Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni diplomski studij matematike i računarstva

Seminarski rad

Arhitektura softvera

Softversko inženjerstvo

Davor Kolarević

UVOD

Jedna od ključnih stvari prije samog procesa kodiranja softvera je definiranje arhitekture softvera, iako nam je ponekad previše uobičajeno da samo počnemo kodirati softver bez da smo uopće razmišljali o njegovoj strukturi. Nedostatak softverske arhitekture može otežati razvoj i onih manjih softvera. Takav softver je usko povezan među komponentama i jako ga je teško kasnije mijenjati. Bitno je također odabrati i pravu arhitekturu s obzirom na vrstu softvera koji radimo. Ona nam olakšava kodiranje, pogotovo novim razvojnim programerima koji se tek uključuju u projekt ili onima koji rade na održavanju i nadogradnji.

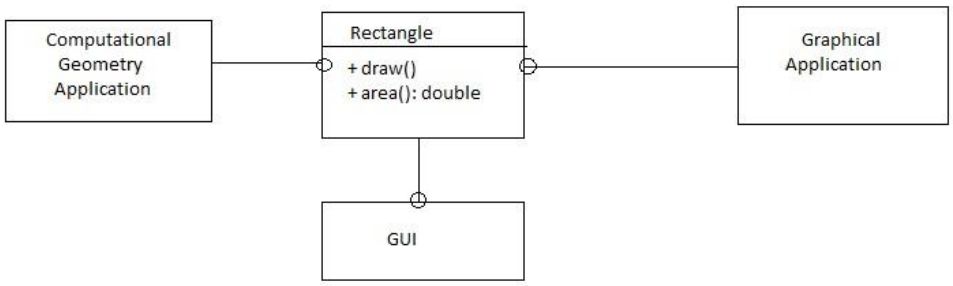
Temelj svake arhitekture, točnije rečeno svakog softvera bi trebali biti S.O.L.I.D. principi, tj. skup od pet principa kojih ukoliko ih se držimo olakšavaju i razvoj i održavanje. U ovom radu prvo obrađujemo slojevitu arhitekturu, arhitekturu koja je najčešće korištena među programerima zbog jednostavnosti implementacije i dobrog odabira onda kada ne znamo koju drugu arhitekturu odabrati. Obrađujemo i arhitekturu navođenu događajima koja u današnje vrijeme puno dinamičkih uređaja koji imaju potrebu za informacijama u raznima situacijama, čini gotovo savršen odabir. Također, mikroservisna arhitektura, koja se nedavno razvila iz monolitnih softvera, koji su se izvršavali kao jedan proces i servisno orijentirane arhitekture. Jedna od prvih IT tvrtki koja je primijenila ovu arhitekturu i udarila temelje je Amazon koji je svoj softver podijelio u više manjih microservisa od kojih svaki obavlja specifičnu poslovnu logiku. Kao zadnju arhitekturu koju obrađujemo je mikrojezgrena arhitektura koja omogućuje lako mijenjanje softvera po potrebi korisnika preko uredbenih modula.

S.O.L.I.D. principi

S.O.L.I.D. principi su skup pet principa koji se tiču objektno orijentiranog dizajna. Ako ih se pravilno držimo, olakšavaju nam razvoj i održavanje softwarea. Riječ S.O.L.I.D. je kratica za:

1. **S** – Single responsibility principle
2. **O** – Open-closed principle
3. **L** – Liskov substitution principle
4. **I** – Interface segregation principle
5. **D** – Dependency Inversion principle

1. **Single Responsability principle** ukratko kaže da „Klasa treba imati jedan i samo jedan razlog da se mijenja, tj. treba imati samo jednu svrhu“. Drugim riječima, svaka klasa treba biti usko vezana za jednu „funkcionalnost“, jer svaka funkcionalnost je vezana za zahtjeve software koji se mogu mijenjati tijekom vremena. Takve promjene dovode do promjena unutar klase. Ukoliko bi svaka klasa imala više funkcionalnosti, imala bi više razloga za promjenom.



