**DAT109**

**Obligatorisk innlevering 1**

**Gruppemedlemmer:**

Endre Strand Norén

Elias Dalland

Marius Borgen

Kristoffer Albrigtsen

Anastasia Vivmond

Makka Sugaipova



Anastasia Vivmond, Marius Borgen



Endre Strand Noren, Makka Sugaipova



Elias Daland, Kristoffer Albrigtsen

1. **Innledning**

I denne oppgaven skulle vi modellere og programmere brettspillet «Slange og stige». Hensikten med modellene er å gi en oversikt over hvordan spillet fungerer, og hvordan de ulike delene (konseptene/attributtene) henger sammen og samhandler. I programmeringsdelen er målet å implementere spillet slik at 2–4 spillere kan spille mot hverandre. Vi har også laget tester som verifiserer at programmet fungerer som forventet.

Spillet går ut på å bevege seg gjennom et brett med 100 ruter (1–100). Underveis kan en spiller lande på ruter med stiger eller slanger. Stiger flytter spilleren fremover til en høyere rute, mens slanger flytter spilleren bakover til en lavere rute. Spillerne kaster terningen etter tur og flytter brikken fremover med antall øyne som trilles. Dersom en spiller triller en 6, får spilleren et ekstra kast. Triller en spiller 6 tre ganger på rad, må spilleren gå tilbake til start og deretter trille en 6 for å kunne fortsette videre.

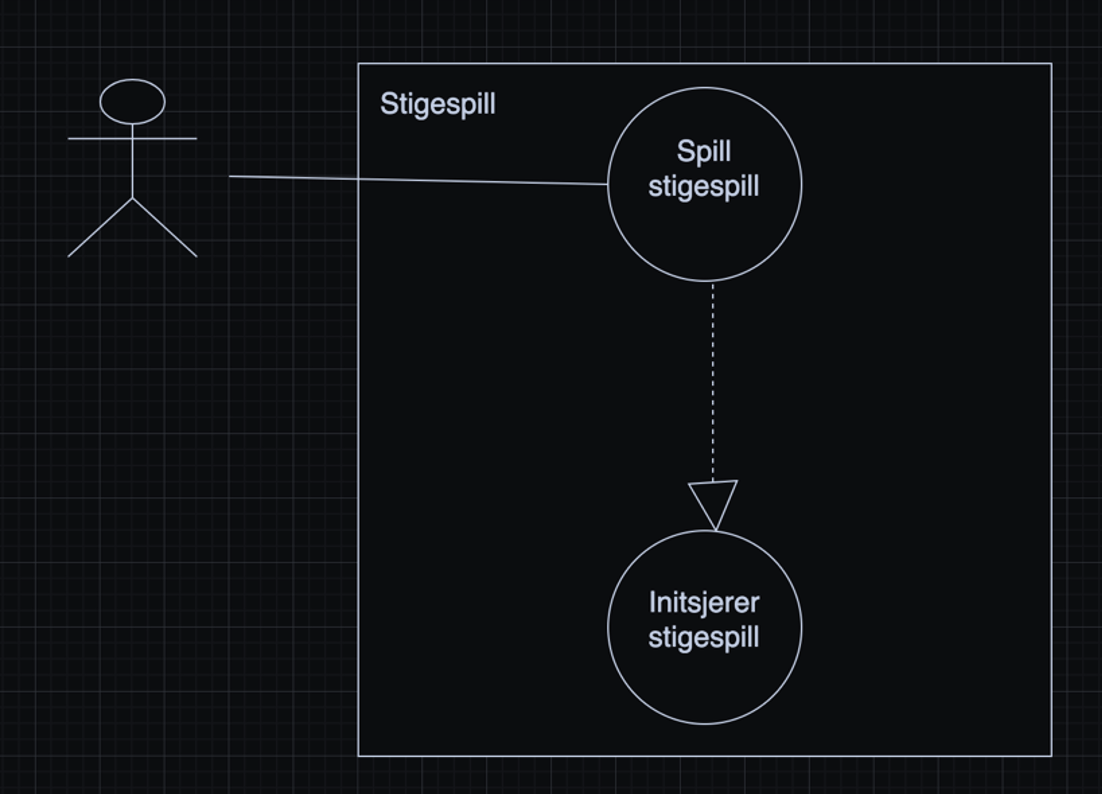
Løsningen har utviklet seg iterativt gjennom to hovedfaser, noe som gjenspeiles i mappestrukturen i innleveringen:

1. **Java-stigespill**: Dette er den første prototypen vi utviklet. Denne inneholder grunnleggende spillmekanikk, men mangler database og avansert arkitektur.
2. **Spring\_Stigespill**: Dette er den endelige, ferdige versjonen og hovedleveransen vår. Denne applikasjonen er bygget på Spring Boot, oppfyller alle krav til MVC-arkitektur, databaseintegrasjon og enhetstesting.

Det er **Spring\_Stigespill** som skal kjøres for å teste den fullverdige løsningen.

1. **Kravanalyse og Modellering**

**Brukstilfellediagram**



Brukstilfellediagrammet viser samhandlingen mellom en bruker og systemet Stigespill. Figuren med strekpersonen representerer brukeren (spilleren) som benytter systemet. Når brukeren starter spillet, utføres brukstilfellet «Spill stigespill», som innebærer at systemet initierer stigespillet og setter spillet i gang.

**Aktivitetsdiagram**

****

Dette aktivitetsdiagrammet viser hvordan en spillrunde fungerer, og hvilke handlinger en spiller kan utføre i løpet av sin tur. Spilleren starter runden med å trille terningen og flytter brikken rute for rute fremover tilsvarende antall øyne som vises. Under flyttingen, hvis spilleren når rute 100, blir den stående.. Etter at spilleren har flyttet, sjekkes det hvilken type rute spilleren har landet på. Dersom spilleren lander på en stige, flyttes spilleren videre fremover. Landet spilleren på en slange, flyttes spilleren bakover, mens en tom rute ikke medfører noen ekstra bevegelse. Hvis spilleren etter alle flyttinger ender på rute 100, har spilleren vunnet spillet. Når disse handlingene er utført, avsluttes spillerens runde, og turen går videre til neste spiller.

