STUDENTRAPPORT

NTNU

KANDIDATNUMMER(E)/NAVN:
Sofiya Yasim Ibrahim Ali - 10052

DATO:	FAGKODE:	STUDIUM:	ANT SIDER/BILAG:
10.11.23	IDATG1003	BIDATA – Dataingeniør	1 /14

FAGLÆRER(E) :

Kiran Raja

TITTEL :

TrainDeparture Application

SAMMENDRAG:

TrainDeparture Application er et resultat av teoretiske- programmeringskonsepter gjennom emnet Programmering 1 ved NTNU.

Applikasjonen er et brukervennlig produkt, for togavgangshåndtering som kan tas i bruk for administrering av toginformasjon. Denne rapporten viser ulike faser og aspekter av utviklingsprosessen.

Bakgrunnen for de forskjellige løsningene som er implementert i koden, presenteres i denne rapporten. Dette er en innsikt på hvordan man kan håndtere komplekse systemer, ved å ta i bruk teoretisk kunnskap som gjør applikasjonen mer robust.

INNHOLD

1 SAMMENDRAG

I denne rapporten blir det presentert arbeidet som har blitt gjort, med utviklingen av TrainDeparture Application, en applikasjon som håndterer forskjellige aspekter ved togtransport. Den tilrettelegger for administrering av blant annet avgangstider, tognummer, spor og relaterte detaljer. Gjennom utviklingen av applikasjonen har det vært fokus på brukervennlighet. Derfor har det også vært fokus på prinsipper som høy kohesjon og lav kobling for god kodestruktur. Denne rapporten gir en bakgrunn for alle valgene som ble tatt gjennom prosessen. Design og resultater blir diskutert, i tillegg til eventuelle begrensninger, utfordringer og triumfer. Det blir en helhetlig vurdering av det ferdigstilte prosjektet, men utviklingsfasen og erfaringer blir også presentert.

2 INNLEDNING – PROBLEMSTILLING

2.1 Formål og bakgrunn

Selve applikasjonen tar sikte på å la brukerne å administrere informasjon om togavganger. Dette inkluderer å skape løsninger som tillater brukere å søke etter, legge til og endre informasjon om togavganger, inkludert forsinkelse, avgangstider og tilordning av spor.
Formålet med denne applikasjonen er å tilby brukere en brukervennlig applikasjon for å administrere togavganger. Gjennom et oversiktlig grensesnitt vil brukerne kunne håndtere forskjellig informasjon om togavganger, basert på forskjellige kriterier.

2.2 Avgrensninger

I utviklingen av denne applikasjonen er det foretatt noen rammer og avgrensinger for å sikre en mer fokusert og målrettet tilnærming. Systemet støtter først og fremst bare en bestemt stasjon, og tar kun hensyn til togavganger som har sin reise fra den spesifikke stasjonen.
Tidsrammen i denne applikasjonen er også svært begrenset, da det bare fokuserer på tidspunktet for togavganger innenfor en enkel dag, og ikke tar hensyn til dato. En annen begrensning med klokkefunksjonen, er at systemets klokke kan bli oppdatert manuelt av brukerne. Det er ingen systematisk oppdatering til det virkelige klokkeslettet hvis klokken har blitt oppdatert manuelt via brukergrensesnittet.

2.3 Begreper

Begrep (Norsk)	Begrep (Engelsk)	Betydning/beskrivelse
Togavganger	Train departures	Informasjon om avgangstider for tog.
Forsinkelse	Delay	Tidsforsinkelse for avgang eller ankomst.
Linje	Line	Spesifikt togrutenummer, som f.eks. RE30 eller L1.
Destinasjon	Destination	Sluttsted for togreisen.
Avgangstid	Departure time	Avgangstidspunkt for toget
Spornummer	Track number	Nummer for togets avgangsspor.
Tognummer	Train number	Unikt identifikasjonsnummer for hvert tog.
Meny	Menu	En liste med tilgjengelige alternativer