Na gornjoj slici vidimo primjer klase *Rectangle* koja narušava Single responsability princip. Zašto? Klasu *Rectangle* ima dvije odgovornosti: prvi je metoda koje je odgovorno za računanje matematičih svojstava, tj. površine koju koristi *Computational Geometry Application*, a druga je renderiranje na grafičkom korisničom sučelju. Ako bi došlo do promjene u *Graphical Application* zbog kojih bi iz nekog razloga morali promijeniti i *Rectangle* klasu, moglo bi se dogoditi da moramo ponovno buildati, istestirati i iznova deployati aplikaciju *Computational Geometry Application*. Bolji način bi bio da matematičke funkcionalnosti i grafičke funkcionalnosti klase *Rectangle* razdvojimo u dvije klase.

2. **Open-closed principle** kaže da „Softverski entiteti ili objekti trebaju biti otvoreni za nadogradnju, ali zatvoreni za promjene“. Drugim riječima, ako je potrebno, moguće je napraviti nadogradnju bez da se mijenja postojeći kod. Ključ je nasljeđivanju, jednom od glavnih karakteristika objektno orijentiranih jezika. Ako su potrebne promjene, nikada se ne bi trebala mijenjati stara klasa, nego se napraviti nova klasa koja će naslijediti staru klasu.

3. **Liskov substitution principle** kaze „Neka je **q(x)** svojstvo definirano kod objekta **x** tipa **T**. Tada **q(y)** bi trebao biti definiran za objekt **y** tipa **S** gdje je **S** podtip tipa **T**“. Pojašnjeno, promjene koje naslijeđena klasa uvodi na baznoj klasi ne smije dovesti do promjena u implementaciji bazne klase na taj način da može izazvati probleme u kodi koji već koristi baznu klasu. Sam princip se oslanja na nekoliko tvrdnji:

a) Metoda naslijeđene klase koja override-a metodu bazne klase, kao ulazne parametre dozvoljeni su isti tipovi parametara koji prima metodabazne klase.

b) Ako metoda bazne klase vraća objekt tipa **T**, onda override-na metoda u naslijeđenoj klasi može vratiti objekt tipa **T** ili tipa **S** ako je **S** naslijedit iz **T**.

c) Novi tipovi iznimki nisu dozvoljeni unutar naslijeđene klase

4. **Interface segregation principle** kaze „Klijenti ne bi trebali ovisiti o sučeljima koje ne koriste“. Ako klijent koristi klasu koja sadrži sučelja koji klijentu nisu potrebni, ali ih koristi drugi klijent, tada klijent može biti zahvaćen promjenama koje ga se ne tiču. Ključ je u razbijanju sučelja na više manjih koje onda možemo kombinirati po potrebi. Na taj način, svaka klasa koja koristi neko sučelje će dobiti točno ono što joj je potrebno. Pogledajmo primjer.

```
1   public interface IOrder
2   {
3       void Purchase();
4       void ProcessCreditCard();
5   }
6
7   public class OnlineOrder: IOrder
8   {
9       public void Purchase() { /*do purchase*/ }
10
11      public void ProcessCreditCard() {/* process through credit card*/ }
12  }
13
14  public class InpersionOrder: IOrder
15  {
16      public void Purchase() { /* do purchase*/ }
17
18      public void ProcessCreditCard()
19      {
20          //Not required for inperson purchase
21          throw new NotImplementedException();
22      }
23  }
```

Sučelje *IOrder* nam stvara problem zato što metoda *ProcessCreditCard* nam je potrebna samo prilikom online kupnje. Stvar možemo popraviti ukoliko sučelje *IOrder* razbijemo na dva nova sučelja, *IOrder* i *IOnlineOrder*. Tada će klasa *OnlineOrder* implementirati oba sučelja, a klasa *InpersonOrder* samo *IOrder* sučelje.

