvertering består af fire hovedkomponenter: Blockly-editor, Kode-editor, Parser og Konvertering, som illustreret på figur 8.10. Blockly og Editor håndterer deres egne funktionaliteter og indstillinger. Konverteringskomponenten fungerer som mellemled og sørger for at synkronisere tilstanden mellem dem. For at konvertere C-kode til blokke bruger Konverteringskomponenten en Parser, som er placeret i back-end. Parseren returnerer et AST, som vi bearbejder og omsætter til et samlet Blockly Json. Derudover har den ansvar for at notificere modsatte editorer, når ændringer opstår. Dette sker

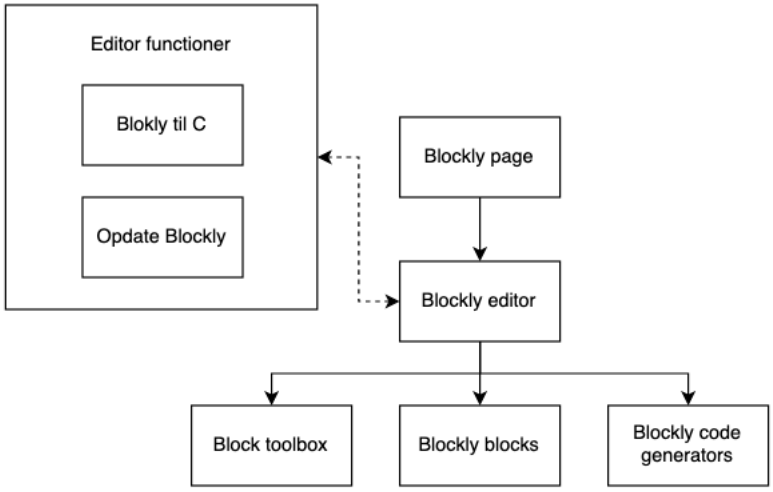
ved, at Live konvertering event-funktionen observerer C-kode og Blockly Json dataene. Når ændring sker et af stederne, sendes et event til den modsatte part, som vist i konverteringsoversigten i 8.10.



Figur 8.3: To-vejs konvertering

8.4.2 Implementering af Blockly

Blockly består af en standard UI-kanvas, som ses i højre hjørne i 8.10 og fjorten custom lavede blokke, som hver skal defineres i Toolbox, Generators og Blok komponenterne. Derudover består den af to funktioner **Blockly til C** og **Opdater Blockly**, som samlet illustreret på figur 8.10.



Figur 8.4: Blockly komponent

Toolbox er en del af kanvas, hvor man definerer, hvilken gruppering blokkene skal være en del af, og hvilke standardværdier de skal have, hvilket ses i 8.10.

Blokkenes UI defineres i et specifikt JSON-format, som er defineret af Blockly. Her beskriver man, hvilke forbindelser, tekst og funktionalitet en given blok skal have. I figur 8.10 ses, hvordan vi har defineret en aritmetik-blok, der har to inputfelter og et aritmetisk tegn, hvor alle har en type og et navn, som bruges til at identificere feltet i kodegeneratoren.

For at generere C-kode fra blokkene bruger vi en modificeret version af Blockly-generator. I figur 8.10 ses, hvordan vi henter feltværdierne fra math-arithmetic blokken, som samler dem i "code-variablen, som derefter returnerer værdierne og beskriver, hvordan det skal konverteres til kode, ift. om blokken er selvstændig eller del af andre blokke.

```
{
    kind: 'category',
    name: 'Math',
    categorystyle: 'math_category',
    contents: [
        {
            type: 'math_arithmetic',
            kind: 'block',
            fields: {
                OP: 'ADD',
            },
            inputs: {
                A: {
                    shadow: {
                        type: 'math_number',
                        fields: {
                            NUM: 1,
                        },
                    },
                },
                B: {
                    shadow: {
                        type: 'math_number',
                        fields: {
                            NUM: 1,
                        },
                    },
                },
            },
        },
```

Figur 8.5: Blockly toolbox

```
{
    type: 'math_arithmetic',
    message0: '%1 %2 %3',
    args0: [
        {
            type: 'input_value',
            name: 'A',
            check: 'Number',
        },
        {
            type: 'field_dropdown',
            name: 'OP',
            options: [
                ['+', 'ADD'],
                ['-', 'MINUS'],
                ['*', 'MULTIPLY'],
                ['/', 'DIVIDE'],
            ],
        },
        {
            type: 'input_value',
            name: 'B',
            check: 'Number',
        },
    ],
    inputsInline: true,
    output: 'Number',
    style: 'math_blocks',
},
```

Figur 8.6: Blockly blok

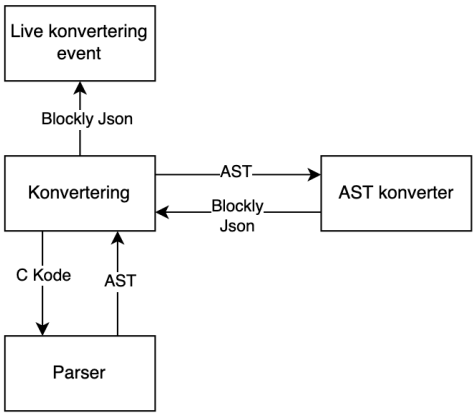
```
forMathBlocks['math_arithmetic'] = function math_arithmetic(
    block: Block,
    generator: JavascriptGenerator
): [string, Order] {
    // Basic arithmetic operators, and power.
    const OPERATORS: Record<string, [string | null, Order]> = {
        ADD: [' + ', Order.ADDITION],
        MINUS: [' - ', Order.SUBTRACTION],
        MULTIPLY: [' * ', Order.MULTIPLICATION],
        DIVIDE: [' / ', Order.DIVISION],
        POWER: [null, Order.NONE], // Handle power separately.
    };
    type OperatorOption = keyof typeof OPERATORS;
    const tuple = OPERATORS[block.getFieldValue('OP') as OperatorOption];
    const operator = tuple[0];
    const order = tuple[1];
    const argument0 = generator.valueToCode(block, 'A', order) || '0';
    const argument1 = generator.valueToCode(block, 'B', order) || '0';
    let code;
    // Power in JavaScript requires a special case since it has no operator.
    if (!operator) {
        code = 'Math.pow(' + argument0 + ', ' + argument1 + ')';
        return [code, Order.FUNCTION_CALL];
    }
    code = argument0 + operator + argument1;
    return [code, order];
};
```

Figur 8.7: Blockly generator

Den ene funktion i 8.10 er Blockly til C-funktionen, som konverterer blokkene baseret på, hvordan de er beskrevet i deres generatorer. Derefter sender den koden videre til konverteringskomponenten, som opdaterer Kode-editoren. Opdater Blockly-funktionen lytter til ændringer fra konverteringskomponenten og opdaterer Blockly, når den registrerer et event. Figur 8.10 viser, hvordan komponenterne og deres funktioner arbejder sammen.

8.4.3 Implementering af konverter

Når C-koden bliver opdateret fra Kode-editor, gennemgår den en konverteringsproces, hvor koden først bliver overført til Parser, som omdanner C-koden til et AST. Dette AST sendes videre til AST-konverteren, som opbygger et samlet Json, som til sidst sendes og loades ind i Blockly, som illustreret i 8.10.



Figur 8.8: Konvertering af C til Blockly

AST består af en række statements, der repræsenterer forskellige C-kode kodestykker. På figur 8.10 vises kernefunktionen executeFunctionByType, hvor en switch/case dirigerer de forskellige statementtyper over til den korrekte konverteringsfunktion, som konverterer AST-statementen til Json. Et eksempel på, hvordan konvertering foregår, illustreres i figur 8.10, som viser konverteringen af en aritmetik-blok med to inputblokke og et aritmetisk tegn. For at kunne bygge dette Json-blok, har vi først brug for at vide, hvilken input der er brugt i C-koden. Dette gør vi ved at kalde executeFunctionByType på begge inputs. Derefter identificerer den anvendte aritmetiske tegn. Til sidst sammensættes det hele i en Json, som returneres.

```
97  function executeFunctionByType(data: allBlockTypes, from?: any): any {
98      switch (data.type) {
99          case 'ForStatement':
100             return custom_for_loop(data as astForloop);
101         case 'IfStatement':
102             return controls_if(data as astIfElse);
103         case 'Literal':
104             return literal_function(data as literal);
105         case 'BinaryExpression':
106             return logic_compare(data as unknown as binaryIdentifier);
107         case 'Identifier':
108             return variables_get(data as identifier);
109         case 'BlockStatement':
110             return blockStatement(data as astBlockStatement);
111         case 'math_number':
112             return math_number(data as astMath);
113         case 'AssignmentExpression':
114             return assignment_expression(data);
115         case 'ExpressionStatement':
116             return text_print(data as astPrint);
117         case 'VariableDeclaration':
118             return VariableDeclaration(data as astDeclaration);
119         case 'VariableDeclarator':
120             return VariableDeclarator(data, from);
121         case 'CallExpression':
122             return text_print(data as unknown as astPrint);
123         case 'ArrayExpression':
124             return lists_create_with(data);
125         case 'EmptyStatement':
126             return;
127         default:
128             console.log('No function mapped for block type:', data.type);
129             break;
130     }
131 }
```

Figur 8.9: AST funktionskalder

```
213 function logic_compare(data: binaryIdentifier) {
214     const MATH_OPERATORS = {
215         '+': 'ADD',
216         '-': 'MINUS',
217         '*': 'MULTIPLY',
218         '/': 'DIVIDE',
219         '**': 'POWER',
220     };
221     let a = executeFunctionByType(data.left);
222     let b = executeFunctionByType(data.right);
223     type MathOption = keyof typeof MATH_OPERATORS;
224
225     if (data.operator in MATH_OPERATORS) {
226         return {
227             type: 'math_arithmetic',
228             id: idGenerator(),
229             fields: {
230                 OP: MATH_OPERATORS[data.operator as MathOption],
231             },
232             inputs: {
233                 A: {
234                     block: a,
235                 },
236                 B: {
237                     block: b,
238                 },
239             },
240         };
241     }
```

Figur 8.10: AST til Blockly

Når hele AST er blevet konverteret, sendes det til Blockly-editoren, som opdaterer Blockly-kanvas med de nye blokke.

I det følgende kapitel evaluerer vi den ovenstående implementering af *Visualiseringsstrategier* og *Blockly*.

