Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Dragsjø, Marius André

Historien om, utviklingen av og fremtiden til programmeringsspråk

Bacheloroppgave i Informasjonsbehandling
Veileder: Hjelle, Torstein Elias Løland
Mai 2024

NTNU
Kunnskap for en bedre verden

Dragsjø, Marius André

Historien om, utviklingen av og fremtiden til programmeringsspråk

Bacheloroppgave i Informasjonsbehandling
Veileder: Hjelle, Torstein Elias Løland
Mai 2024

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU

Kunnskap for en bedre verden

Innholdsfortegnelse

Forord

De fleste programmerere har møtt på en eller flere av følgende «skrekkhistorier»: Man må lete i en gigantisk trestruktur, for å finne ut hvor koden er, og hva den gjør. Eller man må bruke all sin tid på å jobbe med selve strukturen i stedet for selve problemet. Eller man lager tester som av og til feiler uten å forandre på noe som helst. Eller det tar ekstremt lang tid til å få koden til en kjørbar tilstand, noe som stjeler tid til testing og feilretting. Eller man bruker enormt lang tid til å utvikle effektiv kode uten særlig forbedring i ytelse.

Ved slik bortkastet tid, er det vanskelig for bedrifter å innovere og utvikle gode systemer, og de kan bli mer avhengig av utdaterte programmeringsspråk som hindrer progresjon. Mens for programmererne er det frustrerende å arbeide innenfor et programmeringsspråk som ikke sammenfaller med deres ønsker for hva et programmeringsspråk kan gjøre, og unødvendig tid brukes til å arbeide mot programmeringsspråket i stedet med selve problemet. Hvis du skulle havne i en slik situasjon, ville du ha akseptert dette som «status quo»? Eller ville du ha prøvd å forandre arbeidssituasjonen til noe bedre?

For de som ønsker forandring, vet at det ikke alltid er enkelt å finne fram en bedre løsning når man ikke har mye kunnskap om andre programmeringsspråk med dets fordeler og ulemper. Derfor er formålet med dette dokumentet å gi programmerere et bedre grunnlag over hvilke programmeringsspråk som finnes, og gi dem redskap til å finne ut hvilke som fungerer i ulike situasjoner, og om noen er generelt bedre enn andre.

Metode

For å underbygge dette formålet, har jeg delt opp dokumentet i tre hovedkapitler. Det første kapittelet tenker å gi leseren en bakgrunnsforståelse for hvordan programmeringsspråk har utviklet seg gjennom historien, og trekke frem punkter fra historien som vil bli relevant i de neste hovedkapitlene. Det andre kapittelet skal veilede leseren til et begrenset antall programmeringsspråk som kan være verdt å vurdere for en programmerer. Det gjøres ved å diskutere hva som gjør et programmeringsspråk godt, i forhold til hva programmeringsspråket skal bli brukt til, og hvilken oppgave programmereren har. Videre blir det diskutert ulike punkter en programmerer kan ha fokus på, om det skulle være: vedlikehold, lesbarhet, hvor rask man kan kode, eller hvor rask koden er, noe som kan endre valget av et programmeringsspråk. Og til slutt, diskutere ulike domene-spesifikke programmeringsspråk som kan være spesielt rettet mot typiske oppgaver. Det tredje kapitlet diskuterer mulige situasjoner i fremtiden, noe som kan ha konsekvenser for valg av programmeringsspråk. Dette inkluderer populariteten av ulike paradigmer, en økt bruk av kunstig intelligens, og andre spekulasjoner.

Historien

Når det kommer til det første kapittelet, er det veldig mange punkter i historien som kan være verdt å nevne. For eksempel er det sagt «Det brukes flere enn 170 programmeringsspråk i USA i 1972» [2], som betyr at det er urimelig å nevne alle programmeringsspråkene som har blitt laget. Jeg må dermed begrense materialet jeg tar med [3]. Jeg filtrerte dermed på programmeringsspråk basert på i hvor stor grad det har bidratt til teknologiske fremskritt, både for vår tid og potensielt i fremtiden. Dermed kan jeg enklere danne en rød tråd gjennom historien mot nåtiden og fremtiden, noe som gjør det enklere å argumentere for hva som gjør et programmeringsspråk godt, og enklere undersøke mulige situasjoner i fremtiden.

Når det kommer til å hente informasjon gjennom historisk litteratur, bestemte jeg for å fokusere på artikler av individuelle programmeringsspråk skrevet av forfatterne av programmeringsspråket. Dette har jeg funnet gjennom en troverdig søkemotor som google scholar. Lignende har jeg også søkt etter foredrag som handler om

programmeringsspråk, fordi de er mer troverdige enn tilfeldige folk på nett. De snakker om mange forskjellige programmeringsspråk som gjør det enklere å se hvordan programmeringsspråkene henger sammen, og de gir kontekst til hvordan ulike programmeringsspråk har påvirket oss i nåtiden. Til annen informasjon, brukte jeg artikler og videoer fra troverdige individer for å forklare grunnleggende konsepter om hva en datamaskin er, hva et program er, og hva et programmeringsspråk er i historisk kontekst. Men når det kommer til enkeltinformasjon som «Cobol er nå kjent for å opprettholde gamle systemer» eller hvordan folk flest vil definere et programmeringsspråk, så har jeg ikke hatt like mye fokus på hvor kilden kom fra.

Nåtiden

I det andre kapittelet, der vi skal diskutere hva som gjør et programmeringsspråk godt, så finnes det ikke et objektivt riktig svar i alle situasjoner. Dermed må vi få tak i subjektiv informasjon som gir gode argumenter. Dette blir trukket frem sammen med ulike punkter en programmerer kan ha fokus på, der vi diskuterer fordeler og ulemper ved bruk at programmeringsstiler i ulike situasjoner. Lignende, blir dette satt sammen med ulike domenespesifikke programmeringsspråk som kan være spesielt rettet til typiske oppgaver. Grunnen til at jeg ikke direkte forteller hvilke programmeringsspråk man burde bruke, er at de utvikler og forandrer seg selv hele tiden, noe som gjør at jeg har rettet fokuset mot programmeringsstil og hvordan et programmeringsspråk kan støtte arbeidet ditt.

Når man arbeider med subjektiv informasjon, så må man være forsiktig og holde seg selv nøytral til informasjonen man får for å ikke lene mot den ene eller andre siden. Samtidig må man være forsiktig med hvor man henter dataen fra, og sjekke at informasjonen man henter er troverdig nok til å bli brukt. Siden slik informasjon kan være i direkte konflikt med hverandre, så finnes det ikke et objektivt riktig svar.

For å hente slik subjektivt data, bestemte jeg for å bare hente informasjon fra troverdige kilder. For å være troverdig nok, må personen ha jobbet i programmerings-industrien i en relativt lang tid, slik at de har hørt mange argumentasjoner for og mot

ulike programmerings-relaterte emner. Eller så må de allerede ha nok troverdighet til å kunne holde et foredrag på en konferanse. Slik informasjon har jeg ofte hentet i form av videoer på Youtube, fordi det er enklere å dele foredrag og subjektiv informasjon via videoer enn artikler, det er enklere å hente informasjon fra videoer, og det er enklere å verifisere om individet har hørt mange argumentasjoner for og mot ulike programmerings-relaterte emner.

Når det trengs å hente informasjon fra allmenhetens syn på et tema, så har jeg redusert kravet til at de må ha arbeidet innenfor det de snakker om, og samtidig vet om flere synspunkter rundt det de snakker om for å gi en solid argumentasjon. Slik informasjonen har jeg ofte hentet fra diskusjonsfora rettet mot å diskutere og svare på spørsmål som Quora, i stedet for populære sosiale medier som Reddit der brukerne vil argumentere sitt synspunkt i stedet for å forklare emnet.

For å unngå å fokusere for mye på en side av et subjektivt tema, bestemte jeg for å intervjue akademikere fra NTNU og fulltids-arbeidende programmerere. Dette er for å bekrefte at jeg holder meg nøytral overfor informasjonen jeg samler inn, samtidig som at jeg kan sjekke at informasjonen jeg har samlet inn gir mening. Jeg kan også mye enklere samle informasjon fra flere synsvinkler om et tema gjennom intervjuene, på grunn av at jeg kan stille spørsmål til dem direkte.