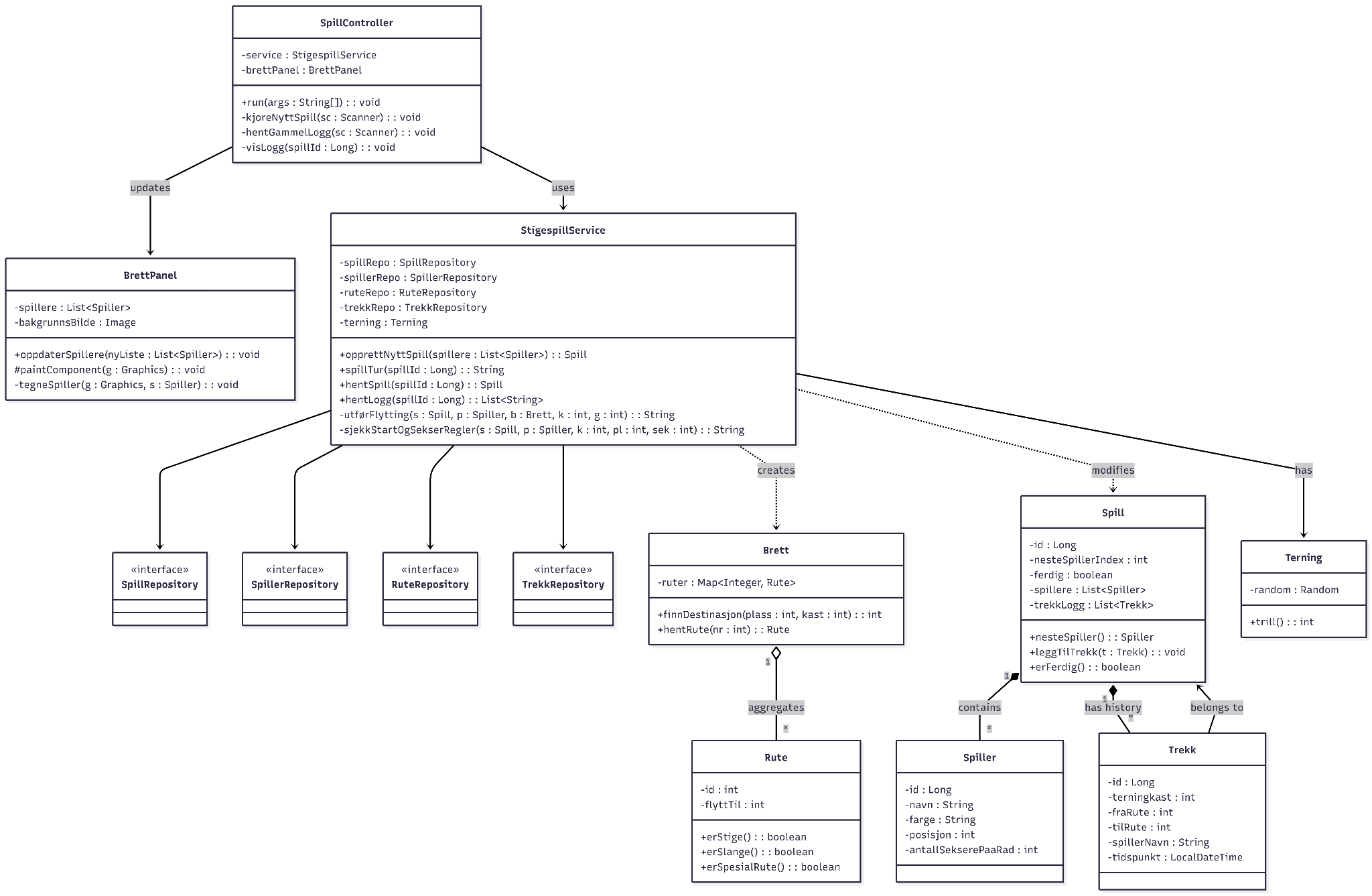
I den implementerte versjonen av spillet er det implementert regler om ekstra kast ved å trille en 6, og regelen om at dersom en spiller triller 6 tre ganger på rad, må spilleren gå tilbake til start. Disse reglene kunne vært inkludert i en utvidet versjon av aktivitetsdiagrammet for å gi en mer komplett beskrivelse av alle mulige spillscenarier.

1. **Design og Arkitektur**

**Domenemodell I form av klassediagram**

Under følger klassediagrammet for vår endelige løsning.

[Hele diagrammet kan også sees her.](https://github.com/EndreNoren/StigeSpill/blob/d102d87ff32120e01f1734940a7e00a1f9c98378/diagrammer/Klassediagram%20Stigespill.png)



Diagrammet viser hvordan applikasjonen er strukturert etter en lagdelt arkitektur, hvor ansvarsområdene er tydelig fordelt mellom Controller, Service, Repository og Model. Vi har valgt en lagdelt struktur (MVC) for å skille tydelig mellom brukergrensesnitt, logikk og datalagring. Teksten på pilene i diagrammet beskriver hvordan komponentene samhandler:

**Controller** (SpillController): Denne klassen styrer spillets gang.

Relasjonen "uses" viser at kontrolleren er avhengig av StigespillService for å utføre logikken, og "updates" viser at den gir beskjed til BrettPanel når grafikken skal tegnes på nytt.

**Service** (StigespillService): Her ligger forretningslogikken.

"Creates": Servicelaget oppretter et midlertidig Brett-objekt for å beregne trekkene.

"Modifies": Servicen endrer tilstanden i Spill-objektet (f.eks. oppdaterer spillerposisjoner).

"Has": Servicen eier en instans av Terning for å generere tilfeldige kast.

**Model** (Spill, Spiller, Trekk): Entitetene som representerer dataene.

"Contains": Pilen med en fylt diamant viser komposisjon; et Spill består av flere Spiller-objekter.

"History": Relasjonen mellom Spill og Trekk er markert som history, da Trekk-objektene fungerer som en komplett logg over alt som har skjedd i spillet.

"Belongs to": Hvert enkelt Trekk vet hvilket Spill det tilhører, noe som er nødvendig for lagring i databasen.