Evaluation

Efter at have implementeret Blockly og to-vejs konverteringen testede vi produktet på os selv. Her indså vi ret hurtigt, at vi nok ikke længere er en del af målgruppen. Med det menes, at vi alle foretrak det tekstbaserede sprog og faktisk fandt Blockly til at være unødvendig støj i vores programmering. Her diskuterede vi, hvordan vores egen erfaring ligeledes kunne stå i vejen for forståelsen for de frustrationer de studerende oplever, omend vi har oplevet nogle af de samme. I sin bog "The Recursive book of Recursion"diskuterer Sweigart, hvordan forståelsen af rekursion fundamentalt ændrer en programmørs perspektiv og gør det svært at genkalde den oprindelige manglende forståelse af konceptet. Specifikt nævner han, at "once you learn recursion, you cannot unlearn recursion,"hvilket understreger pointen om, at det at lære komplekse programmeringskoncepter som rekursion ændrer ens tænkning permanent (Sweigart 2022). Med tanke på *Scaffolding*, hvor støtteværktøjer bliver overflødige når først kompetencerne er opnået, diskuterede vi hvordan vi sandsynligvis ikke kunne vurdere hvorvidt Blockly bragte en værdi eller ej ved at bruge os selv.

Vi ville gerne have indsigt i vores design, og hvorvidt designet var brugervenligt eller ej. Vi havde baseret størstedelen af designet på principperne redegjort for i kapitlet *Design* samt vores personlige erfaring med programmering. Her var vi dog særligt opmærksomme på, at vi kan have glemt, hvordan en ny studerende interagerer med en IDE. Vi benytter os af en lang række små 'teknikker' som genvejstaster eller extensions, der endnu ikke er kendte for de studerende.

Det primære mål med evalueringen var at få indsigt i, om vores løsning skaber værdi for den studerende eller ej. Vi havde forinden testen formuleret flere *Leap of faith*-antagelser af Eric Ries, der beskrives som grundlæggende antagelser, som skal være sande for, at dit produkt eller din virksomhed får succes (Ries 2011). Her beskriver Ries 2011 to forskellige typer af hypoteser, nærmere *Value* og *Growth* hypoteser. *Value-hypoteser* tester, hvorvidt der leveres en værdi til kunden, når produktet er i brug. *Growth* hypoteser tester, hvordan nye kunder vil opdage produktet (Ries 2011).

I dette projekt har vi fokuseret på *Value* hypoteser. Inden denne evaluering formulerede vi to grundantagelser vedrørende de specifikke features, vi havde implementeret i vores produkt. Den første value-hypotese omhandler *Visualiseringsstrategier* og at give de studerende mulighed for at tilgå deres program på et højere abstraktionsniveau. Dette formulerede vi således:

1. *De studerende vil værdsætte muligheden for at tilgå deres CS50-program på et højere abstraktionsniveau*

Den anden value-hypotese omhandlede vores feature *To-vejs synkronisering*, der har til hensigt at tilbyde de studerende at kunne modtage øjeblikkelig visuel feedback på den kode, de skriver. Dette var vores håndtering af de studerendes manifestation *Syntaksfejl*. Dette formulerede vi således:

2. *VizCode vil hjælpe den studerende med at finde og rette syntaksfejl i deres program, ved at de kan modtage øjeblikkelig visuel feedback, mens de programmerer.*

Vi testede vores implementering i en usability test. Denne test var designet ud fra Lazar, Feng og Hochheiser 2017s 8 stadier. Her skal det nævnes, at vi havde udfordringer med at rekruttere deltagere og derfor kun endte med fire deltagere. Vi fravalgte derfor at fokusere på en kvalitativ usability test som beskrevet i (Nielsen 2000), hvor Jakob Nielsen beskriver det optimale antal deltagere til en kvalitativ usability test som værende fem deltagere. De kvantitative mål indsamlet i testen var ligeledes ikke brugbare grundet det lave antal deltagere. Vi fik i stedet kvalitative indsigter i forskellige *usability issues*, som er afrapporteret i bunden af dette afsnit. I de følgende sektioner vil vi redegøre for henholdsvis vores usability test og vores efterfølgende fokusgruppe-interview.

9.0.1 Usability Test

1. *Vælg repræsentative brugere:* I vores usability test deltog fire studerende: Rikke, Stine, Mie og Mads. Alle de fire deltagere har gennemført CS50 eller lignende kursus inden for det sidste år og er i øjeblikket studerende ved Aarhus Universitet ved uddannelser i User Experience-Design og IT og Kommunikation. Deltagerne var i alderen 26-29 år og havde ikke arbejdet videre med programmering siden de gennemførte deres introduktionskursus. Dette er dog med undtagelse af en enkelt deltager, Mads, der siden introduktionskurset har arbejdet videre med objekt-orienteret programmering i mindre projekter.

2. *Vælg omgivelserne:* Lazar, Feng og Hochheiser 2017 anbefaler at situere testen i omgivelser, der minder om det miljø designet skal operere i. Vi foretog testsessionerne i et gruppelokale på Aarhus Universitet, der ligeledes anvendes af de studerende til projekt- og studierelateret arbejde. Lokalerne var dog tomme og stille, hvilket de ikke normalt vil være ifølge de studerende. Efter anbefalingerne beskrevet i (Lazar, Feng og Hochheiser 2017), forsøgte vi ligeledes at skabe en tryg atmosfære, hvor deltagerne følte sig frie til at ytre kritik og feedback.

3. *Vælg hvad brugerne skal udføre:* De studerende fik til opgave at løse indledende øvelser i CS50. Øvelserne var dog ændret på små områder, da vi selv har erfaring med, at de indledende øvelser sidder på rygraden. De studerende fik til opgave at løse nogle opgaver rettet mod specifikke handlinger i vores produkt, hvorefter de fik til opgave at løse de større øvelser. Deltagerne fik 2x30 minutter til at løse øvelserne.

4. *Beslut hvilken type data der skal indsamles:* Vi indsamlede kvalitative data under sessionerne i form af observationer.

5. *Før testsessionen:* Vi modtog samtykke fra begge deltagere og uddybede formålet med testen. Her forklarede vi også, at der til testen ville være en person til at facilitere og hjælpe, samt en observatør der tog noter. Vi forklarede om baggrunden for vores design og besvarede enkelte spørgsmål angående den kommende test. Derefter uddybede vi, at vi gerne ville modtage kritik og feedback på

alt, de kunne komme i tanke om. De studerende blev ligeledes instrueret i, at det var tilladt at tale under testsessionen, men at vi ikke noterede eller på anden vis fortolkede det, de sagde. Her havde vi i mente, at øvelserne kunne kræve en stor mængde koncentration, og deltagerne derfor skulle være frie til at tale med dem selv, tage noter eller på anden vis arbejde, som de normalt ville.

6. *Under testsessionen:* Under testsessionen gjorde vi en indsats for, at vores teammedlemmers tilstedeværelse ikke påvirkede udfaldet af testen. Vi spurgte ikke ind til specifikke handlinger, men besvarede enkelte spørgsmål under processen. Vi forsøgte ligeledes ikke at reagere på hverken stilhed eller tale fra de studerendes side.

7. *Debriefing efter testsessionen:* Debriefing-fasen er vigtig for dataindsamlingen, da det tillader at interagere med testpersonen med spørgsmål ud fra hændelser, der lige er forekommet (Lazar, Feng og Hochheiser 2017). Vi afholdt et fokusgruppe-interview lige efter testen, hvor vi spurgte ind til en række af de observationer, vi havde noteret.

8. *Opsummer resultaterne efter testsessionen:* Grundet det lave antal af testdeltagere så vi ikke et behov for, at data skulle undergå hverken kvantitative eller kvalitative analyser. Vi opsummerede derfor resultaterne i nedenstående tabel. Vi tildelte observationerne en alvorlighedsgrad i tråd med (Nielsen 2000), således vi kunne prioritere løsningerne dertil på et senere tidspunkt. Her tog vi udgangspunkt i, hvor stor en effekt vi troede problemet kunne have på brugerens samlede oplevelse af produktet.

9.0.2 Resultater fra Usability Test

Vi observerede en række *usability issues*, der er opsummeret nedenfor:

Opgave	Observation	Alvorlighed
1	Testdeltageren ledte efter en synkroniser-knap	Lille
2	Testdeltager undrede sig over små ændringer i original tekstkode ved to-vejs manipulation	Medium
3	Testdeltager lavede enkelte ændringer, der ikke synes at blive registreret i Blockly	Høj
3	Int main(void) hoppede frem og tilbage	Lille
4	Testdeltager slettede kode, der ikke blev slettet i Blockly	Høj
Øvelser	Testdeltagerne forsøgte forgæves at resize og trække vinduer rundt og forklarede, at særligt Blockly var meget småt	Høj
Øvelser	Testdeltager skulle nogle gange deklarere variable inden de blev brugt i løkker, andre gange virkede det	Mellem
Øvelser	Deltagerne forsøgte flere gange med Ctrl + S kommando og forventede at se en synkronisering	Høj
Øvelser	Deltager forsøgte at anvende argumenter i main (int main(argc, argv)) og pointerede, at flere af øvelserne i CS50 krævede dette	Høj
Øvelser	Studerende kunne ikke genskabe slettet kode med Ctrl + z	Høj
Øvelser	Deltagere udviste frustration, hvis de 'tabte tråden', når de konverterede tilbage fra Blockly, da formateringen af deres kode blev ændret	Høj

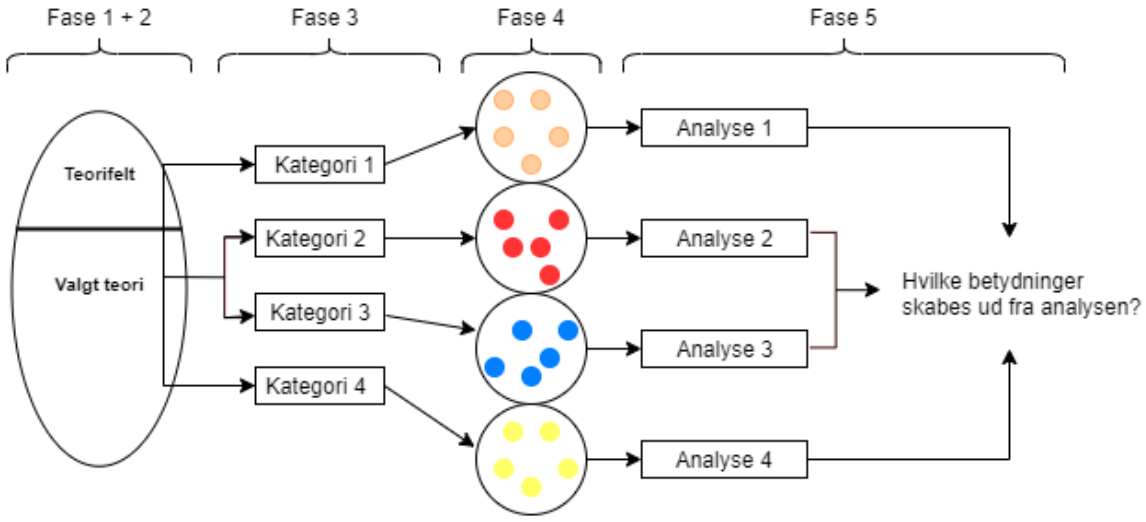
Tabel 9.1: Tabel over observerede usability problemer