Fremtiden

Når det kommer til det tredje kapittelet, er det enda mer fokus på subjektiv data, fordi ingen vet hva fremtiden bringer, og spekulasjoner kan variere veldig. Men hva som kan skje i fremtiden, kan fortsatt ha konsekvenser for valget av programmeringsspråk. Dermed bestemte jeg for å ikke se for langt fram i tid, men heller starte i 2024 og se fremover for å finne mulige situasjoner i fremtiden. I et kort tidsperspektiv kan man ved hjelp av trender kunne si med god sannsynlighet hvilke situasjoner som kan skje og hvordan disse situasjonene kan påvirke valget av hvilket programmeringsspråk man velger å bruke. Dette inkluderer populariteten av ulike paradigmer, en økt bruk av kunstig intelligens, og andre spekulasjoner.

Det å samle sannsynlig informasjon om fremtiden er en veldig lignende prosess som å samle informasjon om nåtiden, der man henter subjektivt informasjon gjennom videoer, forelesninger, diskusjonsfora og intervjuer. Men her har jeg fokusert mer på intervjuer, på grunn av at jeg kan diskutere ulike forslag direkte med de jeg intervjuer for å finne ut om situasjonen er realistisk. Men for å finne trender innenfor funksjonell programmering og kunstig intelligens, har jeg fortsatt tatt i bruk de mer tradisjonelle kildeformene.

Historie

The Analytical Engine

Når man snakker om historien innenfor programmeringsspråk, så er det vanlig å begynne med Ada Lovelace og «The analytical engine» i 1837, som kan gi oss innsikt i hva et program er, hvordan det fungerer, og hva man kan definere som et programmeringsspråk.

«The analytical engine», var det første designet av en datamaskin for generell bruk [5][6][7][8]. Der alle andre maskiner på den tiden var spesialisert til en oppgave, så kunne man ved bruk av denne maskinen (hvis den ble bygget), i teorien lage programmer som kan regne ut hva som helst. Dette kunne maskinen klare ved å lagre, overføre, og manipulere tall basert på instruksjoner som ble gitt til maskinen gjennom mange sammenkoblete hullkort, der maskinen kunne hoppe frem og tilbake i instruksjonen og endre hva den skulle gjøre basert på betingelser [5]. I stedet for kodesetninger som vi bruker i dag, så måtte de fysisk lage hull i kort for å lage variabler og instruksjoner slik at maskinen kunne lese dem. Dessverre, var det ingen som skjønte hvordan dette konseptet fungerte, siden Babbage, som lagde «The analytical engine», snakket ikke med noen om hvordan maskinen fungerte [9]. Dette ble endret når Ada Lovelace, en god venn av Babbage, klarte å forstå hvordan maskinen fungerte i 1843, og lagde flere teoretiske programmer i form av et sett med instruksjoner [5][9]. Dermed, ble Ada Lovelace den første programmereren [10].

Her kan man forstå at et program er hovedsakelig et sett med instruksjoner gitt til en maskin som kan hoppe fram og tilbake i instruksjonene, og samtidig manipulere data. Men om folk i dag vil kalle dette et programmeringsspråk er lite sannsynlig, siden det ikke fantes en universell definisjon av et programmeringsspråk. Men det er sagt at det generelt er et sett med karakterer og regler for å kommunisere med en datamaskin, der det ikke trengs noe form for tolkning [2][11]. I dag er det mer vanlig å si at programmeringsspråk typisk er høyere nivå enn direkte maskinkode.

Det var dessverre ikke mye som skjedde innenfor programmeringsspråk etter hva Babbage og Ada lovelace klarte å gjøre. Men etter en god stund, kom matematikeren Alan Turing (ofte regnet som programmererens far) og gjorde store fremskritt som verden fortsatt føler effekten av [1].

1940s Alan Turing

Alan Turing, klarte å bevise at det er mulig å lage en universell maskin som kan utføre hva som helst en spesiell designet maskin kan gjøre ved hjelp av databehandling. Dette er sant bare hvis maskinen har så mye minne som problemet trenger, og at maskinen klarer å endre hva den skal gjøre basert på betingelser; som for eksempel lese, skrive, hoppe mellom instruksjoner, overskrive, fjerne og endre konfigurasjoner maskinen kan ha [12][13]. En maskin som kan gjøre dette, kan da bli definert som en «Turing maskin», mens et programmeringsspråk som teoretisk kan gjøre alt en Turing maskin kan gjøre, kan bli kalt «Turing complete». Med denne beskrivelsen, hvis man går tilbake til Baggabe sitt design, kan man se at den kvalifiserer for å være en Turing maskin, siden man teoretisk kan legge til så mange hullkort som maskinen trenger, og maskinen klarer å endre hva den skal gjøre basert på ulike betingelser.

Omtrent på denne tiden, begynte elektriske datamaskiner å bli tatt i bruk, som for eksempel ENIAC som kunne kalkulere forskjellige matematiske formler til ulike formål. Men for å operere disse, måtte man programmere ved hjelp av en rad av

mekaniske brytere, eller med ledninger man plugget inn og ut, eller med en haug av hullkort for at datamaskinen kunne forstå hva de prøvde å kalkulere [1][14]. Man kan enkelt forstå at dette ble raskt veldig vanskelig og komplekst å operere hvis man ønsket å lage store og effektive kalkulasjoner, noe som selvfølgelig førte til at folk ville forenkle denne prosessen [15].

Assembly

Dette førte til flere initiativ og forslag til høy-nivå programmeringsspråk. I stedet for å jobbe direkte med maskinkode, så kunne man jobbe med et språk med engelsk syntaks som kunne bli gjort om til maskinkode [2][14][16]. Dermed, mot slutten av 1940 tallet, ble det laget et språk som het Assembly, der man kunne bruke engelske ord som forkortelser for å representere en handling datamaskinen skulle gjøre, noe som var designet for å utnytte individuelle datamaskiner sin arkitektur [14][17]. Dette kunne for eksempel være å flytte data ved «movl», eller å hoppe over instruksjoner med «goto». Men dessverre ble det ikke laget en standardisert versjon av Assembly. Derfor er Assembly avhengig av hvordan maskin-kode instruksjonene fungerte til en spesifikk datamaskin, der det også kunne finnes flere ulike Assembly språk til samme datamaskin sin arkitektur [17][18]. Det å ha engelske forkortelser som forskjellige maskinkode instruksjoner simplifiserte prosessen og økte hastigheten på hvor raskt man kunne programmere et program [14]. Men det å programmere i Assembly ligner fortsatt veldig på det å programmere i maskinkode. Så selv om Assembly var blant de første lav-nivå programmeringsspråk og et stort steg i riktig retning, så var det fortsatt initiativ til å gjøre forbedringer.

1950s

Utvikling av mer høy-nivå programmeringsspråk begynte nå å bli tatt mer seriøst, og noen forsøk på «automatisk programmering» ble gjort, men mislyktes på grunn av ulike grunner. De fleste forsøkene var enkelt og greit Assembly med ekstra funksjonaliteter som var ubrukelige, eller det var et enkelt programmeringsspråk for å fikse problemer noen datamaskiner hadde [2][15]. Og av de som kunne regnes som høy-nivå programmeringsspråk, var de opp til 10 ganger tregere. Dette, sammen med programmerere sin erfaring med problemer ved å bruke disse høy-nivå

programmeringsspråkene, og at bedre datamaskiner begynte å fikse problemer de hadde i stedet for å bruke ulike programmeringsspråk som eksisterte for å fikse disse problemene, har overbevist mange at effektiv programmering ikke kunne bli automatisert [15]. Dermed for at et mer høy-nivå programmeringsspråk skulle bli aktuelt å bruke, og ikke bli ignorert på grunn av den sterke skepsisen, så trengtes det et stort fokus på effektiviteten av programmene som kunne bli laget med programmeringsspråket, og som ikke var for komplisert å bruke.

Fortran

John Backus var en av dem som vil ville ha et bedre redskap til å kalkulere formler, enn å bruke Assembly eller maskinkode. Så i 1954 samlet han et team fra IBM, og i 1957 hadde de bygget et programmeringsspråk kalt FORTRAN (FORmula TRANslator) [1][15]. Fortran er regnet som det første store høy-nivå programmeringsspråket, som ble faktisk brukt av mange, siden det ikke var for komplisert å bruke, og det var like raskt eller raskere enn maskinkode [1][2][15]. Dette var et enormt framskritt. Det å vise folk selve konseptet med et høy-nivå programmeringsspråk, der man ikke trenger kunnskap om hvordan maskinen fungerer på innsiden, eller hvordan man skal lage maskinkode, og at det kunne generere effektiv kode, var revolusjonerende [1]. Det at man kan faktisk lage kode med engelsk syntaks, som kunne bli brukt til mer enn en enkelt forkortelse for maskinkode, la et solid grunnlag for fremtidig utvikling. Det åpnet også muligheten for vanlige folk, som ikke var spesialisert innenfor datamaskiner, å programmere ved å bruke vanlig språk [1][2]. Mye av fokuset ble altså rettet mot selve problemløsningen, og ikke på hvordan maskinen fungerte.