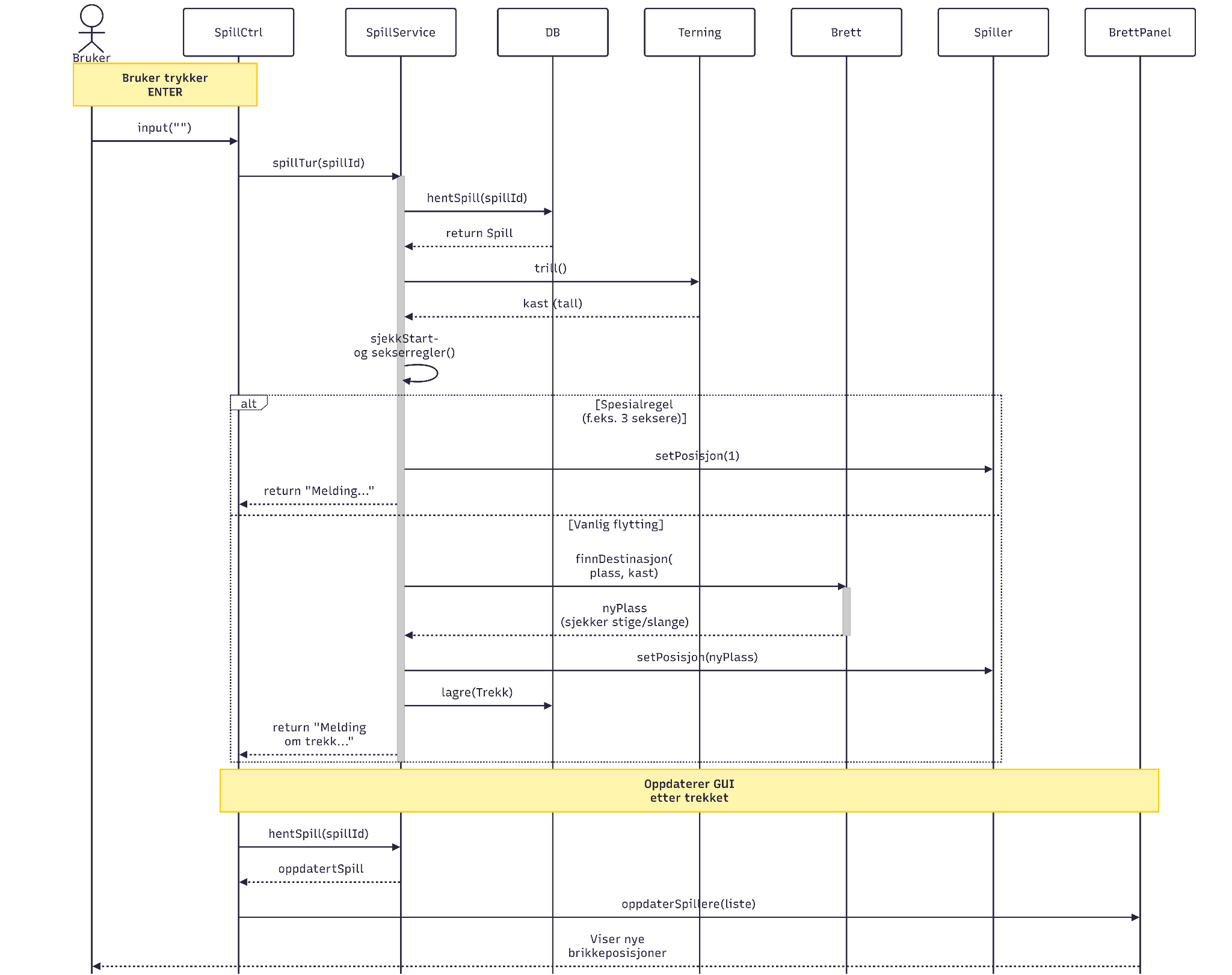
**Brett og Rute:**

"Aggregates": Brett består av (◇) mange Rute-objekter. Dette er en aggregering fordi rutene hentes fra databasen og settes sammen til et brett ved behov.

Denne strukturen sikrer lav kobling mellom klassene. For eksempel vet ikke Spill-objektet hvordan det skal lagres i databasen (det er Repository sitt ansvar) eller hvordan det ser ut på skjermen (det er View sitt ansvar).