9.0.3 Fokusgruppeinterview

Efter usability-testen afholdt vi et fokusgruppeinterview med alle deltagerne. Et fokusgruppeinterview er en interviewmetode, hvor en mindre gruppe, der er relevant for et givet emne eller tematik, interviewes med henblik på at få forskellige perspektiver og synspunkter på emnet (Aalborg University 2024, ss. 182-183). Fokusgrupper giver mulighed for dynamisk interaktion mellem deltagerne, hvor de kan bygge videre på hinandens ideer og perspektiver. Dette kan føre til en dybere forståelse af emnet, da deltagerne kan inspirere og udfordre hinanden (Aalborg University 2024, 182–183). Da vores deltagere havde varierende grader af erfaring med programmering, så vi dette som en mulighed for en informativ diskussion blandt de deltagende og os.

9.0.4 Tematisk Analyse

På baggrund af fokusgruppeinterviewet foretog vi en deduktiv tematisk analyse som beskrevet af Brandi og Sprogøe 2019. De opdeler en *deduktiv analyse* i 6 faser. Brandi og Sprogøe 2019s teori-drevne analyse er bedst egnet til at finde temaer og sammenhænge mellem de forudbestemte kategorier og materialet (Brandi og Sprogøe 2019). Med dette menes, at formålet med vores analyse ligeledes søgte at afdække nye mønstre, der opstod, når vi introducerede VizCode som værktøj. I den forstand håber vi ikke bare at undersøge, hvorvidt VizCode bringer værdi til den studerende, men også at afdække nye måder at interagere med programmering generelt, eftersom VizCode er (så vidt vi ved) et koncept, der ikke eksisterer.



Figur 9.1: Teoridrevet analyse – kerneelementer af Brandi og Sprogøe 2019, s. 63.

I fase 4 kodede vi vores interview, hvor vi ledte efter konkrete udsagn, der kunne relateres til hypoteserne. I denne proces deltog to fra vores projekt. I fase 5 identificerede vi mønstre mellem de

forskellige udsagn med henblik på at skabe en forståelse for de temaer, der opstod inden for kategorierne. I fase 6 fortolkede vi de forskellige kategorier og søgte at danne en overordnet forståelse for, hvordan hypoteserne relaterede sig til datamaterialet. Her skrev vi ligeledes vores resultater af analysen ud, hvor hver kategori fik en sammenhængende beskrivelse.

9.1 Resultater

9.1.1 Hypotese 1: *De studerende vil værdsætte muligheden for at tilgå deres CS50-program på et højere abstraktionsniveau*

Positiv Oplevelse med Blockly

Deltagerne, især Rikke, Stine og Mie, udtrykte en generelt positiv oplevelse med at bruge Blockly. Rikke forklarede blandt andet, at konceptet tiltalte hende meget, og at *"Jeg kunne også godt bruge det nu for at være ærlig"*, hvor hun refererer til sit daglige virke som UX-studerende. Mia uddybbede med, at hun havde gavn af at kunne slippe for syntaks og få sit program til at virke. En af deltagerne, Mads, udtrykte, at han var glad for konceptet, men ikke særlig glad for lige netop Blockly, der virkede *"unødvendigt rodet"*. Han forklarede i stedet, at det for ham var nemmere at både skrive og forstå koden ved bare at bruge C. Når spurgt om, hvorvidt der var andre værktøjer med samme princip, her med en bagtanke om at foreslå diagrambaserede sprog, svarede Mads, at han gerne ville *"slukke for Blockly og få Google frem [i vinduet] i stedet"*.

Tilvænning af Nyt Værktøj

Flere af deltagerne udtrykte, at det var en helt ny måde for dem at programmere på. Rikke fortæller blandt andet: *"Jeg tror det første, jeg lige tænkte, var sådan 'Wow, det skal jeg lige vænne mig til'. Men jeg synes, det gik ret hurtigt"*. Her kom flere af deltagerne ind på, at Blockly i sig selv er et sprog, man skal lære at læse. En af deltagerne, Mads, fandt dén måde at visualisere på som særligt rodet, mens de resterende deltagere fandt den blokbaserede repræsentation som lettere at læse og forstå. Her nævnte Mads også, at det kunne tage lidt tid at "fange konceptet".

9.1.2 Semantik

Deltagerne oplevede, at programmet var lettere at forstå og fejlsøge semantik, når koden var repræsenteret på et højere abstraktionsniveau. Stine forklarede det blandt andet med: *"Fordi koden er bare nemmere at læse i Blockly, det er jo ligesom Scratch. Så det er nemmere at regne ud med rækkefølge, fordi man forstår koden."* En af deltagerne, Mads, var dog uenig i dette og mente ikke, at Blockly havde givet noget særligt i forbindelse med dette. Her refererede han særligt til begyndelsen af en øvelse, hvor han nævnte, at *"Det er jo ikke fordi, man får sådan helt konkret hjælp til at komme i gang."* Her refererede Mie til, at hun havde lettere ved at finde semantiske fejl, da hun sagde: *"Mest fordi jeg bare kunne få*

koden til at virke for en gangs skyld", eftersom syntaksen blev et mindre problem for hende. Her fortalte hun, at hun via Blockly kunne *"følge med i koden trin for trin"*, fordi hun lettere kunne læse det.

Opsummering

Generelt indikerer resultaterne fra fokusgruppen, at deltagerne fandt konceptet i VizCode attraktivt og brugbart. Flere af deltagerne forklarede, at Blockly var en værdsat tilføjelse, mens en enkelt deltager ikke fandt Blockly værdifuldt. Flere af deltagerne beskrev VizCode som en tilvænningsproces, hvor eksempelvis Stine mente, at den *"ændrede måden, man lige normalt tænker på i de her øvelser"*. Stine, Mie og Rikke mente, at det højere abstraktionsniveau hjalp dem med lettere at forstå deres program. Her uddybede de, at Blockly var nemmere at læse, mens Mads mente det modsatte. Mads fremhævede, at VizCode måske gør programmet lettere at læse, men det anvender ikke andre virkemidler til at få den studerende i gang med øvelserne.

9.1.3 Hypotese 2: *VizCode vil hjælpe den studerende med at finde og rette syntaksfejl i deres program, ved at de kan modtage øjeblikkelig visuel feedback, mens de programmerer.*

Hjælp til Syntaks

Flere af deltagerne nævnte, at Blockly hjalp dem med at huske og anvende korrekt syntaks, specielt i de situationer, hvor de var usikre eller havde glemt det. Rikke og Mie brugte det blandt andet som skabeloner, specielt til for-loops. Derudover udtalte Stine: *"Jeg brugte det mest i starten, fordi jeg ikke lige kunne huske, hvordan de forskellige ting skulle stå."* Mie anvendte Blockly i særlig grad og udtrykte, at det var en stor hjælp for hende.

Interaktion med Feedback

Rikke, Stine og Mie bemærkede, at Blockly gav dem en visuel feedback, som hjalp dem med at identificere fejl tidligt i processen. Fælles for både Rikke, Stine og Mie er, at de synes fejlfindingen er nemmere, når man tager det løbende, hvilket den visuelle feedback kan være behjælpelig med. Da Mads udtrykte, at der jo endnu ikke er nogen fejlmeddelelser, før man har kørt programmet, svarede Stine: *"Jeg ved ikke helt, om det er blokkene eller hvad det er, men der er noget over det der med, at du står på linjen. Sådan at computeren siger 'ups, der trådte du over linjen.'"*

Alle deltagerne anvendte Blockly på lidt forskellige måder. Rikke forklarede, at hun primært skrev i den tekstbaserede kode, men hele tiden holdt øje med Blockly for at se, om blokkene nu også blev genereret, som hun ville. Stine anvendte Blockly og C mere fifty-fifty, mens Mie primært anvendte Blockly testen igennem, da hun havde problemer med at genkalde den korrekte syntaks.

Mads nævnte dog, at han ikke fandt det behjælpeligt til at fejlfinde, og at han foretrak at bruge fejlmeddelelser fra konsollen og aflæse dem. Mads har, som nævnt tidligere, en smule mere erfaring med koden og fortæller, at han allerede har en indlært praksis, når det kommer til at debugge.

Mindre Frustration

Både Mie og Stine nævnte, at de oplevede færre frustrationer med fejlfindingen, da Blockly gav dem mulighed for at rette syntaksfejl, før koden blev kørt. Mie nævnte ligeledes, at hun nogle gange kan synes, at hendes kode ikke er til at redde, hvis hun først fejlfinder til sidst og støder på mange fejl. Da interviewer spurgte ind til, hvad hun mente med 'reddes', svarede hun: *"Ja, nogle gange kan man ligesom få så mange fejl, at man bare ikke gider til at ordne dem."* Rikke fremhævede, at to-vejs synkroniseringen bidrog til en følelse af succes, da hun sagde: *"Når man var sådan nogenlunde sikker på, at man havde skrevet det der i parentesen rigtigt, og så kunne se det blive til et for-loop. Det var lidt sådan en du gjorde det rigtigt-følelse."* Her uddybede Stine, at det særligt hjalp på lokalisering af fejl, da *"Nu ved man jo, at 'okay, det er lige her, hvor jeg er nu, der er noget galt."*

Alle deltagerne fremhævede det positive i at kunne tage fejlene løbende. Flere af deltagerne fremhævede dog også, at man som Stine sagde, er *"overladt til sig selv"*. Her refererede hun til, at den primære indikator af fejl i en linje, mens den skrives, er, at Blockly-blokkene endnu ikke er genereret. Her forklarede Rikke, at hun af og til var nødt til at *"...google nogle gange for så at finde ud af, hvorfor jeg ikke kunne komme videre."* Her mente flere af deltagerne, at fejlmeddelelserne eller anden assistance kunne give endnu mere.