Algol

Utviklingen av Fortran var fantastisk, men det var fortsatt noen som ikke var helt fornøyd med hvordan det fungerte. Siden mye av fokuset var på optimalisering i stedet for strukturering av koden [15], så var det flere som mente at fokus på struktur var noe som burde være med i programmeringsspråket. En av disse var Edsger Dijkstra. Han var ikke fornøyd med hvordan man lagde kode i Fortran, og han argumenterte for at programmering burde være strukturert [1]. Dermed, rundt 1958

begynte han sammen med et team å utvikle et programmeringsspråk kalt ALGOL (ALGOrithmic Language) for å danne strukturen han ønsket seg [1][2]. For å gjøre dette, bygget de videre på ideen om å lage kode som er mer enn enkle forkortelser for maskinkode [19].

Algol ble et programmeringsspråk der man kunne definere type variabler og sette verdi på disse navngitte variablene, i stedet for uklare deklarasjoner. Dette fjernet mye kode som var vanskelig å forstå, og hjalp til med å lage kode som har lignende syntaks som et faktisk språk, som for eksempel engelsk [2][19]. Lignende utviklet de ulike muligheter til å kontrollere flyten over hvordan koden blir kjørt, altså rekkefølgen av ulike instruksjoner/prosedyrer utover koden, som for eksempel: if-then-else, for-løkker og funksjoner [1][19]. Til slutt, gjorde teamet det mulig å samle sammen ulike instruksjoner/prosedyrer inn i en blokk, slik at man bare trenger å peke til navnet av blokken for å gjøre alt innenfor den [1][19]. Dette gjorde det langt enklere å strukturere og organisere koden som man ville, individuelle blokker var mye enklere å forstå, og man kunne enkelt kontrollere hvordan koden skulle leses gjennom forskjellige blokker.

Hvis man har brukt et programmeringsspråk i dag, kan man se at denne ideen om struktur ble adoptert i så si alle programmeringsspråk. Lignende, finnes det mange ulike forslag på hvordan man kan videreutvikle strukturen av koden, altså paradigmer, eller som jeg kaller dem: ulike stiler innenfor programmering. Disse har hatt, og har fortsatt stor effekt på nåtidens programmeringsspråk, noe som vil være relevant i diskusjonen rundt hva som gjør et programmeringsspråk godt.

Funksjonell Programmering

John McCarthy var en av dem som introduserte et paradigme kalt funksjonell programmering, gjennom programmeringsspråket «LISP» (LISt Processing) [2]. Rundt 1958, var John McCarthy interessert i å danne et programmeringsspråk mer rettet mot kunstig intelligens, og argumenterte for å få til det, så må man bruke noe han mener mennesker gjør for å tenke, altså lister [1][20]. Dermed utviklet de et liste-

basert programmeringsspråk hvor man blant annet kunne behandle funksjoner som om de var vanlige variabler, som en «integer» eller «string» [1][20]. Selv om John McCarthy mente det eller ikke, så introduserte dette funksjonell programmering ved at funksjoner kunne sende seg selv inn i andre funksjoner [1][22]. Det finnes ingen formell definisjon på hva funksjonell programmering er, men det er sagt at funksjonell programmering er ment for å unngå «mutasjoner» og «bivirkninger» så mye som mulig, ved at man ikke kan endre på verdier utenfor seg selv, og ved å bruke en kopi av input-verdiene for ikke å endre disse verdiene globalt for andre funksjoner [21].

Grunnen til at funksjoner som variabler introduserte funksjonell programmering, er at funksjoner kan nå enkelt sende sin output til andre funksjoner uten å måtte lagre verdien inne i en variabel. Dette hindrer at man må lagre verdiene funksjonene sender ut når det vil kommunisere med andre funksjoner, noe som gjør at man ikke trenger å endre verdien når man kjører funksjonen igjen med annet start-input. Altså, en direkte måte for funksjoner å sende sin verdi til andre funksjoner uten å forandre på verdier utenfor funksjonen [22]. Dette er sagt å redusere risikoen for feil ved å gjøre det enklere å skrive og rette feil i kode [21][22][23]. Mer om dette blir diskutert i kapittelet «hva gjør et programmeringsspråk godt?».

1960s Objekt-orientert

Ole-Johan Dahl og Kristen Nygaard var også noen som introduserte et paradigme, kalt objekt-orientert programmering (OOP), gjennom programmeringsspråket «SIMULA» (SIMULAtion) [1]. De var interessert i å simulere flere biter som samhandler inn i et system, der man kan «pakke inn» de forskjellige bitene som en selvstendig byggekloss, og se hvordan den samhandler med andre biter i systemet [1]. Dette var en utvidelse av Algol, som hadde fokus på strukturen av koden, mens Simula introduserte funksjonalitet til å simulere flere biter. Dette var klasser som definerer objekter, objekter som holder data og funksjoner, og «inheritance» altså «arv» slik at klasser kunne arve fra andre klasser [1][21]. Man kunne bruke objektene som forskjellige biter av systemet, som ifølge kildene gjør koden mer organisert [21][32] og enklere å strukturere koden [24][25]. Dette introduserte objekt-orientert

programmering der man bruker objekter og klasser sammen med arv som kan hjelpe programmereren til å bygge opp struktur.

Men det er uenigheter om dette faktisk er objekt-orientert programmering. Alan Kay, som lagde terminologien «objekt-orientert programmering» gjennom sitt programmeringsspråk SmallTalk, likte ikke hvordan Simula implementerte «arv» [26]. Han mente at objekt-orientering handler om beskjeder, der man må beskytte lokale verdier fra andre, og at mottakeren skal bestemme hva som skal skje, ikke senderen. Han snakker altså ikke om arv, «polymorfisme» eller «encapsulation» som er mer kjent som objekt-orientering for mange i dag. Han ville ha objekter som hver var en egen datamaskin som bare kunne bli snakket med gjennom beskjeder [26][27], slik at de faktisk blir en «bit av systemet», i stedet for å gjøre det Simula gjorde ved å ha abstrakte datatyper gjennom objekter og arv [26][27][28][29]. Jeg vil referere dette som «Message-orientert programmering» som vi kommer tilbake til i kapittelet «hva gjør et programmeringsspråk godt?» der skillet mellom Simula og SmallTalk vil være relevant.

1970s C

Frem til 1970 hadde alle programmeringsspråk og de fleste operativsystemer blitt laget gjennom maskinkode eller Assembly, siden det ikke fantes et dedikert programmeringsspråk for å lage programmeringsspråk og operativsystemer [14][30][33]. Dette endret Dennis Ritchie da han lagde sitt programmeringsspråk C, som mange sier er «Mother of all languages» [34]. Dennis, sammen med Ken Thomson, ville lage et systemutviklings-programmeringsspråk for å utvikle et operativsystem som kunne hjelpe med å koble sammen mange ulike koder/programmer på samme datamaskin [14][30][33]. Språket C er hovedsakelig en videreutvikling av programmeringsspråket B, som igjen er en videreutvikling av et system-programmeringsspråk kalt BCPL, sammen med en syntaks som lignet på Algol [1][33]. De bestemte at de ikke bør legge inn noen «restriksjoner», «rare funksjoner» eller «funksjoner som tar lang tid å kompilere». Men i stedet bevisst legge inn få funksjoner de vet vil fungere slik at de kan fokusere på å gi programmerere enklere og nær tilgang til maskinvaren som gjør koden effektiv,

kortfattet og uttrykksfull. Dette gjør at programmerere ikke trenger å bruke Assembly eller andre store programmeringsspråk som ikke er så effektive eller uten restriksjoner [30][31]. Sammen med hvor enkelt det er å bruke C [30][31], gjorde C det mest suksessfulle programmeringsspråket som ligner på hvordan en datamaskin fungerer [31], der C ble systemet som sto bak mange programmeringsspråk og systemer generelt [33].

1980s C++

Dette førte til utviklingen av et veldig kjent programmeringsspråk kalt C++. Han som lagde C++, Bjarne Stroustrup, ville ha effektiviteten til C, men også støtte til å enklere strukturere koden [32]. Bjarne likte hva Simula gjorde, der man lager objekter for å definere sine egne typer og regler, mens programmet sjekker at disse typene og reglene blir fulgt. Bjarne mente at man kan lage sine egne definerte regler, som kan hjelpe deg med å tenke og designe kode [32]. Denne ideen, sammen med strengere typer og regler enn det Simula ga [21], pluss andre funksjonaliteter, endte med opprettelsen av C++, der noen mener at dette var starten på den store popularitetet innenfor objekt-orientert programmering [27].