2.4 Rapportens oppbygning

I denne rapporten kommer relevant teoretisk grunnlag som ble brukt for å fullføre dette prosjektet. Deretter kommer metodene som formet designet og implementasjonen av prosjektet. Det blir også beskrevet planlegging, utviklingsmetoder og verktøy som ble benyttet for å løse oppgaven.
Resultatene til arbeidet blir presentert, inkludert sluttproduktets funksjonalitet og endringer av opprinnelig kode med tanke på forbedring av kvalitet, som robusthet og brukervennlighet. Videre kommer et drøftingskapittel med diskusjon rundt arbeidet, som endringer, feilkilder og tanker rundt kvalitet. I tillegg til kildekritikk gjennom arbeidet og lærdom bak prinsipper innenfor programmering.
Mot slutten av rapporten følger konklusjonen, som oppsummerer de viktigste resultatene og erfaringene fra arbeidet. Denne delen utpeker også eventuelle forbedringer for fremtidige prosjekter. Til slutt finner man en referanseliste som inkluderer kildene som er referert til i rapporten. Denne strukturen viser en oversiktlig og sammenhengende gjennomgang av prosjektet, med fokus på ulike faser og aspekter av arbeidet.

3 TEORETISK GRUNNLAG

Kohesjon er en av de prinsippene innenfor programvareutvikling som referere til graden av funksjonell samhørighet innad i en enhet av kode, som kan være en klasse eller en metode. Essensen bak prinsippet kohesjon er at en enhet av en kode bare skal ha ansvar for en veldefinert oppgave (Barnes, David J & Kölling, Michael, 261). Dette kalles høy kohesjon eller «high cohesion» på engelsk, og er en viktig praksis i god programvareutvikling siden det styrker kodens robusthet (Barnes, David J & Kölling, Michael, 281).

Kobling refererer til graden av avhengighet mellom ulike deler av et program, og hvor tett disse delene er knyttet til hverandre. Lav kobling eller «low coupling» på engelsk, betyr at endringer i en del av koden ikke vil påvirke andre deler. Dette er et mål for å styrke robustheten til et program, og i motsetning innebærer høy kobling en sterk avhengighet, der endringer i en del kan føre til endringer i andre deler. (Barnes, David J & Kölling, Michael, 259).

Tydelig definerte arbeidsområder gjør ikke koden bare enklere å lese, men også å forstå og vedlikeholde. Når hver metode og klasse har en spesifikk rolle, fører det til at utvikling av koden blir mer lesbar for meg og andre utviklere. (Barnes, David J & Kölling, Michael, 279)

ArrayList og HashMap er noen av de mest nyttige Javastandardbibliotekene. ArrayList tillater enkel tilgang til elementer i en liste ved hjelp av indekser, og har flere metoder for å legge til, fjerne og endre data (Barnes, David J & Kölling, Michael, 110). HashMap derimot, organiserer data i nøkkel-verdi par og er nyttig for å hente og lagre data basert på nøkler (Barnes, David J & Kölling, Michael, 191).

Modularisering og abstrahering er to viktige konsepter i programvareutvikling. Modularisering dreier seg om å håndtere komplekse problemstillinger ved å dele det opp i mindre biter. Abstrahering betyr å fokusere på individuelle problemer, og ignorere detaljene. Dette fører til at man kan jobbe med et komplekst system innenfor programvareutvikling, samtidig som man skaper struktur for at spesifikke deler kan bli håndtert separat. (Barnes, David J & Kölling, Michael, 69)

4 METODE – DESIGN

Tanken bak metodene og designet i utviklingen av TrainDeparture prosjektet, innebar å prøve å utvikle en effektiv løsning for administrering av togavganger. Fremgangsmåten inkluderte å lage nødvendige klasser med eget ansvar. Hovedklassene var TrainDeparture og TrainRegister, som ble designet med egne ansvarsområder. TrainDeparture klassen ble designet for lagring av togavgangsinformasjon, mens TrainRegister ble designet for håndtering av tog registeret og bruk av brukergrensesnitt. Bruk av testklasser var også en del av utviklingsprosessen, for å sikre at koden sin funksjonalitet hadde full dekning. JUnit er testrammeverket som ble brukt for disse enhetstestene, og dette sikret kvaliteten på koden.
For å holde styr på utviklingsprosessen, ble versjonskontroll benyttet med sentral repository i GitHub. Det ble brukt objektorientert programmering med Java, og benyttet IntelliJ IDEA 2023.2 som utviklingsprogram gjennom hele prosjektet. Dokumentasjon av koden inkluderte dokumentasjon med JavaDoc-standard og kommentarer ved kodesnutter der det behøvde mer utdypning. Utover i prosjektet ble CheckStyle og SonarLint også brukt som verktøy for analyse, for å oppnå best mulig kodekvalitet med god dokumentasjon.