Opsummering

Resultaterne fra fokusgruppen indikerer, at vores løsning potentielt kan hjælpe studerende med at finde og rette syntaksfejl ved hjælp af øjeblikkelig visuel feedback. Rikke, Stine og Mie fandt værdi i Blockly som et værktøj, der kan hjælpe dem med at huske samt anvende syntaks korrekt. Ligeledes brugte de aktivt to-vejs synkroniseringen til at identificere og rette fejl tidligt i processen, hvilket gjorde, at de fandt debugging mindre frustrerende. De forskellige deltagere interagerede forskelligt med Blockly, hvor en enkelt deltager ligeledes helt undlod at benytte sig af det. Tre af deltagerne oplevede, at to-vejs synkronisering medvirkede til både at tage fejl løbende og til en mindre frustrerende fejlsøgningsproces. Her fremhævede Rikke, som anvendte Blockly som et referencepunkt for hendes tekstbaserede kode, at hun ligefrem fik små *"...Du gjorde det rigtigt-følelser"* når hun lykkedes med at generere blokke. Flere af deltagerne fremhævede dog, at de stadig havde oplevet situationer, hvor de var nødsaget til at tage andre værktøjer i brug. Dette mente de skyldtes, at VizCode hjalp dem med at lokalisere en fejl, men ikke altid *hvilken* fejl.

9.2 Nye Løsninger og AI

Vi observerede ikke deltagerne anvende ChatGPT eller anden kunstig intelligens. Alle deltagere forklarede dog, at det er noget, de dagligt anvendte i CS50-øvelserne. Vi forklarede det med, at de var opmærksomme på testens formål og i god tro ikke anvendte ChatGPT af den grund. AI er en overvejelse, vi tidligere har haft, men som ikke er repræsenteret i dette projekt. I kapitlet *Diskussion* diskuterer vi netop dette aspekt af programmering.

Deltagerne havde en række idéer til funktionalitet, som VizCode kunne inkorporere i fremtiden. Både Stine og Mads refererede til funktionalitet, der hjalp med bl.a. CS50-forelæsningen, filhåndtering og generelt funktionalitet rettet mod at kunne bruge VizCode til flere aspekter af CS50-forløbet. Vi diskuterede netop dette aspekt i kapitlet *Visuel Programmering*, hvor vi undersøgte alternativer til problemet i Essences Problem Scenario.

Deltagerne havde ligeledes en række forslag til funktionalitet, vi endnu ikke selv havde overvejet. Her tænkes eksempelvis på Mads' forslag om at bruge vinduerne i VizCode mere kreativt til eksempelvis webbrowsing, så man ikke skal skifte fane for eventuelt at følge med i en instruktionsvideo. I vores forløb havde vi kun behandlet vinduerne i VizCode, som kan ses i kapitel *Design* figur 7.1, som værende vinduer, hvori forskellige repræsentationer kunne befinde sig. Det var derfor en øjenåbner for os, at flere elementer i både programmering og CS50 faktisk egner sig til at blive præsenteret side om side. Om end vi ikke nåede videre med dette i projektet, har vi gemt idéerne herfra til vores videre arbejde med VizCode udover rammerne i dette projekt.

Diskussion

10.1 Resultat af evaluering

Overordnet var vi tilfredse med de tidlige indikatorer for, hvordan VizCode hjalp de studerende. Om end designet af vores test og naturen af vores data ikke nødvendigvis fordrer fund af højeste validitet. I dette afsnit diskuterer vi fundene fra vores evaluering, hvorefter vi indgår i en diskussion af kvaliteten af vores undersøgelsesdesign og data. Her forsøger vi både at nå til en konklusion om, hvorvidt vores løsning kan løse nogle af de problemstillinger, vi har afdækket i løbet af processen, og dermed udgøre et støtteværktøj til CS50, som beskrevet i vores problemformulering. Eftersom vores arbejde med VizCode fortsætter ud over rammerne i dette projekt, søger vi ligeledes at informere vores videre arbejde med produktet.

10.1.1 Støtteværktøj til CS50

På baggrund af vores to features, *Visualiseringsstrategier* og *To-vejs synkronisering*, havde vi fremstillet to value-hypoteser. Hvor disse ikke dækker over alle de features, vi har foreslået i dette projekt, så udgør de en væsentlig del af vores løsning til at besvare problemformuleringen. I dette afsnit diskuterer vi de resultater, vi opnåede i forbindelse med disse to hypoteser. Her vil vi ligeledes forsøge at inddrage relevant teori.

Hypotese 1: ***De studerende vil værdsætte muligheden for at tilgå deres CS50-program på et højere abstraktionsniveau***

I undersøgelsen af denne hypotese blev vi opmærksomme på hypotesen selv. Hvad mener vi med *værdsætte*? Hvilken værdi håber vi at kunne tilbyde? Er hypotesen falsificerbar? I formuleringen af denne hypotese blev det klart, at vi måske ikke havde konkretiseret præcis, hvad vi egentlig håbede at opnå ved at lade de studerende tilgå programmet på et højere abstraktionsniveau. I formuleringen anvendes *højere abstraktionsniveau* også som en generel betegnelse for konceptet *Visualiseringsstrategier* og i undersøgelsen specifikt *Blockly*. De forskellige visualiseringsstrategier varierer ligeledes på hvilket abstraktionsniveau de opererer. Derfor blev vi særligt opmærksomme på, at svaret på spørgsmålet *Vil de studerende værdsætte at kunne tilgå deres CS50-program på et højere abstraktionsniveau?* i høj grad afhænger af *hvilken* visualiseringsstrategi vi anvender til dette formål, og *hvilket* abstraktionsniveau koden præsenteres på. Her har vi tidligere været inde på begrebet om *Scaffolding* og Vygotskys *Zone for nærmeste udvikling*. Her vil en visualiseringsstrategi med al sandsynlighed værdsættes mere,

hvis dét, den assisterer med, er passende for det sted i læringsprocessen, brugeren befinder sig, men kan være overflødig før og efter dette punkt.

Dette medførte, at vi både i formuleringen af vores spørgsmål og i den efterfølgende analyse faktisk kunne blive i tvivl om, hvorvidt udsagn understøttede hypotesen eller ej. I fokusgruppen fandt vi, at alle deltagerne var begejstrede for Blockly som værktøj i deres øvelser, på nær en. Her fremhævede tre af deltagerne, at Blockly var nemmere at læse og derfor var en værdsat tilføjelse. Vi besluttede os dog for, at vores grundantagelse *De studerende vil værdsætte at kunne tilgå deres CS50-program på et højere abstraktionsniveau* ikke var sigende for det, vi søgte at finde ud af. Vi dykkede derfor ned i konceptet *Visualiseringsstrategier* for bedre at afdække, hvad det egentlig er, vi antager med dette bidrag. Dette blev til en revideret antagelse, der med fordel kan omsættes til fremtidige, falsificerbare hypoteser.

> Hypotese 1: *De studerende vil værdsætte at kunne tilgå deres CS50-program på et højere abstraktionsniveau, fordi det mindsker den kognitive belastning, der opleves i introduktionen af et nyt, komplekst fag som programmering og Computer Science.*

Hvis vi kigger på, hvordan de studerende benyttede sig af Blockly, var brugen meget forskellig.

1. **Mie** anvendte primært Blockly, især fordi hun havde problemer med at huske korrekt syntaks.
2. **Stine** anvendte en fifty/fifty tilgang til Blockly og C.
3. **Rikke** anvendte primært tekstbaseret kode, men brugte Blockly som en konstant spejling af hendes program.
4. **Mads** anvendte udelukkende tekstbaseret kode og fandt Blockly støjende.

Med dette menes særligt hvorvidt en deltager aktivt lavede blokke i højre side af VizCode eller om de kun skrev tekstbaseret kode. Til dette fænomen frembringer vi to væsentlige diskussionspunkter, der er relevante at undersøge videre. Et af dem er brugerens *niveau*, i.e. hvor langt de er i forløbet og hvor dygtige de er til programmering. Et andet er idéen om, at mennesker har forskellige måder at lære på.

I forhold til de studerendes niveau kan vi benytte os af vores empiri til at frembringe en pointe. Mads har mere erfaring, og vi kunne ved øjensyn konstatere, at han ligeledes havde ret godt styr på tekstbaseret programmering generelt. Mie derimod beskrev sig selv som noget mere uerfaren og anvendte Blockly meget mere end andre deltagere.

Hvis man behandler Blockly som en *Visualiseringsstrategi*, der skal agere *Scaffold* ud fra Vygotskys *Zone for nærmeste udvikling*, er de to datapunkter i overensstemmelse med dette. Her er Mie på et tidligt stadie i programmering og kan anvende Blockly som støtte til at tilegne sig kompetencer, der

bygger videre på sine egne. Mads derimod har hverken brug for støtten og finder den i stedet til hindring for hans arbejde. Her er Blockly blevet overflødigt og fungerer måske mest af alt som støttehjul til en person, der allerede har lært at cykle.

Et andet perspektiv kunne være at inddrage forskellige teorier som VARK af Niels Fleming, der beskriver fire forskellige læringsstile (Fleming og Mills 1992).

1. **Visuel (V)**, hvor personer foretrækker at lære gennem billeder, diagrammer og andre visuelle hjælpemidler.
2. **Auditiv (A)**, hvor personer foretrækker at lære gennem forelæsning og diskussioner ved at lytte.
3. **Læsning (R)**, hvor personer foretrækker at lære gennem læsning og skrivning af tekst.
4. **Kinæstetisk (K)**, hvor personer foretrækker at lære gennem hands-on aktiviteter og praktiske eksempler.

Her kan deltagernes interaktion med VizCode i stedet forklares med, hvilken læringsstil de hver især foretrækker. Mie kunne foretrække at lære gennem det visuelle aspekt af Blockly, hvor informationen er præsenteret i form af et visuelt hjælpemiddel. Mads derimod kunne foretrække at lære primært gennem læsning (R), hvor C-koden udgør netop dét at skrive og læse. Således kan vi også frembringe den pointe, at diversiteten i hvordan VizCode blev anvendt, kan understøtte ideen om, at VizCode kan supportere flere forskellige læringsstile, i hvert fald hvis Mads i fremtiden kunne få lov til at lukke for Blockly.