På dette punktet i historien om utviklingen av programmeringsspråk har de viktigste komponentene allerede blitt oppfunnet, og antallet programmeringsspråk vokste raskt nå som det ble enklere å utvikle dem. Hvor stor påvirkningskraft et individuelt programmeringsspråk har vil dermed minke i konkurransen med andre programmeringsspråk. Med det kommer vi frem til i dag, hvor objekt-orientert programmering har for det meste dominert markedet i flere år.

Populariteten med OOP

Ifølge Bjarne Stroustrup, da objekt-orientering sammen med strengere typer og regler først ble lagt inn i C, var det bare et «medium suksess». Men da annen funksjonalitet ble lagt til, ga det langt mer suksess enn hva objekt-orientering med strengere typer og regler gjorde alene [21][32]. Senere, kom programmeringsspråket Java inn i bildet, hvor de ville ligne på C++ for å danne en enkel overgang for C++ brukere til å

bli Java brukere. Dermed kopierte de mye fra C++ som brukerne allerede var kjent med, noe som inkluderte objekt-orientering [21][24], og de forenklet programmeringsspråket som gjorde overgangen mer appellerende [24][35]. Det samme kan også bli sagt om C# som også ville være veldig lik Java, slik at brukere kunne ha et alternativ som er «lignende Java» [21]. Lignende situasjoner skjedde også med andre programmeringsspråk. For eksempel objectiv-c, der han som lagde programmeringsspråket ville ha «modularity/encapsulation», altså Smalltalk sin definisjon av «encapsulation», hvor informasjon inne i et objekt skulle bli gjemt fra andre objekter, og objekter ikke trengte å vite hvordan ting fungerte på innsiden av et annet objekt [21][36]. Men han som lagde objectiv-c visste bare om å gjemme informasjon gjennom objekter, og dermed utviklet et programmeringsspråk som hadde C med objekt-orientering. Så kom programmeringsspråket Swift som måtte være bakover-kompatibel med objectiv-c, og gjorde at objekt-orientering ble tatt enda mer i bruk [21][37].

Hvis man følger denne listen med programmeringsspråk som henter objekt-orientering fra de forrige programmeringsspråkene, så kan man se at objekt-orientering ved C++ får æren for suksessen. Men her er det vist at det var bare en «medium suksess», der det var de andre funksjonalitetene som fikk C++ til å bli ganske populær [21]. Lignende med Swift fra objektiv-c, der han som lagde programmeringsspråket ville bare gjemme informasjon som han bare visste fantes via objekter. Så man kan si at objekt-orientert programmering ble populær ved en tilfeldighet, der encapsulation/modularitet er en god ide, men fikk det tilfeldigvis fra OO, samtidig der OO fikk mye av æren [21][27]. Dette kan være en mulig forklaring på hvorfor OOP dominerer markedet, der mange ikke er fornøyd med hvordan objekt-orientering fungerer, noe som vil være relevant i kapitelet «hva gjør et programmeringsspråk godt?» [29][39].

Hva gjør et programmeringsspråk godt?

Det er mange som sier at det ikke finnes et objektivt svar på dette. Her gis allikevel et forslag til en oversikt over hva som gjør et programmeringsspråk godt, ved å finne ulike eksperter sin mening og innsikt rundt temaet.

Ifølge kildene så finnes det ikke noe «silver bullet» [40], altså det finnes ikke et enkelt svar på det vanskelige spørsmålet om hva gjør et programmeringsspråk godt. Det finnes heller ikke noe som er «objektivt god» i alle situasjoner, med unntak av god basis funksjonalitet, god dokumentasjon og et økosystem for gjenbruk av andre sin kode [41][43][44]. Derimot er det enighet om at det kommer veldig an på oppgaven programmereren skal løse [27][40][41][42][43], og om det skal være fokus på vedlikehold, lesbarhet, hvor rask man kan kode, pålitelighet, hvor rask koden er, og så videre.

Hvis man ser gjennom historien om programmeringsspråk, så ser man at struktur og oversiktlighet var tidlig sett på noe som var viktig å ha fokus på, med tanke på at størrelsen av koden blir større og mer forvirrende etter gjentatt videreutvikling av koden. Dette fokuset av struktur, kan man se i Algol, Objekt-orientering, funksjonell programmering og C++. Flere av kildene påpeker at god struktur også er viktig hvis man vil at koden skal holde ut over lang tid, spesielt når man jobber i en bedrift der flere personer jobber med koden som forhåpentligvis skal være i bruk i mange år [40][41][42][43]. Så angående struktur og lesbarhet, hva skal man gå for?

Struktur og lesbarhet

Hvor stort fokus man skal ha på struktur og lesbarhet, spørs på størrelsen av oppgaven programmereren har, hvor for eksempel ved en liten oppgave, så trengs det ikke mye fokus på strukturen eller vedlikehold, der man kan relativt enkelt lage en ny løsning [42]. I en slik situasjon, kan det være lurt å gå for paradigmet, eller stilen man er mest komfortabel med, der man velger det som passer til hvordan man tenker [42][44]. For eksempel om man er en matematiker, kan man bruke funksjonell programmering. Om man tenker på koding som en «modell» eller «simulering av

noe», kan man gå for objekt-orientert. Om man vil lage kode raskt uten mye planlegging, kan man velge prosedyre-programmering der man ikke har pålagt struktur fra starten av [42].

Men når oppgaven begynner å bli større, kan det være lurt å gjøre koden mer strukturert og lesbar fra starten av. Her argumenteres det ofte på nett om to ulike paradigmer, enten objekt-orientert programmering eller funksjonell programmering [21][23][24][27][29][39]. Gjennom ulike kilder, intervjuer og foredrag, har jeg kommet til at objekt-orientert programmering passer best innenfor oppgaver med medium størrelse. Det første argumentet, er at det ikke skalerer så bra, der objekt-orientert løsninger basert på Simula har hatt skalerings problemer [28]. Siden objekt-orientering har strenge regler, og begrenser programmereren ved at man måtte tenke abstrakt gjennom objekter, noe som fører til at strukturen man bygger opp ikke blir godt egnet til forandringer [45]. Fordi når en forandring skjer, eller krav til oppgaven forandres, er det ofte at man lager hull i strukturen slik at andre biter i strukturen kan snakke med hverandre for å oppfylle de nye kravene [24][39][40][43], noe som ofte ender opp med spagetti-kode i stedet for en oversiktlig struktur. Så det argumenteres at det å lage god abstraksjon som bruker definerte typer og regler, er veldig vanskelig og tar veldig lang tid å få til riktig, og dermed bør det gjøres bare hvis absolutt nødvendig, i stedet for å gjøre det så ofte som mulig [25][24][46].

Det andre argumentet, er at objekt-orientering ble laget for å modellere og simulere systemer [32][42][43]. Dette kan man kan se ved at Simula har et navn som ligner på simulering, og at C++, som gjorde OOP populært, er også basert på Simula. I intervjuene kom det også frem at objekt-orientering er godt egnet og laget for å modellere ting [42][43], og under et intervju kom spill-design som et eksempel. Skaperen av C++ mener også at, når programmeringsspråket sjekker at typene og reglene som man selv har laget blir fulgt, så kan det hjelpe deg med å tenke og designe din kode, noe som jeg er enig i bare hvis uforventete endringer eller nye krav til oppgaven ikke skjer [32].

Men når oppgaven blir stor, og løsningen krever mange tusenvis med linjer av kode, er det nesten umulig å unngå uforventet endringer eller nye krav, noe som gjør det ganske vanskelig å bygge videre på en gigantisk struktur. Dermed mener jeg at det er best å bruke Message-orientert, eller funksjonell-orientert programmering, eller begge ved store oppgaver. Message-orientert og funksjonell programmering, er lignende i at begge fokuserer mer på oppførselen og beskjedene i stedet for selve dataen og objektet [45]. Begge kan mye enklere bli skalert og endret på når man ikke har fokus på en gigantisk struktur, og i stedet har fokus på hvordan bitene snakker sammen med hverandre [45]. Ulik fra hverandre, så gjør funksjonell programmering dette ved å ha tilnærmet 100% selvstendig kodeenhet, der kodeenhetene ikke endrer på andre sine verdier [22][24][43]. Dette gjør at man ikke trenger å tenke på noe annet når man endrer noe, legger til noe nytt, bruke funksjoner igjen, eller bytter den ut med andre funksjoner [21][22][23]. For eksempel så har ikke rekkefølgen av funksjonene noe å si hvis de ikke klarer å endre på andre sine verdier, samtidig som at det gjør at en «samtidighets-problem» ikke er mulig, fordi en slik situasjon skjer bare når funksjonene kan endre på andre sine verdier [21][23]. Og basert på hvordan programmeringsspråket håndterer verdier som absolutt trenger å ha muligheten til å bli endret av andre, så kan man drastisk redusere søkeområde når man skal finne ut hvor det skjedde noe feil [21][39]. Message-orientert derimot gjør det litt enklere å implementere kode, der den ikke følger reglene som funksjonell programmering har. Og det kan gi ulike kodeenheter gjennom objekter [45], der funksjonell programmering kan bruke de for å lime sammen bitene til å snakke med hverandre [22], noe som ofte kan gjøres gjennom mindre syntaks enn andre paradigmer kan [48].