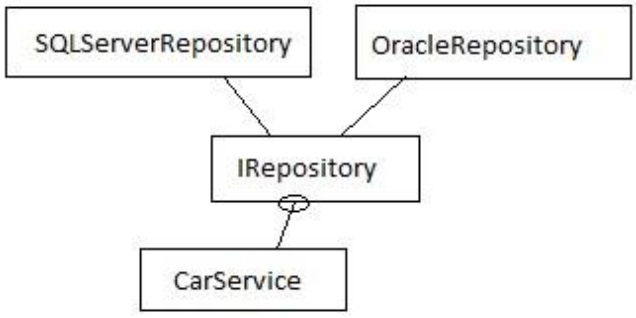
```
26  public interface IOrder
27  {
28      void Purchase();
29  }
30
31  public interface IOnlineOrder
32  {
33      void ProcessCreditCard();
34  }
```

5. **Dependency Inversion Principle** kaže da „Klase moraju ovisi o apstrakcijama, ne o konkretnim implementacijama“. Ako klase ne ovise o konkretnim klasama, nego samo o apstrakcija, onda su lako zamjenjive. Jedan primjer je komunikacija prema bazi.

Pogledajmo primjer. Neka je *IRepository* generičko sučelje koje treba implementirati klase koje su zadužene za komunikaciju sa određenom bazom podataka.

```
1  public interface IRepository
2  {
3      IEnumerable<TModel> getAll<TModel>();
4      TModel getById<TModel>(string id);
5      TModel insert<TModel>(TModel model);
6      void update<TModel>(string id, TModel model);
7      void remove<TModel>(string id);
8  }
```

Na sljedeće dvije sliku su primjeri repozitorija koji su specifični za komunikaciju s SQL Server *SQLServerRepository* bazom podataka i Oracle bazom podataka *OracleRepository*. Servis *CarService* koji je zadužen za operacije entiteta *ICar* prima repozitorij kao dependency preko konstuktor. Primijetimo, da nije eksplicitno navedena konkretna klasa repozitorija nego samo apstrakcija *IRepository*. Na taj način, servis cijelo vrijeme komunikaciju s repozitorijem vrši preko apstrakcije, tj. ne zna s kojom konkretnom klasom komunicira. Na taj način lako možemo zamijeniti konkretne klase bez da mijenjamo naš servis.



```
10  public class SQLServerRepository: IRepository
11  {
12      private string connectionString;
13
14      SQLServerRepository(string connectionString)
15      {
16          //create SQL Server connection
17      }
18
19      IEnumerable<TModel> getAll<TModel>()
20      {
21          // get all models from SQL Server database
22      }
23
24      TModel getById<TModel>(string id)
25      {
26          // get model by id from SQL Server database
27      }
28
29      TModel insert<TModel>(TModel model)
30      {
31          // insert new model in SQL Server database
32      }
33
34      void update<TModel>(string id, TModel model)
35      {
36          // update model in SQL Server database
37      }
38
39      void remove<TModel>(string id)
40      {
41          // remove model in SQL Server database
42      }
43  }
```

```
47  public class OracleRepository: IRepository
48  {
49      private string connectionString;
50
51      ORacleRepository(string connectionString)
52      {
53          //create Oracle database connection
54      }
55
56      IEnumerable<TModel> getAll<TModel>()
57      {
58          // get all models from Oracle database
59      }
60
61      TModel getById<TModel>(string id)
62      {
63          // get model by id from Oracle database
64      }
65
66      TModel insert<TModel>(TModel model)
67      {
68          // insert new model in Oracle database
69      }
70
71      void update<TModel>(string id, TModel model)
72      {
73          // update model in Oracle database
74      }
75
76      void remove<TModel>(string id)
77      {
78          // remove model in Oracle database
79      }
80  }
```