Vi konkluderede på baggrund af denne diskussion, at vores resultater kunne bruges som en tidlig indikation på, at konceptet med *Visualiseringsstrategier* i form af Blockly fungerede efter hensigten. Med dette menes, at vi anså VizCode for at have ydet støtte til deltagerne, særligt til de deltagere der var mest uerfarne. VizCode blev af nogle deltagere anvendt, som vi havde regnet med, hvor Blockly tillod deltagerne at programmere med kodeblokke i situationer, hvor deres syntaksfærdigheder ikke var tilstrækkelige.

> Hypotese 2: *VizCode vil hjælpe den studerende med at finde og rette syntaksfejl i deres program, ved at de kan modtage øjeblikkelig visuel feedback, mens de programmerer.*

Det primære fund fra vores tematiske analyse var, at To-vejs konvertering ændrede måden, deltagerne normalt programmerede på med hensyn til syntaks. Tre af deltagerne forklarede, at To-vejs synkronisering gav dem mulighed for at tage syntaksfejl løbende ved at spejle deres kode i Blockly. Alle tre deltagere forklarede, at dette gjorde fejlsøgning nemmere. Her forklarede de ligeledes, at det sandsynligvis ikke gjorde antallet af fejl mindre.

Mads nævnte dog, at han ikke fandt det behjælpeligt til at fejlfinde, og at han foretrak at bruge fejlmeddelelser fra konsollen og aflæse dem. Mads har, som nævnt tidligere, en smule mere erfaring med koden og fortæller, at han allerede har en indlært praksis, når det kommer til at debugge.

To-vejs synkroniseringen er den feature, der bygger direkte på Paperts konstruktionisme. Her tænker vi særligt på at manipulere et objekt og se en øjeblikkelig, fysisk (i vores eksempel, digital) feedback. Deltagerne fremhævede særligt to pointer. Først og fremmest beskrev de, at dét at tage fejlprocessen løbende fjernede risikoen for at få én stor bunke af fejlmeddelelser. Dette fænomen kender vi selv til, hvor, som Mie beskriver det, "...nogle gange kan man få så mange fejl, at man bare ikke gider til at ordne dem."Den øjeblikkelige feedback af deres handlinger i VizCode har tilsyneladende den effekt, at deltagerne stoppes op, som Stine beskriver det, "okay, det er lige her, hvor jeg er nu, der er noget galt."

En anden pointe ved synkroniseringen er, at deltagerne ligeledes havde mulighed for at anvende Blockly-blokke som skabelon. Dette var et af de punkter, vi var nervøse for, eftersom vores MVP-test af *Flowgorithm* viste, at der var en risiko for, at brugeren kun orienterer sig mod det visuelle programmeringssprog. Dette argumenterede vi for ikke var hensigtsmæssigt i et introduktionskursus til tekstbaseret programmering. Vi kan ikke sige, om det fænomen kan være et problem med Blockly. Hvor ingen af deltagerne udviste den adfærd, er vi bevidste om, at deltagerne kendte til formålet med vores produkt. Det er derfor en sandsynlighed, at alle deltagere bevidst forsøgte at anvende Blockly som støtteværktøj til tekstbaseret programmering og ikke som en erstatning.

I analysen belyste vi også aspekter af vores to-vejs synkronisering, vi ikke havde overvejet. Hvor de studerende fandt to-vejs synkroniseringen anvendelig, fremhævede de alle, at den primære indikator for fejl i koden var, at blokkene ikke blev genereret. De studerende modtager altså ikke en fejlbesked eller lignende, før programmet kompileres. Her forklarede de, at det i nogle tilfælde var fint, da viden om, at en fejl eksisterer, allerede er en hjælp. Mads fremhæver dog, at det derefter er op til deltageren selv at finde ud af *hvilken* fejl, der så er tale om. Alle deltagere var enige om, at fejlmeddelelse eller anden assistance i den situation kunne give endnu mere.

Vi vil *gerne* have de studerende til at opleve fejl i deres programmering. Fejlsøgning er et kritisk komponent på niveau med algoritmisk tænkning, dekonstruktion m.m. i Computational Thinking (Denning og Tedre 2019). De studerende lærer, hvordan computeren fungerer og udvikler deres problemløsningsevner ved at fejlsøge, hvorfor deres program ikke fungerer (Denning og Tedre 2019). Derfor ser vi positivt på, at VizCode ikke nødvendigvis reducerer mængden af syntaksfejl, men letter belastningen ved i stedet at tage fejl løbende.

Vi konkluderede på baggrund af fokusgruppeundersøgelsen, at der var indikatorer for, at to-vejs synkronisering var en hjælp til de studerende. Ved at håndtere fejlsøgning som en løbende proces, oplevede flere af de studerende, at fejlsøgning blev lettere og mindre frustrerende. Den øjeblikkelige

feedback som beskrevet af Papert 1980 synes at være forenelig med et koncept som VizCode. Vi vil dog arbejde videre med undersøgelser af, hvilken assistance brugeren skal ydes, da de lige nu er overladt til sig selv, når de bliver bevidste om en fejl. Her menes, at fejlmeddelelser relateret til syntaksen først præsenteres, når programmet eksekveres.

10.1.2 Diskussion af problemformulering

De to value-hypoteser udgør en væsentlig del af vores problemformulering *"Kan en kombination af grafiske og tekstuelle repræsentationer, i et introduktionskursus som CS50, støtte indlæringen på to måder: Computational thinking og Programmeringsbegreber?"*. Derfor argumenterer vi ligeledes for, at vores fund er indikatorer for at Visualiseringsstrategier og To-vejs synkronisering kan agere støtte i indlæringen af Computational thinking og programmeringsbegreber.

Vi anerkender dog et væsentligt problem i evalueringen af vores produkt, i relation til vores problemformulering. Efter analysen diskuterede vi, at vi kan argumentere for at deltagerne havde en anderledes, positiv oplevelse med fejlsøgningen uden de frustrationer vi havde identificeret i brugerundersøgelsen. Med undtagelse af en enkelt deltager, der ikke anvendte Blockly. Ligeledes lavede deltagerne stadig en række fejl, og indgik derfor stadig i *debugging* der er en væsentlig del af Computational Thinking. Her opstår så en ny række spørgsmål: Hvis det bliver lettere og hurtigere at finde fejl, læres der så stadig lige så meget ved at rette fejlene? Vil en deltager som Mie, der anvender Blockly meget, på et tidspunkt skifte til den tekstuelle repræsentation?

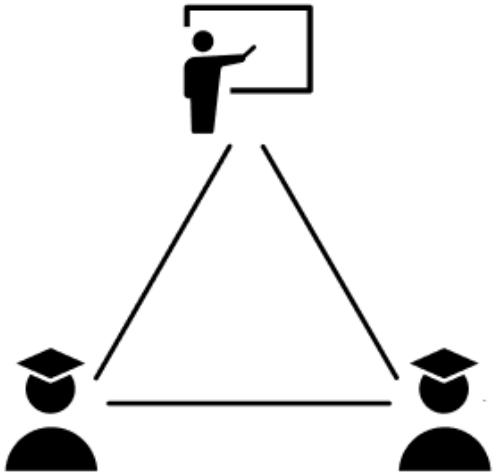
Vi anså dette for en naturlig udvikling i videre forståelse af problemet på baggrund af ny viden. Vi forsøger at løse et problem, og skaber en ændring i problemet, der måske skal skabe en ny ændring i vores løsning. Derfor argumenterede vi for følgende: Visualiseringsstrategier og To-vejs synkronisering synes, på baggrund af fundende i vores evaluering, at afhjælpe en række af de frustrationer relateret til syntaksfejl vi identificerede i brugerundersøgelsen. Ud fra teorierne om Dual Coding og Paperts konstruktionsisme, burde de studerende ligeledes få et højere udbytte af øvelserne. Deltagerne indgik i en ny, løbende fejlsøgningsproces hvor flere af deltagerne bragte positive tilbagemeldinger om dette. Vi observerede ligeledes deltagerne anvende Blockly efter hensigten, hvor særligt Rikke anvendte det som en *ekstra, visuel repræsentation* hvori hun kunne spejle den tekstuelle repræsentation. VizCode kan derfor støtte indlæringen af Computational Thinking og Programmeringsbegreber, ved at afhjælpe frustrationerne i fejlsøgningsprocessen og tilbyde de studerende hyppig, øjeblikkelig feedback på deres handlinger.

Disse features udgør dog kun en lille del af vores samlede, foreslåede capabilities. Og vores problemformulering fokuserer ligeledes kun på et udsnit af et helt CS50-forløb. Vores løsning introducerer ligeledes en ny måde at programmere på. Vi er derfor særligt varsomme med at konkludere vores bidrag, særligt med henblik på læring over tid. Eksempelvis om Mies interaktion med Blockly er hensigtsmæssig, fordi den gør det nemmere for hende at færdiggøre øvelserne. Eller om den er

uhensigtsmæssig, fordi hun får valget om at abstrahere helt fra syntaksen.

10.2 Til- og fravalg igennem processen

Vi har hovedsageligt koncentreret os om at identificere og adressere udfordringerne for individuelle studerende. Om end vi har foreslået en række features til samarbejde, har vi ikke synderligt inkluderet underviserens rolle. I denne sammenhæng har vi anerkendt betydningen af samarbejde og pairprogramming, men vi har ikke i tilstrækkelig grad undersøgt de komplekse dynamikker, der opstår i en læringstriade bestående af underviser, studerende og medstuderende.



Figur 10.1: Læringstriade

En anden interessant vinkel på at opnå bedst mulige vilkår for, at nye studerende kan lære programmering og computational thinking, kunne derfor være at have fokuseret på forholdet mellem den studerende og medstuderende, eller mellem den studerende og underviseren. Ifølge Johnson, Johnson og Smith 2007 er samarbejde mellem underviser og studerende afgørende for at skabe et effektivt læringsmiljø. Deres studie fremhæver, at læringsrelationer kan forbedre de studerendes akademiske præstationer og sociale færdigheder gennem struktureret interaktion og feedback. Denne tilgang er allerede anvendt på AAU, og projektarbejde fordrer, at man arbejder sammen, og ofte i større grupper, hvorfor det også kunne være interessant at tilpasse en løsning specifikt til denne konstellation (Aalborg University 2024). Her kan der trækkes tråde tilbage til de problemstillinger, der blev beskrevet i vores brugerundersøgelser, hvor teknologier som Git (versionsstyring, der anvendes af grupper til at skrive kode i det samme stykke software) skabte en lang række tekniske udfordringer.