Effektivitet for programmereren

Men hva om man ikke jobber i en bedrift, eller med mange andre personer der koden skal vedlikeholdes over en lang periode. Hva om man bare jobber for seg selv? Når oppgaven ikke er så stor, kan det være lurt å gå for paradigmet, eller stilen man er mest komfortabel med, der man velger det som passer til hvordan man tenker [42][44]. Samtidig kan man også tenke på ting som gjør programmeringen enklere og mer effektivt. For eksempel om programmeringsspråket inneholder god dokumentasjon og økosystem [41][43][44], og om man kan stole på at gjenbrukt kode

holder seg til det paradigmet man selv ønsker. For eksempel med funksjonell programmering, så må man vite om koden laget av andre endrer på verdier utenfor seg selv eller ikke [21]. Man kan også tenke på programmeringsspråk som oversetter til andre programmeringsspråk som oftest gir deg enklere syntaks å programmere med, der den fortsatt gir mye av mulighetene fra programmeringsspråket de oversetter til. Som for eksempel Elm til Javascript, [49], Kotlin til Java [40], Julia til Python [43], eller Python til C++ [50].

Domenespesifikk

Men hva om man har fokus på et spesifikt område, noe som antageligvis er den faktoren som har størst påvirkning, som for eksempel systemutvikling, kunstig intelligens, webside, databaser og så videre.

Innenfor systemutvikling er det mye fokus på hvor rask selve systemet kjører og hvor rask koden klarer å utføre ulike oppgaver, fordi ingen vil lage programvare ved hjelp av et tregt system. Det som er viktig å få frem, er at hvor rask koden kjører avhenger mye av hvordan en datamaskin fungerer, der den endrer på verdier hele tiden. Dette passer greit sammen med prosedyre-programmering, der endring av verdier kan gjøres når som helst [27]. Samtidig er ønsket om en kode som kjører raskest mulig ofte i konflikt med paradigmet om å danne en struktur som er oversiktlig for oss mennesker [53]. Men her må man være forsiktig, siden hvis kompilatoren til et programmeringsspråk er veldig prosedyre-orientert, sammen med mange optimaliseringer basert på et paradigme, så kan andre paradigmer være nesten like bra som prosedyre-programmering. Men dette varierer veldig fra programmeringsspråk til programmeringsspråk [49]. Hvis man også har en stor oppgave, så kan funksjonell programmering gi deg en bonus i effektiviteten til koden, der man kan kjøre mange kodeenheter samtidig, siden det har ikke noe å si i hvilken rekkefølge de kjører [21][49].

Hvis man for eksempel vil lage en webside, så er det hovedsakelig et programmeringsspråk som har vært dominerende innenfor front-end webutvikling:

JavaScript [21]. Dette er på grunn av at JavaScript har så si monopol på front-end webutviklings markedet [21], og det har blitt gjort om til en «de-facto standard» for webutvikling sammen med HTML og CSS for henholdsvis struktur og utseende [47], uavhengig av hva andre sin mening rundt JavaScript er [38]. Men som sagt i avsnittet effektiv koding, har man muligheten til å bruke programmeringsspråk som oversetter til JavaScript, noe som kan gi andre fordeler.

Hvis man derimot vil behandle data for eksempel i en database, så finnes det ulike programmeringsspråk for databaser som SQL og XQuery. Men her finnes også programvare laget i andre programmeringsspråk som er ment for databaser. En database er for det meste, oppbevaring, manipulering, og uthenting av data [51], noe som er enkelt nok til å bli skrevet i andre programmeringsspråk og gjort om til en programvare man kan bruke. Men selv innenfor databaser, finnes det ulike paradigmer og stil man kan bruke for ulike databehandlings formål [52].

Etter å ha gått gjennom struktur og lesbarhet, effektivitet for programmereren, og domenespesifikt programmeringsspråk, blir det mulig for leseren å se at det som har noe å si, er hvor fri man er til å bruke programmeringsspråket til å gjøre det man vil, og hvor enkelt det er å programmere basert på sin oppgave. Fordi i teorien, kan man bruke hvilken som helst programvare eller programmeringsspråk til å programmere, man kan til og med programmere ved å bruke Excel hvis man ønsker, selv om det ikke hadde vært så effektivt [60].

Men det kan hende noen fortsatt er usikker på valget sitt, og kan tenke: «Hva om noe bedre kommer i fremtiden?», eller «Hva om programmeringsspråket jeg har valgt mister populariteten?». Slike spørsmål leder oss inn i den siste biten av dette dokumentet der det skal diskuteres hva fremtiden kan bringe basert på paradigme, trender og eksperter sine meninger. Dette er for å hindre at nåtidens programmerere blir fastlåst i utdatert programmeringsspråk som hindrer progresjon og interesse for ny utvikling.

Fremtiden

Ingen vet hva fremtiden bringer, og spekulasjoner kan variere veldig. Vi skal derfor ikke se for langt fram i tid, men heller starte i 2024 og se fremover for å finne mulige situasjoner i fremtiden.

Funksjonell programmering

Noe som har blitt hintet til i dokumentet, er en økning av popularitet innenfor funksjonell programmering. Ifølge intervjuer og foredrag [21][22][23][24][40][41][43][49], så kan dette være på grunn av flere årsaker. For eksempel, så er funksjonell programmering veldig god til å prosessere store mengder av data [41], noe som bedrifter trenger nå for tiden på grunn av Big Data og kunstig intelligens. Siden funksjonell programmering er veldig skalerbart og enklere å distribuere [41][43], ved at man kan kjøre mange av kodeenhetene samtidig, noe som kan øke effektiviteten til koden i en stor oppgave [21][49]. Og «samtidighets-problem» er generelt ikke mulig i funksjonell programmering [23]. Lignende, så har funksjonell programmering 100% selvstendig kodeenheter, der man ikke trenger å tenke på noe annet når man endrer noe, legger til noe nytt, gjenbruker funksjoner, eller bytter den ut med andre funksjoner [21][22][23], noe som gjør utviklingen av koden mye enklere. Dette er også sant når man finner ut at man trenger mindre syntaks for å sette opp hvordan de skal snakke med hverandre enn andre paradigmer som objekt-orientert programmering [48]. Som bonus innenfor effektiv prosessering av store mengder data, så har det kommet ut funksjonelle programmeringsspråk der koden klarer å kjøre nesten like fort som prosedyre-orientert programmering gjør [49], som er brukt ved å lage systemer, der det å være rask har veldig mye å si. Og til slutt, basert på hvordan programmeringsspråket håndterer verdier som absolutt trenger å ha muligheten til å bli endret av andre, så man kan også drastisk redusere søkeområde når man skal finne ut hvor det har skjedd en feil [23][39].

Objekt-orientert

Med tanke på paradigmer, så er det tro på at objekt-orientert programmering vil bli mindre viktig, mens andre paradigmer vil bli mer populært som for eksempel funksjonell programmering [43]. På en annen side er OOP allerede godt etablert i mange bedrifter, så det kan ta lang tid før bruken av det minker [43]. Et godt eksempel er programmeringsspråket Cobol, som for det meste bare brukes til å opprettholde allerede eksisterende programvare og infrastruktur [58]. Dette er videre vist til hvor mye folk som har hatt problemer med objekt-orientert programmering, der den ikke skalerer så bra [28], motstår forandringer [45], og som etter hvert fører til spagetti-kode hvis man har en for stor oppgave [24][39][40][43].

Kunstig intelligens

Når det snakkes om populariteten av funksjonell programmering, kommer man også inn på veksten av kunstig intelligens (KI). Det har vært mye snakk om KI overalt helt siden Chat GPT ble publisert, hvordan det vil forandre framtiden, og hvordan programvare-utvikling har blitt mer avhengig av kunstig intelligens [40][43][54]. Ut fra dette, har det kommet argumenter både for og mot bruken av kunstig intelligens innenfor programmering. Poenget med å bruke kunstig intelligens er å øke produktiviteten, og hvor raskt man klarer å skrive kode. Man kan også spørre kunstig intelligens om mulige feilmeldinger, eller gjøre den om til en læringsassistent til programmering [40][54][55][56]. Men her er det mange som poengterer utfordringer rundt bruken av kunstig intelligens.

