DAT108

Består av Kristoffer, Simen, Herborg og Fredrik

Oppgave 1

1.1    Controller

    Et brukstilfelle-mønster brukes til å håndtere alle systemhendelser i et brukstilfelle. I dette eksempelet et Stigespill.

    Kontroll-mønsteret tildeler ansvaret til ulike kontrollobjekter (klasser) i et overordnet system og bruksscenarioer. Dette kontrollobjektet har et ansvar for mottak eller håndtering av en eller flere systemhendelser. I dette spillet er det en spiller som starter med å delegere arbeidet slik at systemet går rundt. Kontrolleren gjør ikke mye arbeidet selv men setter i gang en hendelse som får de andre kontrollobjektene til å utføre sine systemhendelser.

1.2    Creator

* Hva er det som skaper selve spillet?

    En *spiller* i *stigespillet*, spiller med en *kopp* som inneholder *terninger.* Stigespillet spilles på et *brett* som inneholder ruter. Denne spilleren eier en *brikke* i *spillet.* Brikken er på *rutene* på *brettet*. Hvor brikken havner er basert på terninger, *slanger* og *stiger*.  Hver av disse objektklassene har sine roller og utgjør sammen et system som lager et spill.

* Stigespillet konstruerer en kopp og et brett.
* Et brett inneholder ruter med slanger og stiger på.

1.3    High cohesion

Sammenkobling av klasser. High Cohesion betyr at tilhørighet mellom klasser skal vere “robustness, reliability, reusability, and understandability”. Low cohesion er når desse punkta ikkje er oppnådd. For å oppnå high cohesion skal klassene tilhøre kvarandre med begrensninger: ein klasse skal ikkje høyre til alt for mange andre klasser.

Domenemodellen består av mange klasser for å fordele ansvar på mange klasser. Da har klassene til stigespill få og konkrete koblinger. Derfor er modellen lett å forstå, lett å gjenbruke og pålitelig. Eksempel på dette er at *kopp* har ansvar på alle terningene i spelet framfor at *stigespill* har det.

Klassen *spiller* bruker metoder som er direkte koblet til kopp, brett og brikke. *Spiller* har mange metoder koblet til disse klassene og er veldig sentral i spillet.

1.4    Indirection

Indirection går ut på å frakoble objekter fra hverandre. Dette gir mulighet for low coupling og gjør at potensialet for gjenbruk av objekter er høy. Dette gjøres i første omgang ved å se på modellen vi jobber med og lokalisere klassene med low coupling eller direkte coupling. Vi skal da bruke en eksisterende klasse til å ta ansvaret for funksjonaliteten som skaper low coupling, og funksjonaliteten som har høyt potensial for gjenbruk. Bruk av indirection-mønster vil resultere i god vedlikeholdbarhet, robuste og gjenbrukbare systemer.

1.5    Information expert

Kopp er informasjonsekspert for terning. terning har ansvar for å finne verdi mellom 1-6, og kopp skal ta vare på verdien.. Spiller ber kopp om å trille terning, og henter summen med metoden getSum(). Spiller sjekker selv om summen tilsier sekser.

Spiller henter hvilken rute brikken er på ved å kalle på metoden getPlass() i klassen brikke. Spiller bestemmer ny plass ved å kalle på setPlass(nyPlass). nyPlass finner spilleren ved å kalle på metoden finnRute(plass, sum) i brett. plass finner spilleren ved å kalle på getPlass() i brikke.

1.6    Low coupling

“Low coupling” handler om i hvor stor grad forskjellige klasser er avhengige av hverandre. Jo flere forskjellige klasser en klasse er avhengig av, jo vanskeligere blir det å vedlikeholde koden og endre den i ettertid. Grunnen til dette er at ved å endre en klasse får det ringvirkninger for alle andre klasser som er koblet sammen med denne. I tillegg er det vanskeligere å gjenbruke kode hvis den i stor grad er avhengige av andre klasser.

I domenemodellen kan vi se at koblingsgraden er veldig liten. Det er bare klassen “Stigespill” som har mer enn to koblinger, og grunnen til dette er at den har to ansvarsområder. Den har ansvar for å opprette selve spillet og kontrollere gangen i spillet. Vi kunne da argumentert for å heller lage en kontroller-klasse som har ansvar for selve kontrollen i spillet, mens “Stigespill” da bare har ansvar for å opprette spillet.

1.7    Polymorphism

polymorfisme gjør så Objekter i et program kan ha varierende funksjonalitet av forskjellige metoder ettersom hvilke subtype den er av en superklasse. I Java vil alltid kompilatoren sjekke om et objekt har en underklasse hvor et metodekall har en annen funksjonalitet enn superklassen. Ved å gi forskjellige Objekter mulighet til å klassifiseres som underobjekter av en annen klasse kan man utføre polymorfisme og få en enklere struktur i koden, så man slipper å gi alle underklassene egne metoder for metodekall som kan utføres på superklassen.

For å ta et eksempel på polymorfisme i oppgaven; Det man kan se på her er objektet rute. objektet rute er en overklasse til de to underklassene slange og stige. her sier man at slange ER en rute og stige ER en rute. Begge disse klassene vil ha en metode som flytter spilleren fra den ruten de er på til en annen. Polymorfisme gjør så denne metoden kan ha samme navn og bli kalt på en rute, men ha forskjellig utfall. metoden kan kalles for flytt(). Flytt() metoden vil være i både Slange klassen og i Stige klassen så hvis metoden flytt blir kalt på en rute vil kompilatoren kjøre denne på Rute objektet, men sjekker alltid først om dette Rute objektet har en underklasse med metoden flytt() og bruker denne i stedet. så hvis det er en slange, vil Rute. flytt() gjøre så spilleren faller tilbake et visst antall ruter og hvis det er en stige vil spilleren klatre opp og frem i spille.

1.8    Protected variations   
1.9    Pure fabrication

Kan GRASP brukes til å forbedre diagrammet når vi skal lage et klassediagram for løsningen?