Hvis vi havde valgt at fokusere på triaden, kunne vi have opnået en dybere forståelse af, hvordan interaktionerne mellem underviser og studerende samt mellem studerende indbyrdes påvirker læringsprocessen, og hvordan en løsning kunne facilitere netop dette. Vi har i løsningen idegenereret

over samarbejde, men kun i en kontekst af, at forskningen viser, det er en effektiv læringsmetode. Roller i form af underviser og studerende har derfor haft minimal indflydelse på vores valg af capabilities til løsningen. De blev kort introduceret i kapitel 6, da det var en del af vores idegenereringsproces. Flere læringsteorier understreger ellers den sociale interaktion, hvor vi eksempelvis i kapitel *Related Work* formulerede det som en af styrkerne ved et fysisk design.

Der er dog også udfordringer ved at fokusere på triaden. Det kræver flere ressourcer i form af tid og interviews at inkorporere underviserens perspektiv i en løsning. Vores valg har som nævnt tidligere været på baggrund af *bird-in-hand* princippet, og vi valgte derfor denne vinkel fra, da vi så et samarbejde med undervisere som værende mere ressourcekrævende. Selvom vi har valgt at fokusere på individuelle studerende og til dels pair-programming, er det vigtigt at anerkende de potentielle fordele ved at undersøge forholdene i triaden og i særdeleshed underviserens rolle.

10.2.1 Valg og fravalg

Vores nuværende fokus på individuelle læringsstrategier og pair-programming er baseret på vores erfaringer og brugerundersøgelser. Vi har prioriteret at udvikle et visualiseringsværktøj, da disse adresserer de udfordringer, de studerende og vi stod overfor. Dog anerkender vi, at der er betydelige fordele ved at undersøge og integrere dynamikkerne i triaden mellem undervisere, studerende og medstuderende. Ved at balancere individuelle og sociale læringsaspekter kunne vi skabe en mere omfattende læringsoplevelse, der bedre understøtter de studerendes udvikling og engagement.

10.2.2 Læring som et forløb

Vores nuværende løsning fokuserer mest på at løse specifikke udfordringer, som studerende står overfor her og nu, såsom syntaksfejl, tekniske udfordringer og feedback-gennemsigtighed. Dette øjeblikkelige fokus kan være effektivt til at løse aktuelle problemer, men det tager ikke nødvendigvis højde for den overordnede læringsproces, som strækker sig over et helt undervisningsforløb. Vi diskuterede dette, da vi anvendte Problem Scenarios i kapitlet *Visuel programmering* til at situere VizCode i specifikke situationer versus generelle forløb.

I vores evaluering bragte deltagerne ligeledes dette op, hvor denne prioritering af funktionalitet fra vores side blev diskuteret. Her bragte deltagerne det på banen, at de sagtens kunne anvende det til specifikke øvelser, men vores produkt manglede essentielle ting for dem, hvis det skulle erstatte eksempelvis CS50. Her var tale om bl.a. basisfunktioner som at kunne gemme og loade projekter, men også funktioner vi ikke selv havde overvejet. Vi anser dette for at være en væsentlig tilføjelse til VizCode, vi vil arbejde med når specialet afsluttes.

10.2.3 Alle bruger AI

Vi vidste det godt selv - men fokusgruppen bragte det også op. ChatGPT anvendes i høj grad i programmering. I diskussionerne af ChatGPT i vores løsning overvejede vi nøje, hvordan AI kan anvendes til effektiv debugging uden at det resulterer i misbrug. Ved at integrere AI direkte i brugerfladen forestillede vi os, at ChatGPT kunne assistere de studerende med at identificere og korrigere kodefejl, hvilket minimerer behovet for at søge hjælp eksternt. Ayala-Pazmiño 2023 konkluderer på baggrund af et litteraturreview kaldet "Artificial Intelligence in Education: Exploring the Potential Benefits and Risks" at AI kan have flere uddannelsesmæssige fordele:

- **Kontekstforståelse:** AI kan analysere den kode, som de studerende arbejder på, og give kontekstspecifik feedback. Dette betyder, at forslag til korrektioner og forklaringer er direkte relevante for de problemer, studerende står overfor. Denne målrettede assistance kan effektivisere læringsprocessen og gøre den mere personlig.
- **Real-time debugging:** Ved at integrere ChatGPT direkte i et udviklingsmiljø kan de studerende få øjeblikkelig feedback på syntaks eller logiske fejl i deres kode. Dette kan hjælpe med at opretholde et flydende læringsflow og reducere den tid, det tager for studerende at forstå og rette deres fejl, hvilket kan være behjælpeligt i forhold til at lære nye begreber og ikke sidde fast i opgaver, og derfor miste motivationen.
- **Undgå afhængighed af AI:** Ved at begrænse AI's rolle til at foreslå løsninger snarere end at levere komplette svar opmuntres de studerende til selv at tænke igennem problemerne. Dette vil derfor understøtte udviklingen af problemløsning og forbedre deres evner inden for computational thinking.
- **Forbedring af selvstændighed:** Ved at give de studerende et værktøj til selv at løse deres problemer kan deres selvstændighed styrkes og tillid til egne evner øges, hvilket Ayala-Pazmiño 2023 mener kan være afgørende for at udvikle robuste programmeringsfærdigheder.

Dog understreger Ayala-Pazmiño 2023 også betydelige risici, herunder muligheden for snyd og afhumanisering af læringsoplevelser. I sammenhæng med din fejlfindingsfunktion kan AI tilbyde en nuanceret tilgang til løsning af programmeringsfejl ved at foreslå korrektioner og lære af elevernes interaktioner. Denne proaktive assistance kunne accelerere læring og forståelse. Alligevel er der risiko for, at eleverne bliver for afhængige af AI, hvilket potentielt kan underminere deres læreproces ved ikke at engagere sig dybt i problemløsningsaktiviteter. For at afbalancere dette er det afgørende at designe AI-funktionen til at fremme engagement og computational thinking, hvilket vil bidrage til, at studerende forstår begrundelsen bag forslagene og assistancen som AI giver (Ayala-Pazmiño 2023).

Endvidere understreger Ayala-Pazmiño 2023 vigtigheden af etiske overvejelser og behovet for en afbalanceret integration af AI i uddannelsesmæssige læringsmiljøer som vores. Dette inkluderer forberedelse af studerende til en fremtid, hvor AI-værktøjer er almindelige, samtidig med at der lægges vægt på den uerstattelige værdi af menneskelig dømmekraft og interaktion. Undervisere og vores design vil derfor have et ansvar for, at AI supplerer snarere end erstatter de menneskelige elementer i undervisningen, og skaber et miljø, hvor teknologien understøtter uddannelse uden at formindske læringen (Ayala-Pazmiño 2023).

10.3 Essence og Effectuation

Som afslutning af denne rapport vil vi diskutere nogle af de nævneværdige konsekvenser, vi mener skyldes vores valg af strategi og forberedelse. Vi vil ligeledes bringe eksempler på, hvordan Essence har bragt værdi til vores proces, og give et eksempel på, hvor vi ikke fandt Essence anvendeligt. Dette gør vi for at informere forfatterens fremtidige arbejde, da Essence som tidligere beskrevet stadig er under udarbejdelse.

10.3.1 Brugerundersøgelse og empiri

Én klar afvigelse fra vores tidligere projekter er måden, vi har håndteret research af problemområdet. Vores mål var at have et produkt i en kvalitet, der gav mening at aflevere til fremtidige studerende. Vi diskuterede i begyndelsen af dette projekt tre forskellige måder at udvikle på:

1. Vi kunne investere en masse ressourcer i research og afdækning af brugernes behov. På den måde kunne vi indsamle en masse *valide* indsigter, foreslå et design og se, hvor meget vi kan nå at implementere.

2. Vi kunne implementere et design uden at investere ressourcer i undersøgelser, og i stedet håbe på, at vores egne erfaringer var dækkende. Herefter kunne vi præsentere det for brugeren og håbe på, at vi havde ramt rigtigt.

3. Vi kunne forsøge at gøre begge ting samtidigt, alt efter hvad der gav mening og forhåbentlig nå at afprøve en masse elementer.

Om end vi formulerede det i hverdagssprog, så er det udtryk for forskellige tankegange, vi kender fra tidligere projekter og fra tidligere fag. Mulighed nummer 1 er den fremgangsmåde, vi typisk har anvendt i projekter på AAU. Her har vores erfaring dog været, at projektet ofte ender ud i demonstrerbare prototyper og en masse viden. Grundet de store mængder af ressourcer investeret i processer som desk research og brugerundersøgelser, er vores erfaring dog, at andre kerneelementer af design, som afprøvning af prototyper, er svære at nå omkring på grund af manglende tid. Processerne bliver ligeledes lineære og sekventielle, som i en Vandfalds-model (Martin 2024), og eventuelle *pivots* eller

afvigelser fra den originale problemstilling på baggrund af ny information eller nye muligheder kan være besværligt. Dette er beskrevet i det indledende kapitel *Motivation*.

I dette projekt blev brugerundersøgelser vendt på hovedet i forhold til vores normale fremgangsmåde. Her undersøgte vi først vores egne baggrunde og foreslog en række bidrag, hvorefter vi forsøgte at anvende disse i dialogen med brugerne. Det primære store gode ved dette skulle være at nå at afprøve flere elementer af vores produkt samt nå længere i implementeringen af selvsamme. I reflektionen af dette har vi kun os selv at sammenligne med. I forhold til afprøvningen af features i vores løsning er vi ikke nået så langt, som vi håbede. Her har vi foretaget to MVP-tests af Visualiseringsstrategier samt en evaluering af vores implementerede produkt, hvor vi ligeledes afprøvede to-vejs synkronisering.

I implementeringen af vores produkt har vi nået mere, end hvad vi håbede på. Vores produkt er udviklet efter *best practice* i det omfang, vi har kompetencerne til. Intet af funktionaliteten er efterlignet for at kunne demonstrere en feature, men eksisterer i stedet som fuldt implementerede funktionaliteter, der kan anvendes med det samme. Vores produkt indeholder én visualiseringsstrategi, der fungerer efter hensigten, samt to-vejs synkronisering. Vi har foruden dette inkluderet en række standardfunktioner som *Login*, *Kompilering*, *Eksekvering* og opsat servere, hvorpå vores applikation på nuværende tidspunkt hostes. Vi har implementeret sikkerhedsforanstaltninger som Authentication, Authorization, Containers til safe-code execution og meget mere. Med andre ord - vores produkt er faktisk klar til release.

