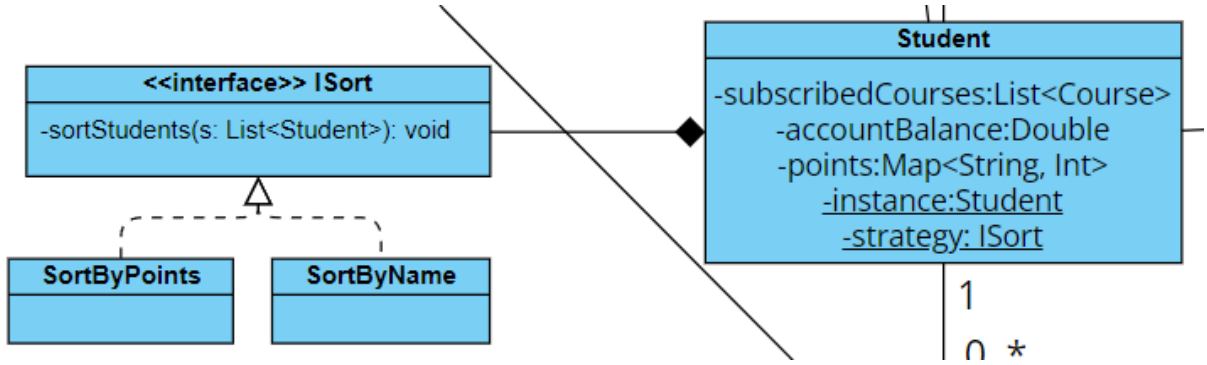
Patterni ponašanja

Strategy pattern

Strategy pattern je design pattern koji omogućava objektu da promijeni svoje ponašanje dinamički tokom izvršavanja. Ovaj pattern se koristi kada želimo da objekt koristi različite strategije ili algoritme za izvršavanje određene operacije, pri čemu se odabir strategije može mijenjati u pokretanju programa.

U našem slučaju, primijenili smo Strategy pattern na klasu Student. Prvo smo definisali interface ISort koji predstavlja strategiju sortiranja, koji sadrži metodu sortStudents().

Zatim smo implementirali dvije klase koje implementiraju interface ISort: SortByPoints i SortByName. Svaka od ovih klasa će sadržavati vlastitu implementaciju metode sortStudents() koja će sortirati listu studenata prema odgovarajućem kriteriju. Nakon toga, u klasi Student smo dodali atribut tipa ISort koje će predstavljati strategiju sortiranja.



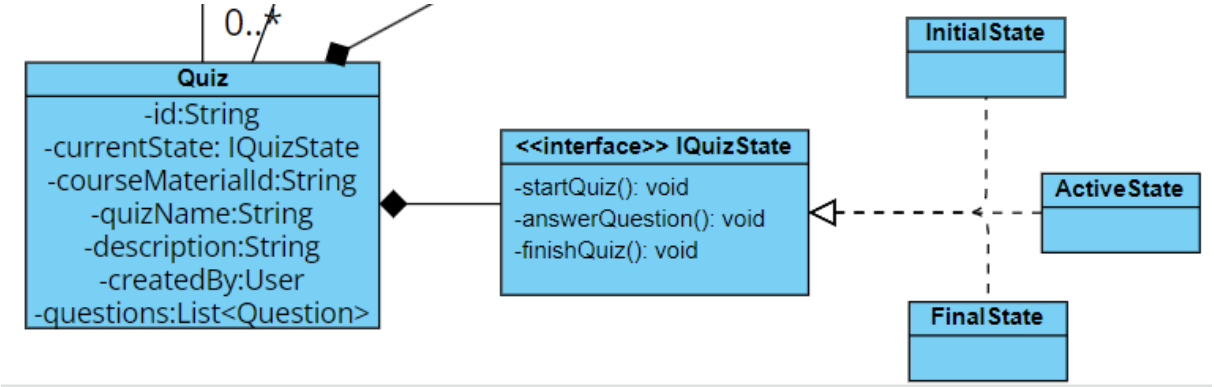
State pattern

State pattern je design pattern koji omogućava objektu da promijeni svoje ponašanje ovisno o trenutnom stanju. U slučaju klase Quiz, možemo primijeniti State Pattern kako bismo upravljali različitim stanjima kviza.

Na primjer, možemo definisati različita stanja kviza kao što su "početno", "aktivno" i "završeno". Svako stanje će imati svoje specifično ponašanje i moguće akcije koje se mogu izvršiti u tom stanju.