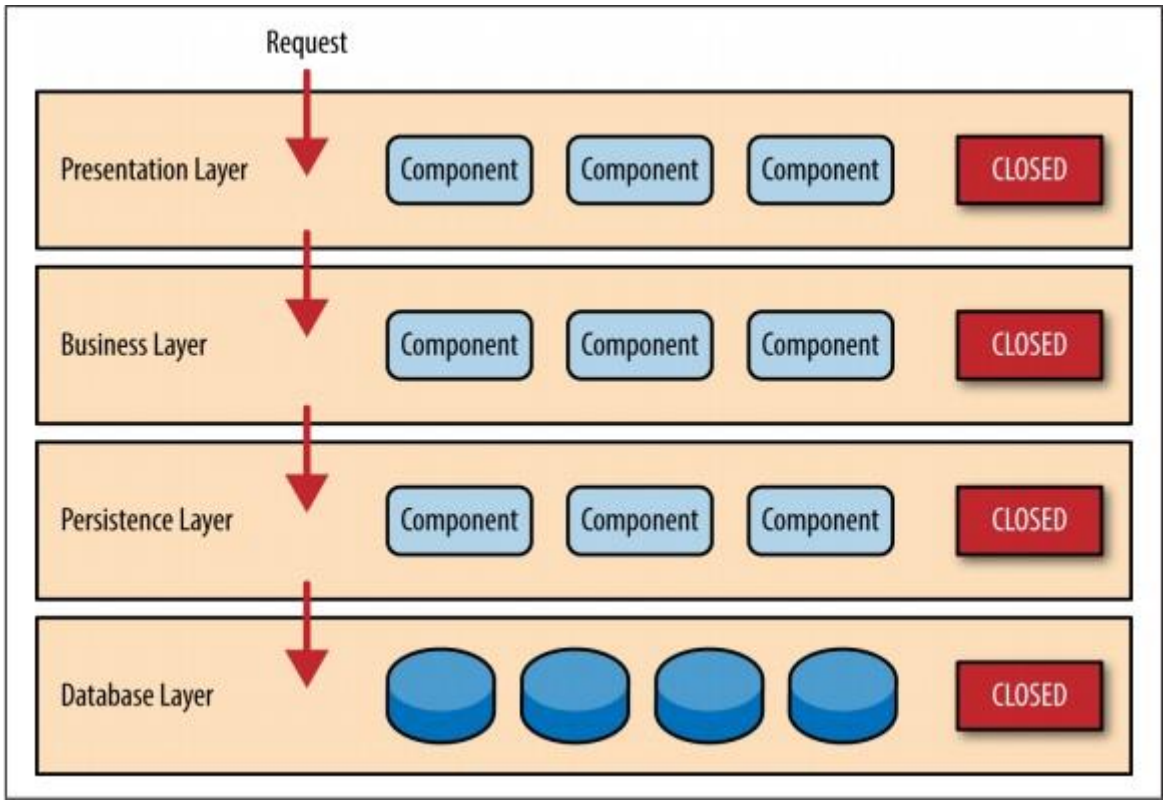
```
public class CarService: ICarService
{
    protected readonly IRepository _repository;

    public CarService(IRepository repository)
    {
        this._repository = repository;
    }

    public IEnumerable<ICar> getAll()
    {
        return this._repository.getAll<ICar>();
    }

    public ICar getById(string id)
    {
        return this._repository.getById<ICar>(id);
    }

    public ICar create(ICar car)
    {
        return this._repository.insert<ICar>(car);
    }

    public void update(ICar car)
    {
        return this._repository.update<ICar>(car.id, car);
    }

    public void remove(string id)
    {
        return this._repository.remove<ICar>(id);
    }
}
```

Slojevita arhitektura

Najpoznatija arhitektura organiziranje koda je slojevita arhitektura, poznata još kao i n-slojevita arhitektura. Komponente softvera organizirane su u horizontalne slojeve, od kojih svaki obavlja neku specifičnu zadaću unutar softvera. Slojevita arhitektura nigdje eksplicitno ne određuje koliko slojeva mora sadržavati. Međutim, većina softvera je organizirana u četiri sloja: prezentacijski sloj, sloj poslovne logike, perzistentni sloj i sloj baze podataka. Ponekad su sloj poslovne logike i perzistentni sloj objedinjeni u jedan sloj. Što je manji softver, moguće je imati samo tri sloja, dok veliki softveri mogu imati pet ili više slojeva.

Svaki sloj ima neku zadaću koju obavlja. Na primjer, prezentacijski sloj je zadužen za posluživanje korisničkog sučelja korisnicima ili komunikaciju s web preglednikom preko REST API ruta, dok je sloj poslovne logike zadužen za obavljanje specifičnih procesa manipuliranja s objektima ovisno o poslovnoj prirodi softvera. Svaki sloj je apstrakcija s obzirom na zadaću koju obavlja. Na primjer, prezentacijski sloj ne mora znati kako i odakle se podaci dohvaćaju ili kako se manipulira podacima. Njega se tiče samo kako ti podacima se trebaju prikazati ili doći do korisnika. Isto tako, sloj poslovne logike ne mora znati kako se podaci šalju korisniku ili odakle dolaze podaci. On je samo zadužen kako se primjenjuje poslovna logika na podatke koje dobije ili koje treba proslijediti prezentacijskom sloju.