For det første, hvis man bruker kode direkte generert fra en kunstig intelligens, så kan det begrense hvor mye man lærer [54][55][56][57]. Hvis man ikke bruker tid selv på å løse vanskelige problemer, som ville gjort deg til en bedre programmerer, vil det bli langt vanskeligere i fremtiden å feilrette kode, noe som utgjør en stor del innenfor programmering [54][55][56][57].

Dette leder inn i hovedproblemet, som er at kvaliteten på koden som blir generert er ganske dårlig, og uventede feil sniker seg inn uten noen feilmeldinger [55][56][57],

fordi KI ikke kan alltid være helt sikker på hva du faktisk vil ha. Dette sammen med at det hindrer læring, vil gjøre det veldig sannsynlig for feil å komme inn i koden og vil gjøre opplevelsen av å utvikle kode veldig frustrerende. Samtidig vil feilretting bli vanskelig og tidkrevende for en som ikke selv har skrevet koden [43][55][57]. Dette fører selvfølgelig til vanskeligheter å vedlikeholde koden, siden koden ikke er laget for å være like strukturert og lesbar [56], og at store kodeenheter oftere må bli skrevet om på grunn av feil i koden [55][56]. En programmerer sin opplevelse underveis i opplæring og arbeid må heller ikke undervurderes, og det sies at man må ha det hvert fall litt morsomt om man skal bli skikkelig god til å programmere [55]. Dermed er det sagt at kunstig intelligens ikke er godt egnet for problemer som er komplekse, der den kan ved uhell legger til mange feil i koden [54][55].

Men kunstig intelligens kan fortsatt bli brukt produktivt uten å redusere kvaliteten av koden. Man må bare være forsiktig på hva slags kode man tar fra KI, og man må samtidig verifisere at koden er av god kvalitet [54]. Derfor bør man i stedet for å kopiere første forslag fra kunstig intelligens og lime det direkte inn i koden, og på den måten lar KI lede hvordan koden din blir, så bør man i stedet selv lede KI frem til hvordan koden skal bli [54][55][56][57]. Dette kan gjøres ved å godta enkel kode som må repeteres med veldig lite variasjon. Som for eksempel funksjoner/objekter som ligner veldig på det du allerede har. Eller man kan godta kode ment for å teste koden [55][57]. Dette kan gjøres med mye mindre tak av kvalitet, siden da gir man kunstig intelligentes mye mer en ide over hvordan du vil at ting skal fungere, for eksempel funksjonen av en test, eller en funksjon som allerede er veldig lignende, noe som gjør at du kan styre forslagene den gir mye enklere [55][57]. Med dette, kan man mye enklere selv bestemme hvordan koden skal være, gjøre det enklere å holde fast på et paradigme, og være mer produktiv uten å redusere mye på kvaliteten. Så jeg mener KI er for det meste et redskap til å skrive mer kode i stedet for å erstatte programmerere. [43][54][55][56][57]

Bruk av kunstig intelligens innenfor selve kompilatoren

En annen mulighet i fremtiden, er bruk av kunstig intelligens innenfor selve kompilatoren [59]. Siden Fortran klarte å vise at kompilatorer genererte raskere kode

enn det en programmererne kunne realistisk gjøre [1], så kan det kanskje være mulig for en kunstig intelligens å gjøre det enda raskere enn en kompilator skrevet av programmerere. Hvis dette skjer, kan det kanskje gjøre det mindre relevant hvilket programmeringsspråk man velger med tanke på hastigheten av koden, der en kunstig intelligens kan tilpasse ethvert programmeringsspråk. Så langt er det bare fokus på å bruke det for å koble maskinvare som bruker flere kjerner samtidig, til programvare som kan effektivt bruke de kjernene samtidig [59], men det er en mulighet som bør i det minste bli vurdert.

Programmeringsspråk som oversetter til andre programmeringsspråk

Underveis i intervjuene oppdaget jeg en trend om økt fokus på programmeringsspråk som oversetter til andre språk. Altså i stedet for å bruke de dominerende programmeringsspråkene, så finnes det andre programmer som har all funksjonaliteten den kompilerer til, men kan gi et forbedret økosystem og utviklingsområde som oftest gir deg enklere syntaks å programmere med. Som for eksempel Elm til Javascript, [49], Kotlin til Java [40], Julia til Python [43], eller Python til C++ [50]. Dermed finnes det flere muligheter innenfor domene-spesifikke områder, og generelt flere valg en programmerer kan realistisk velge imellom enn det generelt blir framstilt.

Og med det, vil spørsmål relatert til fremtiden forhåpentlig ha blitt besvart, slik at nåtidens programmerere kan møte på mindre frustrasjoner gjennom et godt tilpasset programmeringsspråk.

Konklusjon

Hva gjør et programmeringsspråk godt? Dette har blitt utforsket i dokumentet for å veilede en programmerer i valget av programmeringsspråk. Vi har også diskutert forskjellige fordeler og ulemper i forhold til hva programmeringsspråket er tenkt å bli brukt til, og hvilken oppgave programmereren står overfor. For å gi programmerere redskapene de trenger til å finne ut hvilke programmeringsspråk som fungerer for

dem, har jeg først gitt leseren en bedre bakgrunnsforståelse for hvordan programmeringsspråk har utviklet seg gjennom historien, som vil bli relevant i de neste hovedkapitlene.

Historie

Det har blitt snakket om «The analytical engine» og hvordan det lagde den første programmereren og første programmet, og hva som defineres som en datamaskin og et programmeringsspråk gjennom Allan Turing og Assembly. Leseren har blitt vist at fokuset på struktur var viktig tidlig i historien, for eksempel med Algol, og hvordan den ideen spredte seg til alle andre former for programmeringsspråk. Det har blitt snakket om de ulike populære paradigmene, altså funksjonell, objekt og Message-orientert programmering gjennom Lisp, Simula og Smalltalk. Leseren har blitt vist at det finnes forskjellige domener, for eksempel systemutvikling via C og C++, noe som videreutviklet mange andre programmeringsspråk, og det har blitt snakket om mulige årsaker til hvorfor objekt-orientert ble så populært.

Nåtiden

Etter at bakgrunnsforståelsen ble satt på plass, ble det diskutert hva som gjør et programmeringsspråk godt. Det er her jeg har kommet fram til en konklusjon gjennom intervjuer, foredrag, eksperter sin mening og diskusjon som sier at det ikke finnes et «beste programmeringsspråk» eller noe «silver bullet» som automatisk leder en programmerer til et godt programmeringsspråk. Det er en grad av ulike «fokus-punkter» og domener en programmerer må velge imellom basert på oppgaven de har. Fordi i teorien, kan man bruke hvilken som helst programvare eller programmeringsspråk til å programmere, man kan til og med programmere ved å bruke Excel hvis man ønsker, selv om det ikke hadde vært så effektivt. Det som derimot har noe å si, er hvor fri man er til å bruke programmeringsspråket til å gjøre det man vil, og hvor enkelt det er å programmere basert på sin oppgave. Jeg har dermed laget noen steg en programmerer kan ta basert på det vi har gått gjennom for å finne et passende programmeringsspråk.

Når en programmerer skal velge et programmeringsspråk, så starter man med faktoren som har størst påvirkning: Er oppgaven innenfor et domene? For eksempel innenfor systemutvikling, kunstig intelligens, webdesign, databaser og så videre. Dermed kan man gjøre søket etter et programmeringsspråk langt enklere ved at det finnes programmeringsspråk innenfor disse spesifikke domenene. Samtidig kan man også undersøke om oppgaven i det hele tatt trenger et programmeringsspråk, der det noen ganger finnes programvare som kan være et mer aktuelt verktøy til oppgaven, som ble diskutert i database eksemplet.

Etter at man har bekreftet at oppgaven trenger et programmeringsspråk, går man videre til hva man bør fokusere på innenfor oppgaven, om fokuset bør være på oversiktlighet, effektivitet, rask kode, eller en balanse av disse. Det er her programmereren bør se på oppgaven man har, og definere fokusområdet basert på den. Her bør programmereren også velge et paradigme eller stil som passer vedkommende best basert på hvordan de tenker/fungerer, slik at det ikke blir så frustrerende å arbeide innenfor et programmeringsspråk som ikke sammenfaller med sine ønsker over hva et programmeringsspråk kan gjøre.

Til slutt, etter å ha gått gjennom domene-spesifikke programmeringsspråk, og en egen stil/paradigme som er passende, bør en programmerer vurdere de spesifikke forskjellene mellom de individuelle programmeringsspråkene personen står igjen med. Disse forskjellene diskuteres ikke her, da programmeringsspråk forandrer seg hele tiden.