5 RESULTATER

Første klassen jeg valgte å opprette i prosjektet var TrainDeparture, for å opprette og lagre togavgangsinformasjon. Videre ble TrainDepartureInitializer klassen utviklet, som skulle initiere de forskjellige togavgangene, for å teste ut applikasjonen og for å ha default-innhold med i togregisteret.

Registerklassen ble kalt TrainSchedule i første versjonen av prosjektet, og inneholdt metodene som nå er delt mellom TrainRegister og TrainRegisterUI. Avgjørelsen om å dele klassene var for å oppnå lav kobling mellom ansvarsområdene, dermed ble de delt opp i klassene TrainSchedule og TrainScheduleUI. Mot slutten av prosjektet bestemte jeg meg å endre navnene til TrainRegister og TrainRegisterUI, siden navnene var mer tydelig med tanke på at de begge hadde ansvar for et register av togavganger, og at TrainRegisterUI hadde ansvar for brukergrensesnitt.

ArrayList blir brukt i TrainRegister klassen for å lagre togavgangene. Det var et bevisst valg, siden metodene i denne klassen gjør det da mulig å manipulere data for togavganger basert på ulike kriterier. På den andre siden er HashMap også brukt i TrainRegister klassen for å lagre togavganger basert på unike tognummer. Dette ble

implementert siden det tillater raskere tilgang til avganger med tognummeret som nøkkel, og er derfor mer ideelt å bruke enn ArrayList.

To klasser som ble opprettet sent inn mot prosjektet er TrainPrinter- og InputHandler klassene. TrainPrinter skulle ha ansvar for utskrift av togavgangsinformasjon, mens InputHandler skulle ha ansvar for å validere brukerens input. Dette bidro til lav kobling ved at andre klasser ikke trenger å bekymre seg for validering av brukerinput eller hvordan utskrift skal håndteres.

I prosjektet ble modularisering implementert med å gi hver klasse en spesifikk rolle. Metodene i klassen TrainRegister blant annet legger til avganger, søker etter avganger basert på tognummer og destinasjon og utfører andre operasjoner relatert til administrasjon av togavganger.

TrainDeparture klassen er abstrahert for å representere informasjon om en enkelt togavgang. Klassen har metoder for å sette togspor og legge til forsinkelser til togavganger, men skjuler kompleksiteten bak hvordan handlingene utføres. Da kan klassen TrainRegister samhandle med TrainDeparture – objekter uten å bekymre seg for implementasjonsdetaljene.

Gjennom utviklingen av prosjektet ble CheckStyle brukt hyppig for å få god JavaDoc-dokumentasjon i koden. CheckStyle ga gode tilbakemeldinger om hvor i koden dokumentasjonen kunne forbedres.

Det er brukt JUnit for enhetstesting for klassene TrainDeparture og TrainRegister. Det er opprettet både negative og positive tester, med bruk av feilsøking metoder som assertEquals, assertNotEquals og assertFalse for å sikre at funksjonaliteten til klassene er samspilt med forventningene deres.