Ovakav način arhitekture softvera omogućuje nam lakši razvoj, testiranje i samo održavanje softvera zbog jasno definirane razdvojenosti slojeva.



Svaki sloj slojevite arhitekture je zatvoren, što znači da zahtjev na putu mora proći redom svakim slojem od prezentacijskog sloja sve do sloja baze podataka.

Iako bi direktan pristup bazi već u prezentacijskom sloju bio brži, ovakav način ima svojih prednosti. Ključ je u konceptu nazvanom *izolacijski slojevi(eng. layers of isolation)*. Koncept izolacijskih slojeva znači da promjene u jednom sloju ne bi trebale utjecati na ostale slojeve. Promjena je izolirana unutar sloja komponente. Nadalje, koncept izolacijskih slojeva također znači da je svaki sloj neovisan o drugim slojevima, te da ne bi trebao znati ništa od logici i arhitekturi ostalih slojeva.

Na slici dolje je dan primjer kako zahtjev putuje komponentama od prezentacijskog sloja do paze. Crne strelice prikazuju zahtjev, a crvene odgovor koji dolazi nazad do prezentacijskog sloja i na kraju do korisnika. *Customer screen* je zadužen za prihvaćanje zahtjeva i prikazivanje informacija o kupcu. Njemu nije poznato gdje se podaci nalaze, kako ih dohvatiti niti koliko tablica u bazi sadrži tražene podatke. Sve što radi je prosljeđuje zahtjev sljedećoj komponenti *Customer delegate*. On treba znati koja komponenta u sloju poslovne logike može obraditi zahtjev na tražen način. *Customer object* u sloju poslovne logike je zadužen za spajanje svih informacija potrebnih da se zahtjev ispuni. On poziva *Customer dao* (*data access object*) u perzistentnom sloju da dohvati podatke. Oni dohvate tražene podatke u bazi podataka pomoću SQL upita i šalju podatke nazada.

Slojevita arhitektura je solidan koncept arhitekture opće namjene, što ju čini pogodnom za početak kodiranja većine aplikacija, posebice kada nismo sigurni koji koncept je najbolji za našu aplikaciju.

Prva stvar na koju trebamo obratiti pažnju su situacije kada zahtjev samo prolazi slojevima bez primjene neke logike na njima. Svaki softver koji je izgrađen na slojevitoj arhitekturi imati će ovakve situacije. Bitno je da je omjer između zahtjeva koje ne zahvaća ovaj problem i onih koje zahvaća bude 80:20. Ukoliko nam se dogodi da naš softver ima puno više zahtjeva koji samo prolaze slojevima

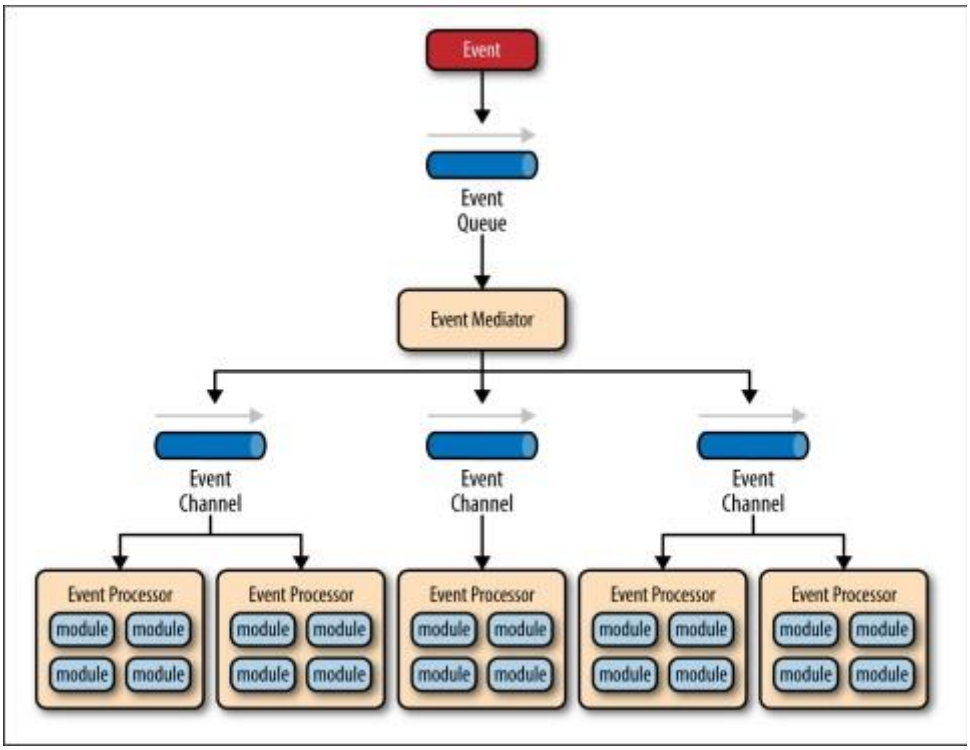
možemo razmisliti da otvorimo neke dijelove softvera i na neki način razbijemo neke slojeve. Ovaj problem naziva se *arhitecture sinkhole anti-pattern*.