Fremtiden

Veien til et passende programmeringsspråk for en programmerer er nå nesten ferdig, hvor det til slutt ble diskutert mulige situasjoner fremtiden kan bringe, for å hindre at nåtidens programmerere blir fastlåst i et utdatert programmeringsspråk. Det har blitt diskutert populariteten av ulike paradigmer, der objekt-orientering muligens ikke vil bli like relevant i fremtiden på grunn av dets problemer, samtidig som at funksjonell programmering vil bli mer populær på grunn av dets effektive databehandling ved

store mengder av data. Lignende ble det diskutert bruken av kunstig intelligens, og hvordan det kan forandre på hvordan programmerere jobber innenfor programmering, der vi kom til en konklusjon at KI er for det meste et redskap til å skrive mer kode i stedet for å erstatte programmerere. Det har også blitt diskutert om kunstig intelligens inne i selve kompilatoren, der det kan blir mindre relevant hvilket programmeringsspråk man velger med tanke på hastigheten av koden, siden en kunstig intelligens kan ha muligheten til å tilpasse til ethvert programmeringsspråk. Og det har blitt påpekt at det har vært et økt fokus på programmeringsspråk som oversetter til andre språk. Noe som gjør at det finnes flere muligheter innenfor domene-spesifikke områder, og generelt flere valg en programmerer kan realistisk velge imellom enn det generelt blir framstilt. Med dette, kan programmerere enklere velge et sett av programmeringsspråk slik at de kan fremme effektiv progresjon.

Referanser

[1] Rendle, M. (2023, mai 26). *Locknote: How JavaScript Happened: A Short History of Programming Languages.* Retrieved from Youtube: https://www.youtube.com/watch?v=hEdzaIa4Heg

[2] Sammet, J. E. (1972, juli). *Programming languages: history and future*. Retrieved from https://dl.acm.org/doi/pdf/10.1145/361454.361485

[3] Sammet, J. E. (1991). *Some Approaches to, and Illustrations of, Programming Language History*. Retrieved from https://ieeexplore.ieee.org/abstract/document/4638280

[4] Berger, E. (2019, sep 15). *Performance Matters*. Retrieved from Youtube: https://www.youtube.com/watch?v=r-TLSBdHe1A

[5] BROMLEY, A. G. (1982, juli). *Charles Babbage's Analytical Engine, 1838*. Retrieved from WaybackMachine: https://web.archive.org/web/20150514005437/http://athena.union.edu/~hemmendd/Courses/cs80/an-engine.pdf

[6] Science Museum. (2023, juli 18). *Charles Babbage's Difference engines and The Science Museum*. Hentet fra Science Museum: https://www.sciencemuseum.org.uk/objects-and-stories/charles-babbages-difference-engines-and-science-museum#the-analytical-engine-

[7] Graham-Cumming, J. (2010, des 15). *Let's build Babbage's ultimate mechanical computer*. Hentet fra NewScientist: https://www.newscientist.com/article/mg20827915-500-lets-build-babbages-ultimate-mechanical-computer/

[8] Computer History Museum. (n.d.). *The Engines*. Retrieved from Computer History Museum: https://www.computerhistory.org/babbage/engines/

[9] Computerphile. (2016, desember 21). *Computer Science's Wonder Woman: Ada Lovelace - Computerphile*. Retrieved from Youtube: https://www.youtube.com/watch?v=wnHHzBY1SPQ

[10] Fuegi, J., & Francis, J. (2003, November). *Lovelace & Babbage and the creation of the 1843 'notes'*. Retrieved from IEEE Xplore: https://ieeexplore.ieee.org/document/1253887

[11] Wikipedia. (n.d.). *Programmeringsspråk*. Retrieved from Wikipedia: https://no.wikipedia.org/wiki/Programmeringsspr%C3%A5k

[12] Newman, M. H. (1955, Nov 01). *Alan Mathison Turing, 1912-1954*. Retrieved from The Royal Society: https://royalsocietypublishing.org/doi/pdf/10.1098/rsbm.1955.0019

[13] Computerphile. (2016, Juli 05). *Turing Complete - Computerphile*. Retrieved from Youtube: https://www.youtube.com/watch?v=RPQD7-AOjMI

[14] Deitel, H. M., Deitel, P. J., & Choffnes, D. R. (2004). *Operating Systems (3rd Edition)*. Retrieved from Dronacharya: https://gnindia.dronacharya.info/CSEIT/4thSem/Downloads/OperatingSystems/Books/Operating-systems-text-book-3.pdf

[15] Backus, J. (1978, Aug 08). *The History of Fortran I, II, and III*. Retrieved from Software Preservation Group: https://dl.acm.org/doi/pdf/10.1145/960118.808380

[16] Rojas, R., Cuneyt Goktekin, G. F., & Kruger, M. (2000, Feb). *Plankalkul: The First HIgh-Level Programming Language and it's Implementation*. Retrieved from Department of Mathematics and Computer Science: https://ftp.mi.fu-berlin.de/pub/reports/TR-B-00-03.pdf

[17] Archer, B. (2016, Nov 16). *Assembly Language For Students*. Retrieved from Digital Library: https://dl.acm.org/doi/book/10.5555/3125846

[18] *How do assembly languages depend on operating systems?* (2011, Aug). Retrieved from Stack Overflow: https://stackoverflow.com/questions/6859348/how-do-assembly-languages-depend-on-operating-systems

[19] BACKUS, J. W., et al. (1960, June 28). *Report on the Algorithmic Language ALGOL 60*. Retrieved from Digital Library: https://dl.acm.org/doi/pdf/10.1145/367236.367262

[20] McCarthy, J. (1981). *LISP Session*. Retrieved from Digital Library: https://dl.acm.org/doi/pdf/10.1145/800025.1198360

[21] Feldman, R. (2019, Sep 30). *Why Isn't Functional Programming the Norm? – Richard Feldman*. Retrieved from Youtube: https://www.youtube.com/watch?v=QyJZzq0v7Z4

[22] Hughes, J. (1990). *Instutisjonen for Datavitenskap*. Retrieved from Why Functional Programming Matters: https://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf

[23] Feldman, R. (2022, Jan 12). *Functional Programming for Pragmatists • Richard Feldman • GOTO 2021*. Retrieved from Youtube: https://www.youtube.com/watch?v=3n17wHe5wEw

[24] Will, B. (2016, jan 18). *Object-Oriented Programming is Bad*. Retrieved from Youtube: https://www.youtube.com/watch?v=QM1iUe6IofM

[25] *Object Oriented Hell*. (2015, Juni 09). Retrieved from Bits and Pices: https://bitsandpieces.it/posts/object-oriented-hell/

[26] Ram, S. (2003, Juli 23). *Dr. Alan Kay on the Meaning of "Object-Oriented Programming"*. Retrieved from Purl: https://www.purl.org/stefan_ram/pub/doc_kay_oop_en

[27] Feldman, R. (2024, Jan 08). *Discussing Roc and functional systems with Richard Feldman | Backend Banter 035*. Retrieved from Youtube: https://www.youtube.com/watch?v=LC1yxlicWUs

[28] Eng, R. K. (2019). *How did SIMULA contribute to object oriented programming?* Retrieved from Quora: https://www.quora.com/How-did-SIMULA-contribute-to-object-oriented-programming

[29] Krol, M. (2015). *Why do many software engineers not like Java?* Retrieved from Quora: https://www.quora.com/Why-do-many-software-engineers-not-like-Java

[30] Ritchie, Dennis M., et al. (1978). *The C Programming Language*. Retrieved from Scholar: https://scholar.googleusercontent.com/scholar?q=cache:rJV3igI_08kJ:scholar.google.com/&hl=en&as_sdt=0,5

[31] Stoustrup, B. (1999). *An Overview of the C++ Programming Language*. Retrieved from UMBC: https://redirect.cs.umbc.edu/courses/undergraduate/CMSC331/resources/papers/Stroustrup%20-%201999%20-%20An%20Overview%20of%20the%20C%20Programming%20Language-annotated.pdf

[32] Stroustrup, B. (1994, Mars). *The Design of C++ , lecture by Bjarne Stroustrup*. Retrieved from Youtube: https://www.youtube.com/watch?v=69edOm889V4

[33] Jensen, R. (2020, Sep 12). *"A damn stupid thing to do"—the origins of C*. Retrieved from Arstechnica: https://arstechnica.com/features/2020/12/a-damn-stupid-thing-to-do-the-origins-of-c/