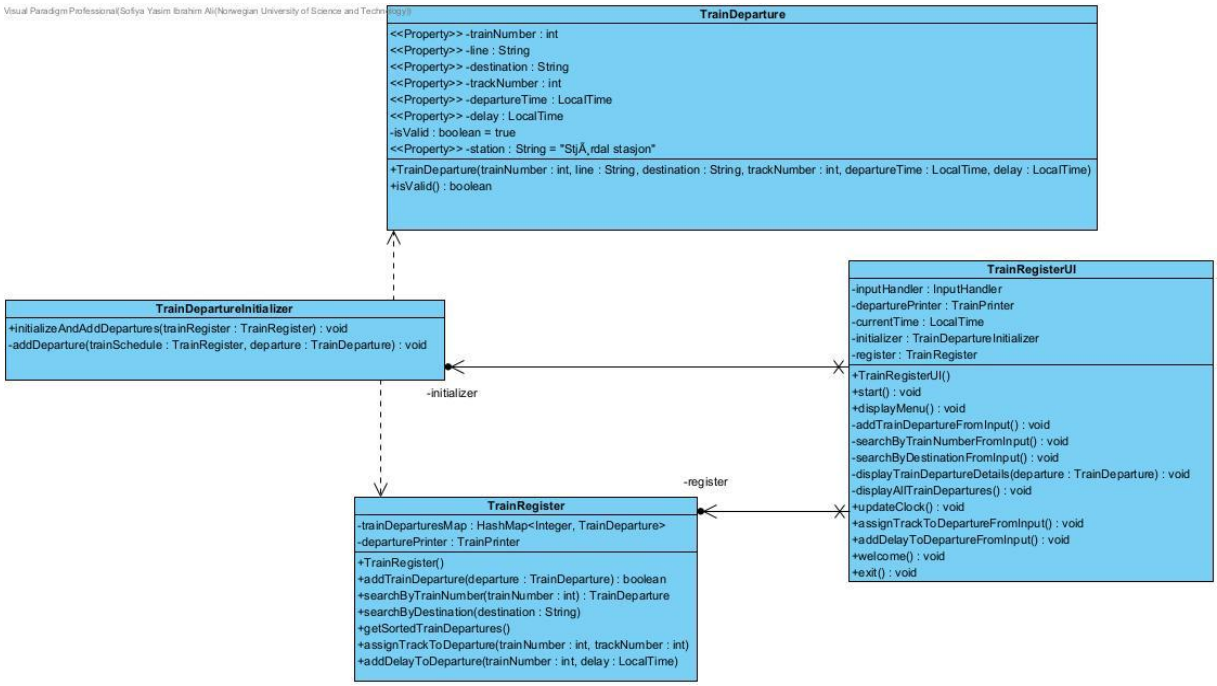
TrainDepartureTest klassen tester at alle grunnleggende 'get'- og 'set' – metodene fungerer som forventet, siden de er kritiske funksjoner som påvirker alle klassene.

TrainRegisterTest klassen tester at metodene som brukerne kan velge i menyen, fungerer som forventet. Det blir blant annet testet om søkemetodene fungerer som forventet, og om systemet legger til forsinkelser korrekt for avganger.

Å teste slike tilfeller er viktig for å sikre at systemet håndterer ugyldige verdier på riktig måte. Derfor er det viktig å lage både positive og negative tester, slik at vi vet om vi får ugyldige data og om vi setter verdier korrekt.

Applikasjonen er også en idiotsikker programvare, da det er implementert logikk i koden for å håndtere forskjellige scenarioer. Hvis en togavgang allerede eksisterer for eksempel, vil ikke den dupliseres hvis brukeren legger til en ny en med samme

tognummer. Brukeren får også tydelig beskjeder hvis de legger til ugyldig input, dermed forsøker applikasjonen å minimere muligheten for ugyldige handlinger.



Klassediagram – 'TrainDeparture', 'TrainDepartureInitializer', 'TrainRegister' og 'TrainRegisterUI'

Dette klassediagrammet viser forholdet mellom de viktigste klassene i TrainDeparture Application prosjektet. TrainDeparture klassen representerer togavgangsinformasjon, som linje, destinasjon, avgangstid, tognummer, spor, stasjon og forsinkelse. TrainDepartureInitializer er en hjelpeklasse som leger til eksisterende togavganger i togregisteret. TrainRegister klassen administrerer en liste av TrainDeparture-objekter , med funksjoner som å søke, legge til og sortere togavgangsinformasjon. TrainRegisterUI håndterer brukergrensesnittet og kommuniserer med TrainRegister gjennom brukerinput og tilbyr menyalternativer. Alle disse klassene samhandler på forskjellige måter for å administrere togavganger på en effektiv og brukervennlig måte.