Arhitektura navođena događajima

Arhitektura navođena događajima je naziv za distribuiranu asinkronu arhitekturu korištenu za stvaranje skalabilnih softvera. Softver napisan ovom arhitekturom vrlo je lako adaptirati i može ju se koristiti kako za male softverske projekte tako i za one velike i kompleksne. Čine ju potpuno razdvojene komponente koje imaju točno jednu zadaću koje asinkrono primaju i obrađuju svaki događaj. Ova arhitektura razlikuje se po dvije topologije, posredničke i brokerske.

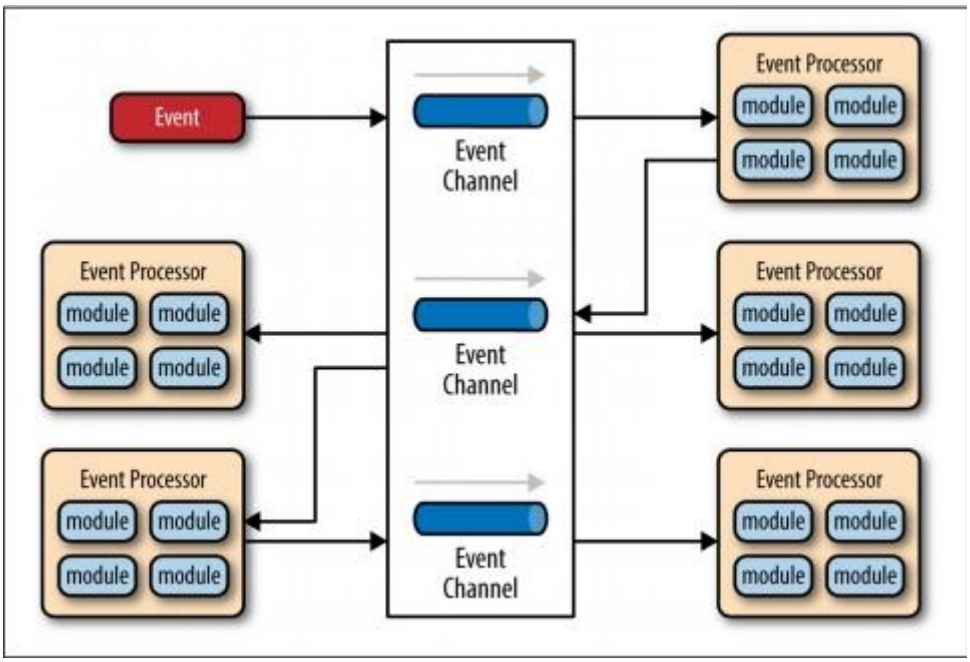
1. Posrednička topologija

Korisna je za događaje koji imaju više koraka koje trebamo poduzeti da obradimo sam događaj. Topologija se sastoji od četiri komponente: *event queues*, *event mediator, event channels* i *event processor.* Proces započinje slanjem događaja u red *event queue* koji kasnije prosljeđuje događaj komponenti *event mediator*. Nadalje, *event mediator za*primljeni događaj šalje dodatne asinkrone događaje komponentama *event channels* da izvrše svaki korak. *Event processor* primjenjuje poslovnu logiku na zaprimljeni događaj. Uobičajeno je da imamo od nekoliko do čak nekoliko stotina komponenti *event queue*. Ova arhitektura nigdje ne određuje kakva mora biti