Havde vi ikke prioriteret dette, havde vi måske i stedet haft mulighed for at afprøve flere elementer af vores design. Vi har i løbet af projektet diskuteret, hvordan vores prioriteter og deraf også vores fremgangsmåde har ændret sig undervejs. Vi påbegyndte processen ud fra et Effectuation, Essence og abduktivt ståsted, i.e. hvad vi karakteriserer som værende fremgangsmåde nummer 3, hvor der handles hurtigt, afprøves og hvor målet udvikler sig undervejs. Efter vores indledende capabilities med Visuel programmering og to-vejs synkronisering blev foreslået, skiftede vi fremgangsmåde til nummer 3, hvor vi placerede fokus på at få skabt produktet og afprøvet det bagefter.

10.3.2 Essence: Problem-Solution Modeling Generation og vores proces

Undervejs i processen har vi gjort brug af Essence til forskellige ting. I dette afsnit fremhæver vi områder, hvor vi fandt Essence særligt anvendeligt, og giver ligeledes et eksempel på, hvor vi fandt et af værktøjerne mindre anvendeligt.

Fulcrums

Påbegyndelsen af dette projekt ud fra Essences *Capability*-fulcrum var for os en helt ny måde at begynde på. Vi fandt det udfordrende at analysere, hvad vores egne *Leverage points* kunne være, særligt da Essence ikke er en designmetodologi, vi har været vant til. Her tænkes også på sammenfatningen af Essence og Effectuation, hvor principperne fra Sarasvathy ligeledes repræsenterer en fremgangsmåde, vi er uvante i.

Om end *Capability*-fulcrum denoterede, hvor vi påbegyndte processen, argumenterer vi for, at det i høj grad har formet vores bidrag. Herunder også, at det har været meget foreneligt med en Effectuation-strategi. Om end det kan virke arbitrært, udgjorde den indledende analyse af vores *Leverage points* et grundlag, hvorpå mange af de essentielle valg i processen blev taget. Her tænker vi eksempelvis på skiftet fra *Teknologiforståelse* til *CS50*, hvor vores egne erfaringer udgjorde en fordel. Vores idé til to-vejs synkronisering stammer oprindeligt fra en diskussion af, hvordan vores løsning kunne tilbyde noget, ingen andre kunne i form af et *Leverage point*, hvorefter idéen ligeledes viste sig særligt forenelig med Papert. På den baggrund har vi fundet Essences beskrivelse og anvendelse af *Capability*-fulcrums og *Leverage points* anvendelige og udslagsgivende i dette projekt.

Problem Solution Canvas og byggeblokkene

Vi anvendte PSC løbende i processen, men hovedsageligt kerneområderne *Situation* og *Contribution*. Om end det ikke er repræsenteret i denne rapport, havde vi ofte forskellige, simultane udfyldte PSC'er, der hver repræsenterede et alternativt system. Eksempelvis under udfærdigelsen af features inden for visuel programmering havde vi sideløbende et PSC, hvor vi forsøgte at repræsentere en løsning, der tog højde for aspektet *Generelle forløb* som beskrevet i vores Problem Scenario.

Koblingen mellem manifestationer og capabilities har vi fundet værdifuld. Igennem projektet har det skabt et fokus på at konkretisere ikke bare vores samlede bidrag, men også delelementerne i vores bidrag. Vi har ligeledes fundet det værdifuldt at fokusere på det *essentielle* i vores bidrag i form af vores capabilities. I Problem Solution Canvas udfyldte vi ikke alle elementer fra vores løsning såsom *Login*. Her har PSC i stedet hjulpet os til at fokusere på præcis de features, vi mener løser problemerne, hvilket vi mener har bidraget til en bedre forståelse af vores produkt som helhed. Vi har primært anvendt Aaens beskrivelse af kerneområderne *Situation* og *Contribution* i dette projekt, da disse var særligt forenelige med vores proces indtil nu. I kerneområderne *Solution* og *Valuation* har Aaen dog dækket en række af de problematikker, vi står overfor nu, såsom at få defineret klare kriterier på forskellige niveauer, såfremt vi vil påbegynde afprøvning af den næste række af features.

10.3.3 Essences scenarier

Vi har anvendt Problem-, Leverage- og Prospect-scenarier igennem dette projekt. Særligt Problem-scenarier har vi anvendt en del, da dette har givet os mulighed for at kunne inspicere forskellige alternativer til problemet. Igennem projektet har vi fået en forståelse for, at problemet ændrer sig alt efter hvilken situation, du forsøger at løse, og hvem du forsøger at designe en løsning til. Et eksempel på dette er, hvordan vi i lang tid uvidende har negligeret underviserens rolle. Vi brugte vores egne erfaringer, og ingen af os anvendte underviseren i vores kursus til noget formål. Først da vi anvendte Problem-scenarier til at belyse alternativer til problemet, indså vi, at underviserens perspektiv kan være meget værdifuldt at inkludere.

Vi havde udfordringer med at anvende Prospect-scenarier til at belyse forskellige løsninger i dette projekt. Vi anvendte det i udfærdigelsen af pair programming, men havde udfordringer med at håndtere den kompleksitet, der opstod ved at inkludere elementerne fra både Problem- og Leverage-scenarier. Her kan vi nævne, at vi anvendte en lignende struktur i kapitlet *Related work*, hvor vi netop var inspireret af Aaens Prospect-scenarier. Vi fandt dog først dette anvendeligt, da vi abstraherede fra en række af elementerne.

Konklusion

I dette projekt søger vi at besvare problemformuleringen "*Kan en kombination af grafiske og tekstuelle repræsentationer, i et introduktionskursus som CS50, støtte indlæringen på to måder: Computational thinking og Programmeringsbegreber?"*

Til dette har vi udviklet et web-baseret, visuelt udviklingsmiljø hvor den studerende præsenteres for en kombination af grafiske og tekstuelle repræsentationer. Dette koncept bygger på Dual Coding af Paivio, der forklarer at mennesker lærer bedst når de præsenteres for information i både den verbale og den visuelle kanal (Clark og Paivio 1991). I vores design har vi implementeret To-vejs synkronisering, der skal emulere principperne indeholdt i Paperts konstruktionisme (Papert 1980). Her lærer mennesker bedst, når de konstrurer viden ved at manipulere fysiske objekter, og modtager øjeblikkelig, fysisk feedback. Vi har emuleret dette koncept digitalt ved brug af det visuelle programmeringssprog Blockly.

Vi har baseret dette på vores egne udfordringer i CS50, såvel som brugerundersøgelse af 21 studerende ved Digitalisering og Applikationsudvikling. Her har vi anvendt en række design-thinking aktiviteter, Essence og Sarasvathys effectuation til at foreslå et bidrag til en løsning, baseret på de kompetencer vi selv besidder (Sarasvathy 2001) (Aaen 2024).

Vi har foretaget to MVP test, som beskrevet af Eric Ries, af Flowgorithm og Cake-Core til at teste vores koncept (Ries 2011). Vi har ligeledes evalueret vores produkt i en Usability test og en efterfølgende fokusgruppe session. På baggrund af en deduktiv, tematisk analyse baseret på to value-hypoteser, der skal teste vores grundlæggende antagelser omkring vores produkt (Brandi og Sprogøe 2019). Her fandt vi, at Visualiseringsstrategier og To-vejs synkronisering afhjalp en række frustrationer vi havde identificeret i vores brugerundersøgelser. VizCode introducerede lige ledes en ny måde at programmere på. Her tænkes på fejlsøgningsprocessen, der i VizCode synes at præsentere sig selv som en kontinuerlig, mere overskuelig proces.

Vi diskuterer, at vores fund kan agere tidlige indikatorer for at Visualiseringsstrategier i.e kombination af tekstuelle og grafiske repræsentationer, samt to-vejs synkronisering, kan agere støtte i indlæringen af computational thinking og programmeringsbegreber. Her beskriver vi dog også en række usikkerheder relateret til vores undersøgelsesdesign, samt en lang række perspektiver vi har diskuteret i løbet af processen som ikke er indeholdt i designet. På baggrund af dette, konkluderer vi følgende: VizCode synes at kunne afhjælpe en række udfordringer relateret til syntaksfejl i introduktionskursus som CS50. Videre undersøgelse er dog nødvendig for både at fastlægge hvorvidt dette er et validt fund.

Bibliografi

Atlassian. 2024. *Kanban.* Tilgået: 01-05-2024. https://www.atlassian.com/agile/kanban.

Ayala-Pazmiño, Mario Fabricio. 2023. „Artificial Intelligence in Education: Exploring the Potential Benefits and Risks". Tilgået: 16-05-2023, *593 Digital Publisher CEIT*, https://doi.org/10.33386/593dp.2023.3.1827. https://typeset.io/papers/artificial-intelligence-in-education-exploring-the-potential-3f4135p9?utm_source=chatgpt.

Baeldung. 2024. *KISS Software Design Principle.* https://www.baeldung.com/cs/kiss-software-design-principle. Tilgået: 24-05-2024.

Bennis, Warren og Burt Nanus. 1985. *Leaders: Strategies for Taking Charge.* New York: Harper & Row.

Bloks, Google: Project. 2016. *Project Bloks: Making code physical for young learners.* Tilgået: 18-04-2024. https://blog.google/outreach-initiatives/education/project-bloks-making-code-physical-for/.

Brandi, Ulrik og Jonas Sprogøe. 2019. *Det magiske øjeblik : kvalitativ analyse skridt for skridt.* Hans Reitze. ISBN: 9788741268675.

Buxton, Bill. 2010. *Sketching User Experiences: Getting the Design Right and the Right Design.* San Francisco, Calif: Elsevier Science. ISBN: 9780080552903.

Caspersen, Michael E., Ole Sejer Iversen, Mogen Nielsen, Arthur Hjorth og Line Have Musaeus. 2018. „Computational Thinking - hvorfor, hvad og hvordan?"

Clark, James M. og Allan Paivio. 1991. „Dual coding theory and education". *Educational Psychology Review* 3 (3): 149–170.

Communications, Zoom Video. 2024. *Zoom Video Communications Home Page.* https://zoom.us/. Tilgået: 18-04-2024.

cra16. 2016. *cake-core.* https://github.com/cra16/cake-core. Tilgået: 18-04-2024.

Creswell, John W. 2014. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches.* 4. udgave. Thousand Oaks, CA: SAGE Publications.

Denning, Peter J. og Matti Tedre. 2019. *Computational Thinking.* The MIT Press.

Design Sprints. 2023. *Components and Modules.* https://designsprintkit.withgoogle.com/methodology/phase3-sketch/crazy-8s. Tilgået: 14-04-2024.

Docker, Inc. 2024. *Get Started with Docker.* https://docs.docker.com/get-started/overview/. Tilgået: 24-05-2024.