Prvo smo definisali interface IQuizState koji sadrži metode startQuiz(), answerQuestion(), finishQuiz(). Zatim smo implementirali tri klase koje predstavljaju određeno stanje kviza, i

koje implementiraju interface IQuizState: InitialState, ActiveState i FinalState. Svaka od ovih klasa će sadržavati vlastite implementacije metoda startQuiz(), answerQuestion() i finishQuiz(). Na kraju moramo dodati atribut currentState u klasu kviz koje je tipa IQuizState, koji predstavlja trenutno stanje kviza.



Template method pattern

Template method pattern se može primijeniti u situacijama kada imamo algoritam ili proces koji se sastoji od zajedničkih koraka, ali se neki od tih koraka razlikuju ovisno o konkretnoj implementaciji.

U našem projektu, primjer primjene Template method patterna već je iskorišten u klasi Courses, pošto imamo dvije različite vrste kurseva (besplatni i plaćeni kursevi), i postoji zajednički proces registracije za sve vrste kurseva, ali se u plaćanju razlikuju. Apstraktna klasa Courses definiše zajedničke korake registracije kursa, ali ostavlja specifične korake kao apstraktne metode koje će biti implementirane u podklasama za plaćene i besplatne kurseve.

Observer pattern

Observer pattern je design pattern koji omogućava komunikaciju između objekata u situacijama kada jedan objekt treba obavijestiti druge objekte o promjeni svog stanja. Ovaj obrazac omogućava uspostavljanje jednosmjernih veza između objekata, gdje promjena u jednom objektu automatski obavještava druge objekte koji su vezani za tu promjenu.

U našem projektu, primjena Observer patterna može biti na primjeru praćenja napretka studenata u kursu. Prvo možemo definisati interface Observer koji će sadržavati metodu update() koja će biti pozvana kada se promjene dogode u napretku studenata. Zatim u klasi CourseProgress možemo dodati atribut koji će predstavljati listu objekata (List<Observer>) koji implementiraju interface Observer. U klasi Student možemo dodati metode setProgress() i getProgress() koje će ažurirati i dohvaćati napredak studenta.

Iterator pattern

Iterator pattern se može primijeniti u našem projektu na primjeru klasa koje sadrže kolekcije podataka, npr. klase CourseMaterial koja sadrži listu kvizova.

Prvo bismo trebali definisati interface IIterator koji sadrži metode za iteriranje kroz listu kvizova, poput hasNext() (provjera postojanja sljedećeg kviza) i next() (dohvaćanje sljedećeg kviza). U klasi CourseMaterial trebamo implementirati metodu getIterator() koja vraća instancu objekta koji implementira interface IIterator. Ovaj objekt će omogućiti iteriranje kroz listu kvizova. Zatim u klasi koja implementira interface IIterator, trebamo implementirati metode hasNext() i next() za provjeru postojanja sljedećeg kviza i dohvaćanje sljedećeg kviza u listi.

Chain of responsibility pattern

Chain of responsibility pattern možemo primijeniti na obradi plaćanja u našem projektu. Na primjer, kada student želi izvršiti plaćanje kursa, možemo napraviti niz handlera koji provjeravaju različite aspekte plaćanja, kao što su provjera valjanosti kartice, dostupnost sredstava, autentifikacija itd.

Svaki handler u lancu ima odgovornost za provjeru jednog aspekta plaćanja. Ako handler može obraditi zahtjev, obrađuje ga i vraća rezultat. Ako handler ne može obraditi zahtjev, preusmjerava ga na sljedeći handler u lancu.

Na primjer, prvi handler u lancu provjerava valjanost kartice. Ako kartica nije valjana, preusmjerava zahtjev na sljedeći handler u lancu, koji može provjeriti dostupnost sredstava na kartici. Ako svi handleri u lancu ne mogu obraditi zahtjev, korisniku se prikazuje odgovarajuća poruka o neuspješnom plaćanju.

Mediator pattern

Mediator pattern možemo primijeniti na komunikaciju između različitih komponenti za Billing, kao što su BillingCard i Email. Mediator bi bio objekt koji djeluje kao posrednik između tih komponenti i omogućava im razmjenu informacija i suradnju.

Na primjer, kada se naplata (Billing) obavi uspješno, Mediator može poslati obavijest putem Email komponente studentu o uspješnoj naplati. Mediator bi imao metode poput sendEmail() za slanje emaila studentu.

Primjenom Mediator patterna na ovaj način, omogućava se centralizirana komunikacija i suradnja između različitih komponenti za fakturiranje, također omogućava fleksibilnost u dodavanju novih komponenti, bez uticaja na ostatak koda.