implementacija komponente *event queue*. Događaje unutar posredničke topologije dijelimo na izvorne događaje (eng. *initial event*), koje smo zaprimili na početku, i procesirajuće događaje (eng. processing event) koje generira komponenta *event mediator* i koji se kasnije proslijeđuju event-processing komponenti. *Event Mediator* komponenta zadužena je za orkestriranjem svakog koraka koji sadrži izvorni događaj. Za svaki korak generira se procesirajući događaj i šalje dalje. Bitno je još napomenuti kako *event mediator* ne primjenjuje poslovnu logiku za obradu događaja, međutim zna svaki korak koji je potrebno poduzeti za njegovu obradu.



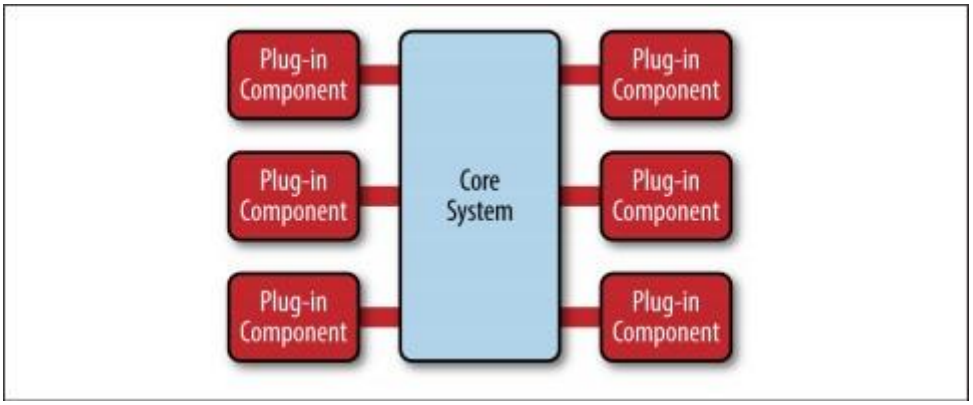
2. Broker topologija

Razlikuje se od prethodne u tome što nema centralne komponente *event mediator*. Čine ju samo dvije komponente, *event processor* i *broker* komponenta. *Broker* komponenta sadržava sve komponente *event channel* koje se koriste pri obradi događaja. Ovdje svaka komponenta *event processor* je odgovorna za obradu događaja i objavu novog događaja koji signalizira da je akcija izvršena. Taj novi događaj onda može pokupiti neki drugi *event processor* i primijeniti neku logiku na njemu, pa ponovno objaviti novi događaj itd.



