# STRUKTURALNI PATTERNI

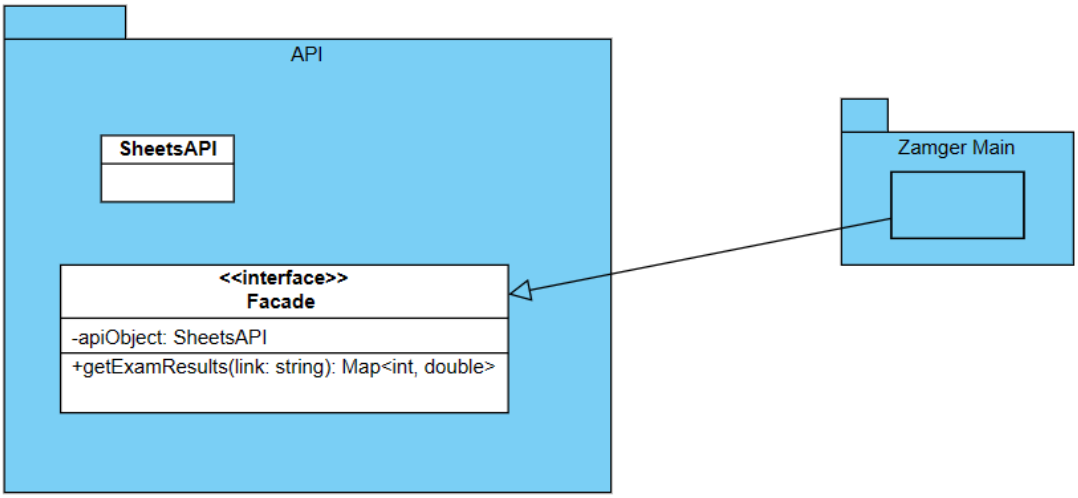
## Adapter

Ovaj design pattern rješava problem nekompatibilnih interfejsa dvaju klasa. U našem sistemu, očekujemo primjenu ovog pattern-a pri implementaciji funkcionalnosti koja koristi Google Docs API za dohvaćanje rezultata ispita sa datog linka. Pretpostavka je da metode klasa koje će za nas dohvatiti ove podatke nisu kompatibilne sa našim dizajniranim klasama, te će korištenje adaptera riješiti ovaj izazov.

## Façade

Ovaj design pattern odvaja implementaciju kompleksnog podsistema od ostatka sistema s kojim komunicira preko interfejsa. Ovo već imamo sa klasom *ApplicationDbContext* koja komunicira s bazom, a fasadu ćemo napraviti i prema web api koji ćemo koristiti. Poziv web servisa I dohvaćanje podataka ćemo odvojiti od ostatka sistema u posebnu klasu, koja će imati interfejs *Façade* za komunikaciju s ostatkom sistema. Ukoliko bude potrebno adaptiranje metoda fasade sa nekom od klasa iz ostatka Sistema, koristit će se adapter.

Mogući dizajn:

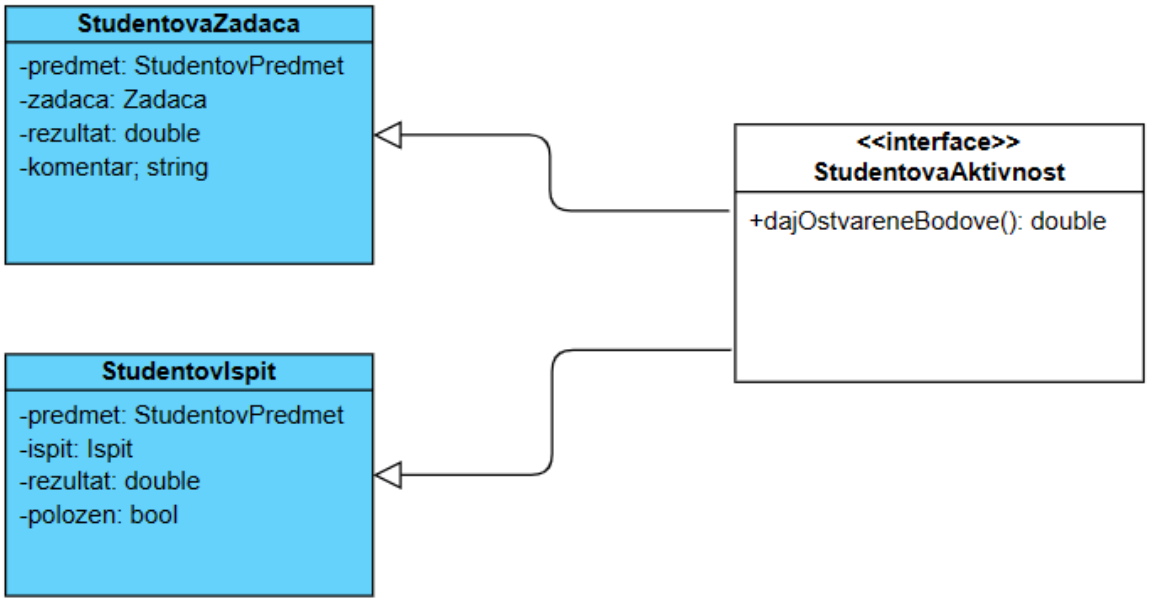


## Decorator

Decorator dinamički dodjeljuje svojstva i funkcionalnosti objektu. S obzirom da naš projekat ima već sasvim dovoljno funkcionalnosti, nećemo ga korstiti, ali ideja bi bila najprije uvođenje interfejsa *Aktivnost* koji bi implementirali klase *Ispit* i *Zadaca*, tj. Aktivnosti studenta na Kojima se ostvaruju bodovi. Dalje bismo *Zadacu* mogli razraditi da obuhvata i studentske projekte, seminarske radove, i svaki drugi vid aktivnosti koja se može bodovati, a nije *Ispit* (hipotetički govoreći, tada ni naziv klase *Zadaca* možda nije najadekvatniji). "Razrada" *Zadace* podrazumijeva njeno pretvaranje u interfejs (dakle, *Zadaca* bi predstavljala *Decorator*), a konkretne aktivnosti bi bile konkretne implementacije dekoratora (npr. *Projekat*).

## Composite

Dekorator nećemo koristiti jer nema potrebe za velikim proširenjem studentskih aktivnosti, a za planirane funkcionalnosti su dovoljne već dizajnirane klase. Međutim, javlja se potreba za sastavljanjem objekta koji će predstaviti sve ostvarene bodove studenta na predmetu, a za to nam trebaju dvije klase *StudentovIspit* i *StudentovaZadaca*. Zato ćemo uvesti interfejs *Aktivnost* koji će implementirati obe prethodno spomenute klase, te ćemo moći napraviti List<*Aktivnost*> sa svim ostvarenim bodovima studenta. Ovo zaista predstavlja composite pattern, ali trivijalnu varijantu bez kompozitnog objekta, tj. obje klase koje će implementirati *Aktivnost* će biti listovi u strukturi stable.



**Bridge**

Bridge se korisiti za odvajanje apstrakcije i implementacije klase. Bridge nećemo koristiti jer naše klase nemaju mnogo djece sa različitim implementacijama metoda. Moglo bi se to npr. učiniti da imamo više različitih implementacija notificiranja (npr. da postoji notifikacija slanjem e-maila). Tada bismo imali tri djeteta klase *Osoba* i više implementacija notificiranja, te bi se između ove dvije grupe klasa mogao uspostaviti bridge.

**Proxy**

Proxy je koristan za zaštitu pristupa objektima. U našem sistemu očigledno je da će biti potrebna autentifikacija, čak je i jedna od ključnih stvari u sistemu, te bi proxy mogao biti koristan. Odluka o tome da li će se koristiti proxy i u kojoj mjeri će se donijeti nakon što sagledamo koji dio autentifikacije će morati biti implementiran, pošto je jedan dio implementiran kreiranjem projekta u VS.

**Flyweight**

Flyweight koristimo kada imamo veliki broj objekata sa sličnim osobinama i stanjima. U našem sistemu će očigledno biti mnogo objekata sa sličnim svojstvima (dobar primjer su klase *StudentovIspit* i *StudentovaZadaca*). Uzmimo za primjer klasu *Zahtjev*. Možemo imati veliki broj zahtjeva jednog studenta za potvrdu o studiranju, tako da svi ovi zahtjevi imaju istu većinu atributa, i prema tome, pogodni su za flyweight.