Oversikt over klassene i «TrainDeparture Application» systemet:

Klasse:	Ansvar i systemet:	Funksjonalitet:
TrainDeparture	*Lagrer og oppretter togavgangsinformasjon.*	Registrerer og lagrer informasjon om individuelle togavganger.
TrainDepartureInitializer	*Initialiserer egenvalgt togavgangsdata.*	Initialiserer togavganger, som default-innhold ved oppstart av applikasjonen.
TrainRegister	*Administrerer togregisteret.*	Lagrer og håndterer operasjoner for togavganger. Har metoder som legger til, søker etter, sorterer og redigerer togavgang i registeret.
TrainRegisterUI	*Brukergrensesnitt spesifikt for togregisteret.*	Presenterer data om tog og gir brukeren mulighet til å utføre handlinger som registrering av tog.
TrainPrinter	*Utskrift av togavgangs – informasjon.*	Skriver ut informasjon om tog og togavganger på konsollen.
InputHandler	*Behandler og validerer brukerens input.*	Håndterer brukerens valg av input, ved å sikre at all input-data er korrekt.

Brukerveiledning for «TrainDeparture Application»:

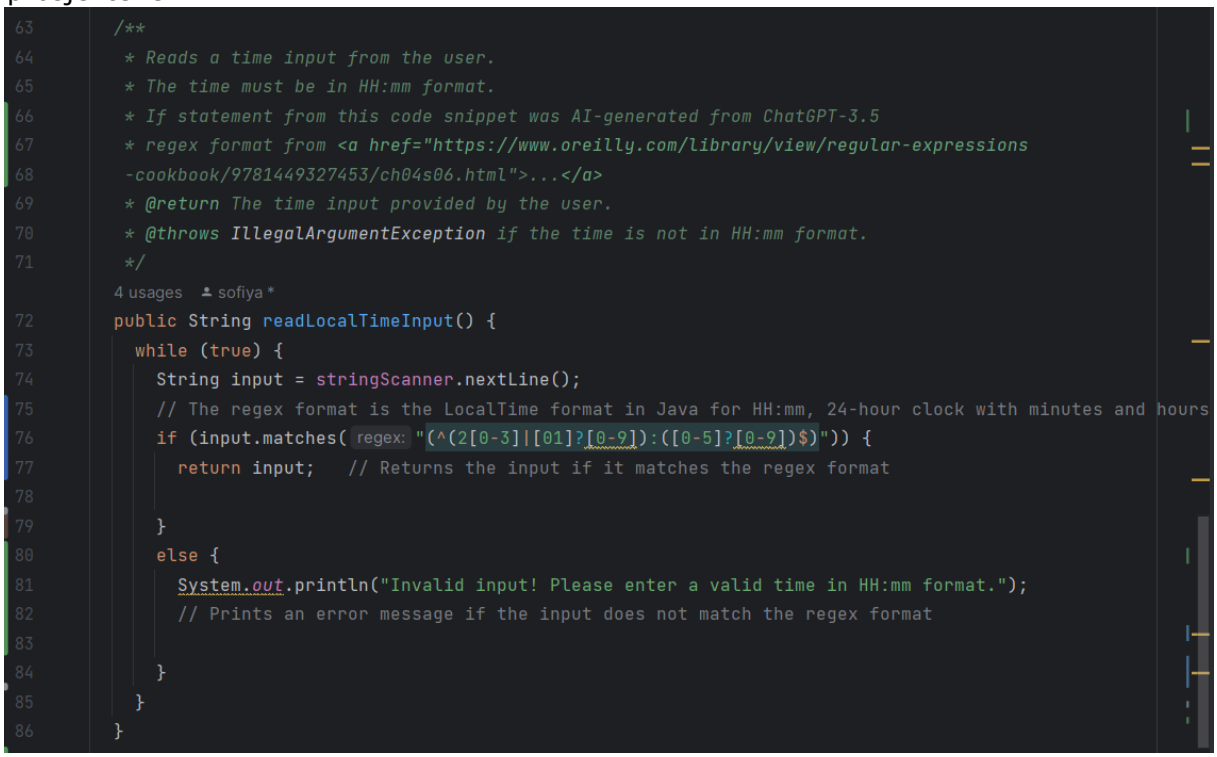
Handling:	Trinn og instrukser:
1. Add Train Departure	1. Velg alternativ 1 for å legge til en ny togavgang. 2. Følg instruksene for å oppgi informasjon som avgangstid, linje, tognummer, togspor, destinasjon og forsinkelse til den nye togavgangen.
2. Search by Train Number	1. Velg alternativ 2 for å søke etter en togavgang basert på tognummer. 2. Følg instruksene og legg til tognummeret til togavgangen du søker etter.
3. Search by Destination	1. Velg alternativ 3 for å søke etter en togavgang basert på destinasjonen. 2. Følg instruksene og legg til destinasjonen til togavgangen du søker etter.
4. Display All Departures	1. Velg alternativ 4 for å vise/skrive ut oversikten over alle togavganger. 2. Bare oversikten over togavganger i fremtiden skrives ut.
5. Assign Track to Departure	1. Velg alternativ 5 for å tildele et spor til en bestemt togavgang. Spornummeret kan bare være heltall. 2. Følg instruksene og oppgi tognummeret til det tilsvarende spornummeret du vil tildele.
6. Add Delay to Departure	1. Velg alternativ 6 for å legge til forsinkelse på en togavgang. 2. Følg instruksene og oppgi tognummeret til togavgangen du vil legge til en forsinkelse.
7. Update Clock	1. Velg alternativ 7 for å oppdatere klokken 2. Du kan ikke oppgi et klokkeslett tidligere på dagen.
8. Exit	1. Velg alternativ 8 for å avslutte applikasjonen 2. Skriv inn Y for «Yes» for å avslutte eller N for «No» når du ser en bekreftelsesmelding.