Mikrojezgrena arhitektura

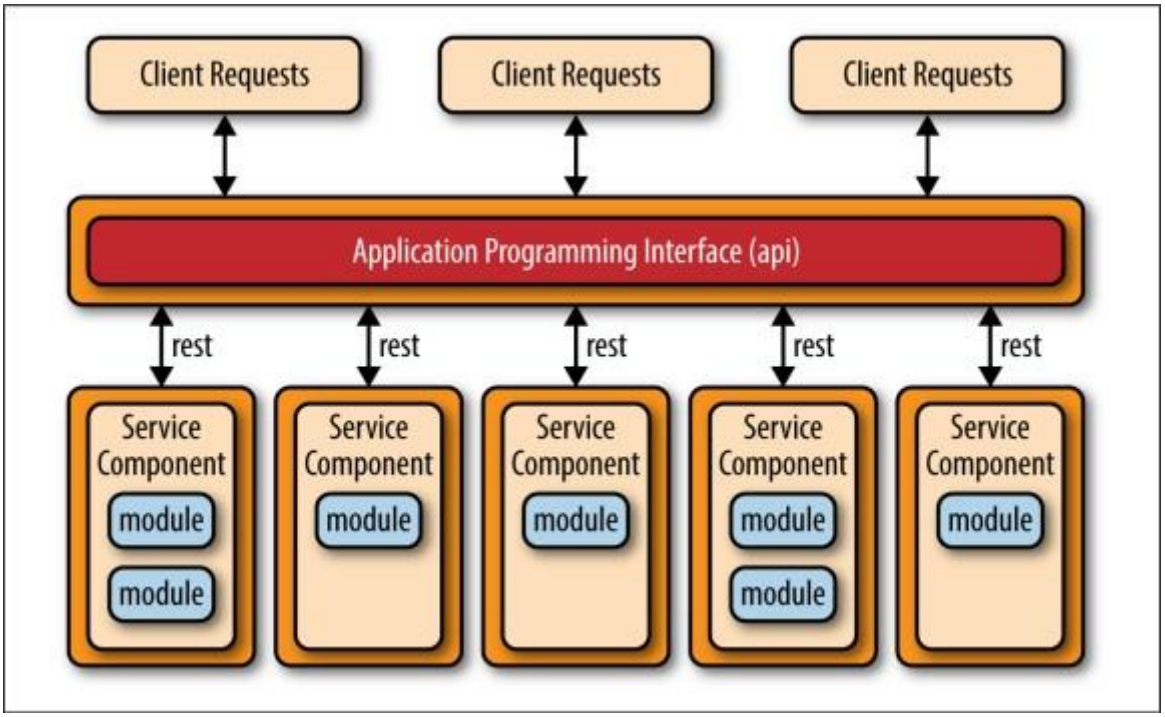
Mikrojezgrena arhitektura je arhitektura za dizajniranje softvera koji je namijenjen za preuzimanje kojeg čini jezgra proizvodi i dodaci. Arhitektura nam dozvoljava da dodatke samo uključimo jezgri softvera pružajući novu funkcionalnost, ekstenziju koja je izolirana i odvojena. Čine ju dvije komponente: jezgra (eng. *core system*) i moduli koje uključujemo (eng. *plug-in modules*). Jezgra obično sadržava minimalnu funkcionalnost potrebnu za rad softvera.



Plug-in moduli su samostalne i nezavisne komponente koje sadržavaju dodatne funkcionalnosti kojima je svrha da poboljšaju postojeće funkcionalnosti ili dodaju neku novu. Generalno, plug-in moduli bi trebali biti nezavisni jedni o drugima, ali moguće je da neki modul za ispravan rad zahtjeva i neki drugi modul. Jezgra mora znati koji moduli su dostupni i kako ih koristiti. Jedan standardan način je preko registara koji sadržava osnovne informacije o svakom modulu kao što su ime, modeli podataka i informacije o tome kako se odvija komunikacija s jezgrom. Veza između jezgre i modula može se uspostaviti razmjenom poruka, preko web servisa ili standardnim načinom instanciranjem objekta koji obavlja direktnu komunikaciju. Tip veze ovisi o vrsti softvera koji radimo te o njegovoj veličini. Neki primjera softvera koji koriste ovakvu arhitekturu su web preglednici koji se mogu proširiti raznim nadogradnjama, te editori kodi.

Mikroservisna arhitektura

Mikroservisni pristup arhitekture softvera svaku funkcionalnost poslovne logike softvera razdvaja u različite procese, manje softvere umjesto tradicionalnog načina arhitekture gdje je cijeli softver jedan proces. Ova arhitektura pruža vrlo fleksibilno skaliranje softvera jer možemo različite dijelove, tj. mikroservise postaviti na različita računala. Ova vrste arhitekture je distribuirana arhitektura što znači da su svi mikroservisi potpuno razdvojeni jedan od drugih te im se pristupa preko nekog protokola, najčešće REST. Na svaki mikroservis možemo gledati kao na jednu komponentu softvera koju je moguće nezavisno zamijeniti ili nadograditi.



Kako je cijeli softver ograniziran oko nezavisnih komponenti, a ne oko tehnologija (UI, server, baza podataka) na svakom mikroservisu mogu raditi različiti timovi koji se fokusiraju samo na određenu funkcionalnost za koju je zadužen mikroservis.