**Sekvensdiagram**

[**Hele diagrammet kan også sees her.**](https://github.com/EndreNoren/StigeSpill/blob/d102d87ff32120e01f1734940a7e00a1f9c98378/diagrammer/Sekvensdiagram%20Stigespill.png)



Sekvensdiagrammet illustrerer interaksjonen mellom systemets komponenter når en spiller utfører et trekk. Diagrammet viser hvordan forespørselen håndteres gjennom lagene i arkitekturen, fra brukerinput til databaselagring og oppdatering av visningen.

**Flyten i diagrammet:**

1. **Initiering:** Prosessen starter når bruker trykker ENTER. Dette fanges opp av SpillController som kaller metoden spillTur(spillId) i servicelaget.
2. **Henting av tilstand:** StigespillService henter nåværende spilltilstand fra databasen via Repository-laget (hentSpill).
3. **Logikk og Regler:** Service-laget ber Terning-objektet om et kast. Deretter utføres en sjekk av regler. Diagrammet bruker en alt-boks (alternative execution) for å vise to mulige utfall:

**Spesialregel:** Hvis spilleren f.eks. får tre seksere på rad, settes posisjonen til 1 (start), og turen avsluttes.

**Vanlig flytting:** Service-laget bruker Brett-objektet til å beregne ny posisjon (finnDestinasjon). Her sjekkes det automatisk om spilleren lander på en stige eller slange.

1. **Persistering:** Før svaret sendes tilbake, lagrer Service-laget det nye trekket i databasen (lagre(Trekk)). Dette sikrer at historikken er bevart selv om programmet skulle kræsje.
2. **Oppdatering av Visning (GUI):** Når logikken er ferdig, henter SpillController det oppdaterte spillet på nytt og sender den nye listen med spillere til BrettPanel (oppdaterSpillere). Panelet tegner da opp brikkene på de nye posisjonene for brukeren.

Denne sekvensen bekrefter at vi følger MVC-prinsippet, hvor kontrolleren kun delegerer arbeid, servicelaget tar tunge beslutninger, og visningen (View) kun oppdateres etter at tilstanden er trygt lagret i databasen.

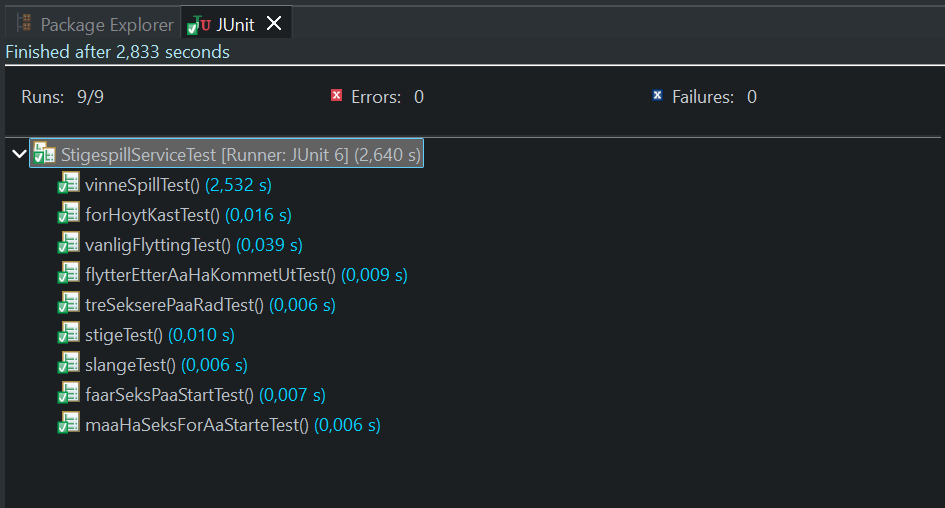
**4. Testing**

For å sikre at spillreglene overholdes, har vi utviklet enhetstester ved hjelp av JUnit 5. Testklassen StigespillServiceTest dekker kritisk funksjonalitet i systemet.

Vi har testet følgende scenarioer:

* Vanlig flytting: At brikken flyttes riktig antall steg.
* Stiger og Slanger: At brikken automatisk flyttes opp eller ned hvis man lander på spesialruter.
* Målstrek: At man ikke vinner hvis man kaster for høyt (brikken blir stående).
* Sekser-regel: At telleren for antall seksere fungerer, og at 3 seksere sender spilleren til start.