6 DRØFTING

I utviklingen av TrainDeparture Application ble en prinsippene om høy kohesjon og lav kobling brukt, for å få en strukturert og robust løsning. Enhetstestene med JUnit sikret at flere kodesnutter fungerte som forventet. SonarLint og Checkstyle ble brukt for å forbedre den generelle kodekvaliteten. Applikasjonen demonstrerte også en viss grad av idiotsikkerhet, siden den minimerte muligheten for ugyldige handlinger.

Begrensingene i prosjektet, som nevnt tidligere, knyttet til stasjonsvalg og tidsrammer, påvirket applikasjonens potensielle funksjonalitet. Det førte til at systemet kun støttet en bestemt stasjon som avreisestasjon, og håndterte bare togavganger innenfor en enkelt dag, uten hensyn til dato. Underveis ble det gjort mye tilpasninger fra den opprinnelige planen, med annerledes struktur på klassene for å vedlikeholde koden på best mulig måte.

Gjennom utviklingen av koden, brukte jeg docs.oracle.com ganske mye. Hvis ikke var læreboka en solid og troverdig kilde som ble brukt for alt teoretisk grunnlag. I tillegg så jeg på mange av de tidligere prosjektene fra Programmering 1 forelesningene med foreleser Kiran Raja. Det hjalp meg svært mye, siden vi hadde hatt forelesninger om det meste som måtte til for å utvikle denne applikasjonen. De fleste implementasjonene var inspirert fra tidligere forelesninger som TelephoneBookApp og CarRegisterApp prosjektene.



```
63      /**
64       * Reads a time input from the user.
65       * The time must be in HH:mm format.
66       * If statement from this code snippet was AI-generated from ChatGPT-3.5
67       * regex format from <a href="https://www.oreilly.com/library/view/regular-expressions
68       -cookbook/9781449327453/ch04s06.html">...</a>
69       * @return The time input provided by the user.
70       * @throws IllegalArgumentException if the time is not in HH:mm format.
71       */
         4 usages  sofiya *
72      public String readLocalTimeInput() {
73        while (true) {
74          String input = stringScanner.nextLine();
75          // The regex format is the LocalTime format in Java for HH:mm, 24-hour clock with minutes and hours
76          if (input.matches( regex: "(^(2[0-3]|[01]?[0-9]):([0-5]?[0-9])$")) {
77            return input;   // Returns the input if it matches the regex format
78
79          }
80          else {
81            System.out.println("Invalid input! Please enter a valid time in HH:mm format.");
82            // Prints an error message if the input does not match the regex format
83
84          }
85        }
86      }
```