[34] *C – THE MOTHER OF ALL LANGUAGES*. (2021, Mai 31). Retrieved from WaybackMachine: https://web.archive.org/web/20210531161841/https://ict.iitk.ac.in/c-the-mother-of-all-languages/

[35] Lawson, J. (2022). *Why do many software engineers not like Java?* Retrieved from Quora: https://www.quora.com/Why-do-many-software-engineers-not-like-Java

[36] Huw. (2023, Mars 20). *Object Orientation is NOT about Objects. The Secret Most Programmers Don't Know!* Retrieved from Youtube: https://www.youtube.com/watch?v=lmAarC0Zhq4

[37] Lewis, S. (2018). *What does Alan Kay think of the Swift programming language?* Retrieved from Quora: https://www.quora.com/What-does-Alan-Kay-think-of-the-Swift-programming-language

[38] Txreq. (2024, Jan 05). *Yet Another ULTIMATE Programming Languages Tier List (2024)*. Retrieved from Youtube: https://www.youtube.com/watch?v=AQccsZpEgYY

[39] Olsen, R. (2018, Nov 09). *Functional Programming in 40 Minutes • Russ Olsen • GOTO 2018*. Retrieved from Youtube: https://www.youtube.com/watch?v=0if71HOyVjY

[40] Tesdal, N. (2024, Mars 11). (M. A. Dragsjø, Interviewer)

[41] Montecchi, L. (2024, Mars 18). (M. A. Dragsjø, Interviewer)

[42] Cherubin, S. (2024, Apr 08). (M. A. Dragsjø, Interviewer)

[43] Hetland, M. L. (2024, Apr 17). (M. A. Dragsjø, Interviewer)

[44] Burnham, C. (2018). *Why is Python so bad?* Retrieved from Quora: https://www.quora.com/Why-is-Python-so-bad

[45] Vakil, A. (2020, Nov 05). *Object Oriented Programming is not what I thought - Talk by Anjana Vakil*. Retrieved from Youtube: https://www.youtube.com/watch?v=TbP2B1ijWr8

[46] Will, B. (2019, Mars 08). *Object-Oriented Programming is Good**. Retrieved from Youtube: https://www.youtube.com/watch?v=0iyB0_qPvWk

[47] juviler, J. (2022, Mars 30). *Static vs. Dynamic Websites: Here's the Difference*. Retrieved from HobSpot: https://blog.hubspot.com/website/static-vs-dynamic-website

[48] Bob, U. (2023, Aug 19). *Uncle Bob LOVES Functional Programming | Prime Reacts*. Retrieved from Youtube: https://www.youtube.com/watch?v=GcJgGy-dfvE

[49] Feldman, R. (2021, Okt 20). *"Outperforming Imperative with Pure Functional Languages" by Richard Feldman*. Retrieved from Youtube: https://www.youtube.com/watch?v=vzfy4EKwG_Y

[50] Queralt, M. G. (2020). *Which is the most used programming language for artificial intelligence?* Retrieved from Quora: https://www.quora.com/What-is-best-programming-language-for-Artificial-Intelligence-projects/answers/132323791

[51] Indeed. (2023, Sep 25). *Types of Database Languages and Their Uses (Plus Examples)*. Retrieved from Indeed: https://www.indeed.com/career-advice/career-development/database-languages

[52] Corless, P. (2021). *Is NoSQL a programming language?* Retrieved from Quora: https://www.quora.com/Is-NoSQL-a-programming-language

[53] Liu, J. (2019). *What does it mean when a programming language is "faster" than another?* Retrieved from Quora: https://www.quora.com/What-does-it-mean-when-a-programming-language-is-faster-than-another

[54] Rouhani, M. (2024, mai 03). How Does Artificial Intelligence Change the Way We Teach Programming?

[55] DreamsOfCode. (2024, April 06). *Why I Quit Copilot | Prime Reacts*. Retrieved from Youtube: https://www.youtube.com/watch?v=GkmUwDXvWiQ

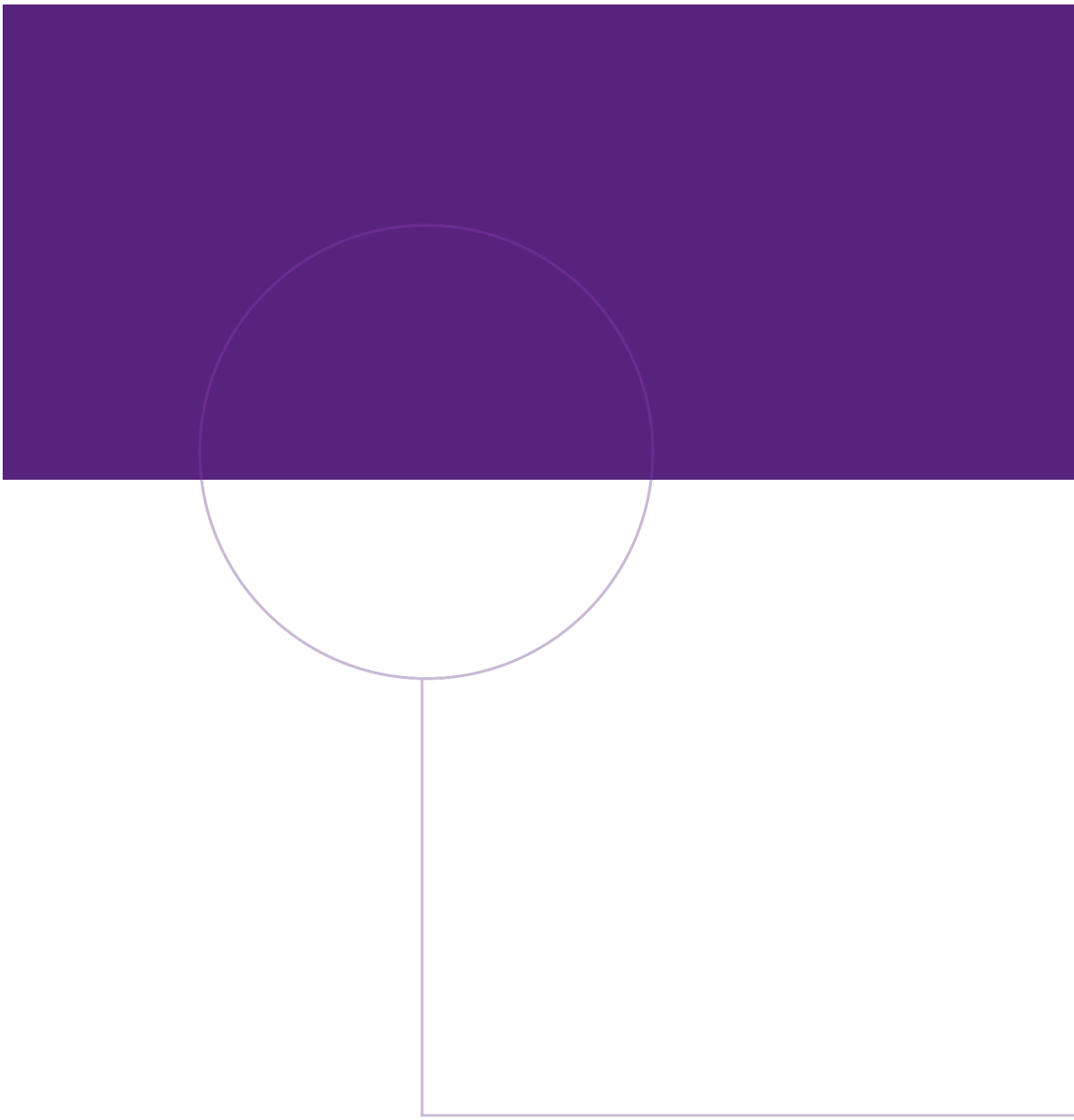
[56] Harding, W., & Kloster, M. (2024, Feb 12). *GitHub CoPilot Is Ruining Code Quality | Prime Reacts*. Retrieved from Youtube: https://www.youtube.com/watch?v=3h-VOo_3J54

[57] ThePrimeagen. (2023, Feb 24). *CoPilot Review: My Thoughts After 6 Months*. Retrieved from Youtube: https://www.youtube.com/watch?v=RDd71IUIgpg

[58] Wikipedia. (n.d.). *COBOL*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/COBOL

[59] O'Boyle, M. (2023, Jan 17). *Rethinking how we build compilers in a heterogeneous world*. Retrieved from PARMA-DITAM workshop: https://parma-ditam-workshop.github.io/2023/slides/oboyle.pdf

[60] ThePrimeTime. (2023, Des 20). *3D Rollercoaster in Excel | Prime Reacts*. Hentet fra Youtube: https://www.youtube.com/watch?v=9HvmqzKgS54



NTNU

Kunnskap for en bedre verden