Alle tester kjører "grønt", noe som bekrefter at logikken i StigespillService er robust.

**

**5. Begrunnelse for utformingsvalg (GRASP)**

For å sikre oss at spillet følger objektorienterte prinsipper, har gruppen valgt å fordele klassene på følgende måte:

1. **Informasjonsekspert:**  
   Dette prinsippet fordeler ansvaret for informasjonen fordelt på de forskjellige klassene.

**Brett:** Er eksperten på brettets layout. Vi har plassert metoden finnDestinasjon(plass, kast) her fordi Brett inneholder oversikten over alle ruteobjekter og vet hvilke ruter som er vanlige eller har slanger og stiger.

**Rute:** Er ekspert på sitt eget felt. Som nevnt har ruteobjektet kontroll på hvilke ruter som inneholder felt med en tom, slange eller stige verdi.

**Spill:** Er ekspert på spilletilstand. Har kontroll på nesteSpillerIndex og om spillet er ferdig.

1. **Skaper:**

Skaper mønstret hjelper oss med å avgjøre hvilke klasser som har ansvar for å opprette objekter.

**StigespillService:** Skaper for spill objektet via metoden opprettNyttSpill(spillere). Denne er best egnet for denne oppgaven siden den har tilgang til nødvendige repositories og koordinerer oppstart av spillet.

**Spill:** Har ansvar for å opprette trekkobjekter som skal lagres og logges i trekklogg.

1. **Kontroller**

Kontrolleren er det første objektet som tar imot systemhendelser fra brukergrensesnitt.

**SpillController:** Tar imot input fra spillerne, f.eks. ved trykk av enter eller nytt spill. Som vil delegere logikken til StigeSpillService. Dette vil skille brukergrensesnittet fra spilllogikken som vil hjelpe med å vedlikeholde koden.

1. **Lav kobling**

Målet med lav kobling er å gjøre klassene minst mulig avhengig av hverandre. Slik at endringer ikke vil ødelegge for hverandre.

**Repositories:** Gjennom bruk av spillRepository og spillerRepository trenger ikke spilllogikken vite detaljer om hvordan databasen fungerer. Logikken er koblet til et grensesnitt i stedet for spesifikke datateknologi.

**Lagdeling:** Ved å ha et tydelig skille mellom spillController, StigespillService og domenemodellene (Spill, Brett, Spiller) oppnår vi et fleksibelt system.

1. **Høy sammenheng**

Dette prinsippet handler om at hver klasse skal ha sitt smale og godt definerte ansvarsområde.

**Fokuserte klasser:** Terning (kun ansvar for å generere tilfeldig tall), Spiller (kun ansvar å holde informasjon om seg selv som navn, farge og posisjon) og brettPanel (kun tegne brettet på skjermen).

Ved å unngå Gud klasser blir koden enklere å forstå og lettere og raskere å enhetsteste.

1. **Konklusjon**

Gjennom denne obligatoriske øvelsen har vi oppnådd hovedformålet om å utvikle et program fra idé til et ferdig, kjørbart produkt. Arbeidet med UML-modellering i forkant ga oss en dypere forståelse av kravspesifikasjonen, noe som gjorde selve implementasjonen mer strukturert og effektiv.

Vi har levert en fungerende Java-applikasjon som oppfyller alle de funksjonelle kravene, inkludert spillregler for stiger, slanger, sekser-kast og målgang. I henhold til de tekniske kravene har vi:

* Dokumentert koden med JavaDoc.
* Verifisert logikken gjennom enhetstester i JUnit.
* Implementert persistering ved bruk av Spring Data JPA, slik at alle trekk og spill lagres i en database.
* Laget funksjonalitet for å hente ut og vise hele spillhistorikken i etterkant.

Ved å benytte en lagdelt arkitektur (Spring Boot MVC) har vi sikret at koden er ryddig, testbar og i tråd med god programmeringsskikk. Prosjektet har gitt oss verdifull erfaring i å koble sammen objektorientert design med moderne rammeverk.