Skjermdump: 'readLocalTimeInput' - metode fra InputHandler klassen

I InputHandler klassen ville jeg opprette en metode som tok hensyn til validering av LocalTime, så jeg sendte 'readStringInput' metoden jeg hadde til ChatGPT 3.5 og spurte om det var mulig å bruke kodesnutten. Da fikk jeg en tilbakemelding med en ny ferdigstilt metode med navn 'readLocalTimeInput'. Det den gjorde var å bruke 'matches' funksjonen til klassen String for å sjekke om innputtet fra brukeren var lik regex Hh:mm formatet. Når jeg kjørte koden, funket ikke denne metoden, og jeg tenkte det kunne vært regex-formatet som var feil, siden koden så ganske riktig ut. Etter å ha funnet en nettside fant jeg riktig regex-format for LocalTime og implementerte det i koden.

Dette var noe som fikk meg til å innse at AI hjelper godt hvis man står fast i en oppgave, men at man samtidig må være kildekritisk og teste forslagene. Jeg brukte også CoPilot for hurtigere JavaDoc dokumentasjon av koden, men den kom med forslag som var helt urelatert til det jeg skulle skrive iblant. Derfor måtte jeg passe på å ikke akseptere alt det den kom med, men likevel avsluttet den en god del setninger for meg når jeg først skrev noe.

Å strukturere applikasjonen med hensyn til klassenes ansvarsområder var det mest utfordrende i dette prosjektet. Utformingen av et brukervennlig grensesnitt var også utfordrende, og jeg måtte teste selve applikasjonen selv flere ganger, før jeg la merke til at jeg kanskje burde inkludere mer validering. Til tross for disse utfordringene, oppnådde jeg et tilfredsstillende resultat med en brukervennlig applikasjon. Implementering av de forskjellige programmeringsprinsippene har resultert til en mer robust kode, og ført til bedre vaner.

7 KONKLUSJON - ERFARING

Gjennom denne prosessen har jeg fått mye erfaring ved å ha anvendt teori i praksis fra emnet Programmering 1, og resultatet har vært å utvikle prosjektet TrainDeparture Application.

Hvis jeg kunne ha startet på nytt, hadde jeg planlagt ansvarsområder og design for hver klasse tidligere. Jeg hadde også lite kunnskap om versjonskontroll med Git og angrer på at jeg ikke tok tid på å lære meg det tidligere. Jeg hadde dårlig uttenkte kommentarer og det tok tid før jeg brukte det på en god måte.

Videre arbeid med denne applikasjonen kan inkludere støtten for at det kan være flere enn en fast avreisestasjon, og at togavgangene ikke bare strekker seg innenfor en enkelt dag, men at man kan endre dato.

Denne prosessen har vært lærerik, spesielt med tanke på struktur av klasser og roller. Bakgrunns teori om designprinsipper og robusthet har vært essensiell og det er tydelig at man må ta i bruk det meste av det i praksis. Det har vært utfordrende å implementere et brukervennlig grensesnitt, men gjennom JUnit testing har jeg fått et tilfredsstillende resultat. Å benytte AI - verktøy som CoPilot og ChatGPT har vært litt nyttig, men det har også vært viktig å være kritisk til resultatene.

Samlet sett har erfaringen gjennom dette prosjektet gitt meg et bedre perspektiv på objektorientert programmering i Java, og jeg vil fortsette å forbedre mine ferdigheter i fremtidige prosjekter.

8 REFERANSER

[1] Barnes, David J & Kölling Michael. (2017). *Objects First With Java, Sixth edition.* Pearson Education Limited, brukt som kilde gjennom hele kapittel 3 – Teoretisk grunnlag

[2] Oreilly. (2023). *Validate Traditional Time Formats.* Oreilly. Hentet 09.12.2023 kl: 21:37 fra: https://www.oreilly.com/library/view/regular-expressions-cookbook/9781449327453/ch04s06.html, kapittel 6 - bruk av regex format i : 'readLocalTimeInput' – metoden