Dybå, Tore og Erik Arisholm. 2005. „Are Two Heads Better than One? On the Effectiveness of Pair Programming". *Software Engineering Notes* 30 (5): 1–7.

Effectuation.org. 2024. *The Five Principles of Effectuation.* https://effectuation.org/the-five-principles-of-effectuation. Tilgået: 18-05-2024.

Figma. 2024. *About Figma.* https://www.figma.com/about/. Tilgået: 12-04-2024.

Fleming, Neil D. og Colleen Mills. 1992. „Not Another Inventory, Rather a Catalyst for Reflection". *To Improve the Academy* 11:137–155.

Flowgorithm. 2024. *Flowgorithm - Flowchart Programming Environment.* Tilgået: 21-04-2024. http://www.flowgorithm.org/.

Funk, Matthias G., Jose Manuel Cascalho, Ana Isabel Santos og Armando B. Mendes. 2021. „Educational Robotics and Tangible Devices for Promoting Computational Thinking". *Frontiers in Robotics and AI* 8:713416. https://doi.org/10.3389/frobt.2021.713416.

Google Developers. 2024. *Blockly: A Visual Programming Editor.* https://developers.google.com/blockly. Tilgået: 27-04-2024.

Grundskolen, Teknologiforståelse i. 2024. *Som Selvstændigt Fag i Indskolingen.* Tilgået: 21-04-2024. https://xn--tekforsget-6cb.dk/forlob/som-selvstaendigt-fag-indskolingen/.

Harvard University. 2023. *Problem Set 1: Mario (Less) - CS50x 2023.* https://cs50.harvard.edu/x/2023/psets/1/mario/less/. Tilgået: 21-03-2024.

———. 2024. *CS50: Introduction to Computer Science - Course Page.* https://pll.harvard.edu/course/cs50-introduction-computer-science. Tilgået: 14-05-2024.

Horn, Michael S., Erin T. Solovey, R. Jordan Crouser og Robert J.K. Jacob. 2009. „Comparing the Use of Tangible and Graphical Programming Languages for Informal Science Education". I *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 975–984. New York, NY, USA: ACM. https://doi.org/10.1145/1518701.1518851.

Hunt, Andrew og David Thomas. 1999. „The Pragmatic Programmer". Kapitel The Evils of Duplication, 20th Anniversary Edition, 27–30. Tilgået: 26-05-2024. Addison-Wesley.

IBM. 2024. *API - Application Programming Interface.* https://www.ibm.com/topics/api. Tilgået: 18-04-2024.

Johnson, David W., Roger T. Johnson og Karl A. Smith. 2007. „The State of Cooperative Learning in Postsecondary and Professional Settings". *Educational Psychology Review* 19 (1): 15–29. https://doi.org/10.1007/s10648-006-9048-1.

Kahoot. 2024. *Kahoot! Home Page.* https://kahoot.it/. Tilgået: 18-04-2024.

Kraaijenbrink, J. 2012. „The Nature of the Entrepreneurial Process: Causation, Effectuation, and Pragmatism". I *New Technology-Based Firms in the New Millennium, Volume IX,* redigeret af A. Groen, R. Oakey, P. van der Sijde og G. Cook, 187–199. Emerald Group Publishing.

KUBO Robotics. 2024. *Meet KUBO.* Tilgået: 18-04-2024. https://kubo-robot.com/meet-kubo/.

Kuhail, Mohammad Amin, Shahbano Farooq, Rawad Hammad og Mohammed Bahja. 2021. „Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review". *IEEE Access* 9:14181–14202. https://doi.org/10.1109/ACCESS.2021.3051043.

Larman, Craig. 2004. „eXtreme Programming (XP)". Kapitel 8 i *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* 137–171. Prentice Hall.

Lazar, Jonathan, Jinjuan Heidi Feng og Harry Hochheiser. 2017. *Research Methods in Human-Computer Interaction.* Kapitel 11, 299–322. Morgan Kaufmann. ISBN: 9780128093436.

Maloney, John, Mitchel Resnick, Natalie Rusk, Brian Silverman og Evelyn Eastmond. 2010. „The Scratch programming language and environment". *ACM Transactions on Computing Education (TOCE)* 10 (4): 16.

Martin, Robert C. 2024. *Principles of Object Oriented Design.* http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. Tilgået: 25-05-2024.

McDowell, Charlie, Linda Werner, Heather E Bullock og Julian Fernald. 2006. „The effects of pair-programming on performance in an introductory programming course". *ACM SIGCSE Bulletin* 34 (1): 38–42.

Michael S. Horn, Marina Bers, Jordan Crouser. 2012. „Tangible interaction and learning: The case for a hybrid approach". *Personal and Ubiquitous Computing* 16 (4): 379–389.

Microsoft. 2024a. *.NET Core: Introduction.* https://learn.microsoft.com/en-us/dotnet/core/introduction. Tilgået: 24-05-2024.

Microsoft. 2024b. *TypeScript: Typed JavaScript at Any Scale.* https://www.typescriptlang.org. Tilgået: 21-05-2024.

———. 2024c. *Why Create TypeScript?* https://www.typescriptlang.org/why-create-typescript/. Tilgået: 24-05-2024.

Newman, Sam. 2015. *Building Microservices.* O'Reilly Media, Inc.

Nielsen, Jakob. 1994. „10 Usability Heuristics for User Interface Design". Tilgået: 16-05-2024, https://www.nngroup.com/articles/ten-usability-heuristics/.

———. 2000. *Why You Only Need to Test with 5 Users.* Tilgået: 29-05-2024. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

Norman, Don. 2002. *The Design of Everyday Things.* Revised and expanded edition. New York: Basic Books. ISBN: 978-0-465-05065-9.

Papert, Seymour. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books.

Qentelli. 2024. *Introduction to Pair Programming.* https://qentelli.com/thought-leadership/insights/introduction-pair-programming. Tilgået: 01-05-2024.

React Contributors. 2024. *React: A JavaScript library for building user interfaces.* https://legacy.reactjs.org/. Tilgået: 15-05-2024.

Regeringen. 2018. *Strategi for Danmarks digitale vækst.* https://www.regeringen.dk/nyheder/2018/strategi-for-danmarks-digitale-vaekst/. Tilgået: 23-03-2024.

———. Februar 2024. *Aftale om en ambitiøs og ansvarlig strategi for Danmarks digitale udvikling.* Politisk aftale.

Resnick, Mitchel et al. 2009. „Scratch: Programming for All". *Communications of the ACM* 52 (11): 60–67.

Ries, Eric. 2011. *The Lean Startup: how today's entrepreneurs use continuous innovation to create radically successful businesses.* New York: Crown Business. ISBN: 9780307887894.

Robe, Patrick, Sandeep K. Kuttal, Ying Zhang og ... 2020. „Can machine learning facilitate remote pair programming? Challenges, insights & implications". I *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 1–9. IEEE. https://doi.org/10.1109/VLHCC50065.2020.9127250. https://ieeexplore.ieee.org/abstract/document/9127250/.

Sapounidis, Theodosios og Stavros Demetriadis. December 2013. „Tangible versus graphical user interfaces for robot programming: exploring cross-age children's preferences". *Personal and Ubiquitous Computing* 17, nummer 8 (): 1775–1786. https://www.proquest.com/scholarly-journals/tangible-versus-graphical-user-interfaces-robot/docview/1458852463/se-2.

Sarasvathy, Saras D. 2001. „Causation and Effectuation: Toward a theoretical shift from economic inevitability to entrepreneurial contingency". *Academy of Management Review* 26 (2): 243–263. https://doi.org/10.2307/259121.

———. 2003. „Entrepreneurship as a Science of the Artificial". Tilgået: 13-03-2024, *Journal of Economic Psychology* 24 (2): 203–220. https://doi.org/10.1016/S0167-4870(02)00203-9. https://www.researchgate.net/publication/222400378_Entrepreneurship_as_a_Science_of_the_Artificial.

Scharf, Florian, Thomas Winkler og Michael Herczeg. 2008. „Tangicons: Algorithmic Reasoning in a Collaborative Game for Children in Kindergarten and First Class". I *Proceedings of the 7th International Conference on Interaction Design and Children*, 242–249. ACM.

SCRUMstudy. 2014. *A guide to the Scrum body of knowledge.* 2014 edition. SCRUMstudy. ISBN: 978-0-9899252-0-4.

Super Mario Game. 2024. *Super Mario Game.* https://supermario-game.com/. Tilgået: 25-05-2024.

Sweigart, Al. 2022. *The Recursive Book of Recursion.* San Francisco: No Starch Press. ISBN: 978-1-71850-203-1.

Sweller, John, Paul Ayres og Slava Kalyuga. 2011. *Cognitive Load Theory.* Springer.

This is service design doing. 2022. *Octopus clustering.* https://www.thisisservicedesigndoing.com/methods/octopus-clustering. Tilgået: 11-04-2024.

Vercel. 2024. *Next.js: The React Framework.* https://nextjs.org/. Tilgået: 23-05-2024.

Wang, Danli, Tingting Wang og Zhen Liu. Februar 2014. „A Tangible Programming Tool for Children to Cultivate Computational Thinking". *TheScientificWorldJournal* 2014 (): 428080. https://doi.org/10.1155/2014/428080.

Wilensky, Uri og William Rand. 2015. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo.* The MIT Press. ISBN: 9780262731898. http://www.jstor.org/stable/j.ctt17kk851.

Williams, Laurie og Robert Kessler. 2000. „The costs and benefits of pair programming". *Advances in computers* 48:277–317.

Williams, Laurie og Robert Kessler. 2002. *Pair Programming Illuminated.* Addison-Wesley Professional.

Wing, Jeannette M. 2006. „Computational Thinking“. *Communications of the ACM* 49 (3): 33–35. https://doi.org/10.1145/1118178.1118215. https://dl.acm.org/doi/10.1145/1118178.1118215.

———. 2010. „Computational Thinking: What and Why?“ *Computer Science,* https://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf.

Wood, David, Jerome S. Bruner og Gail Ross. 1976. „The role of tutoring in problem solving“. *Journal of Child Psychology and Psychiatry* 17 (2): 89–100.

Aaen, Ivan. 2024. *Essence: Problem-Solution Model Generation.* Ikke udgivet.

Aalborg University. 2024. *Samarbejde med Studerende på Projektarbejde.* https://www.aau.dk/samarbejde/virksomheder/samarbejde-studerende/projektsamarbejde. Tilgået: 17-05-2024.