Arhitektura web aplikacija

Principi softverskog inženjerstva, *Elektrotehnički fakultet Univerziteta u Beogradu*

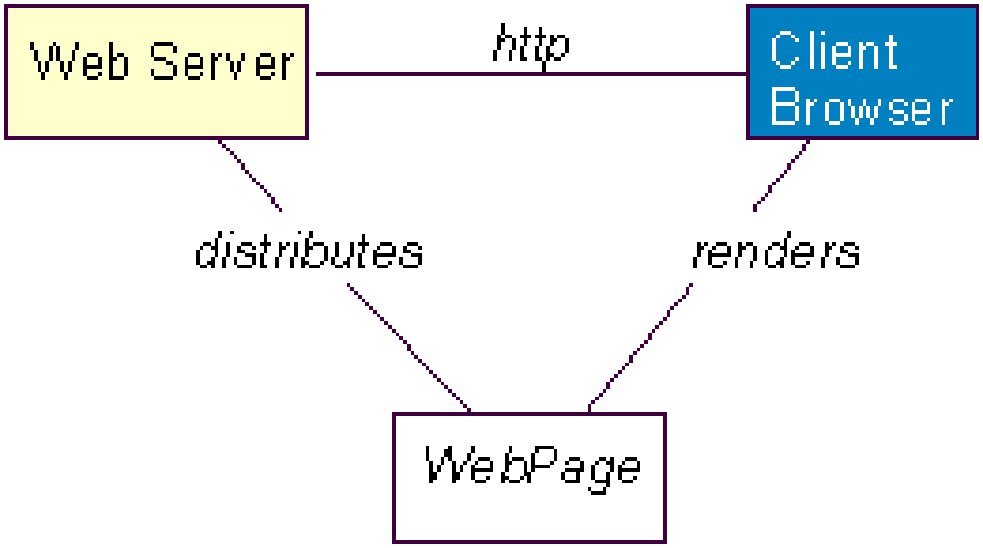
UVOD

Arhitektura softvera

- Definiše celokupnu strukturu sistema, glavne komponente i način na koji međusobno interaguju; daje opšti pregled sistema i definiše kako komponente međusobno sarađuju da bi ostvarile funkcionalnosti sistema.
- Predstavlja deljeno razumevanje dizajna sistema od strane programera, zajednički rečnik i razumevanje strukture i ponašanja sistema.
- Arhitektonske odluke u projektovanju sistema: (važne) odluke o opštoj strukturi i organizaciji sistema, dodela odgovornosti pojedinim komponentama sistema i definisanje načina komunikacije među njima.

Statički veb sajt

- Web sajt sadrži tri glavne komponente: **veb server**, mrežnu konekciju i jedan ili više **klijentskih pregledača**.
- Veb server distribuira **(veb) strane** formatiranih informacija klijentima koji ih zahtevaju.
- Zahtev se postavlja preko mrežne konekcije i upotrebljava HTTP(S) protokol.



URI, URN i URL

- Klijemt mora da ima način da saopšti serveru koji resurs zahteva.
- Za jednoznačnu identifikaciju resura koriste se URI (Uniform Resource Identifier). Oni mogu biti tipa URN i URL
- URN (Uniform Resource Name) imenuje resurs na jednoznačan način na internetu tj. daje informaciju o identitetu ali bez informacije o lokaciji. Primeri:

  urn:isbn:0451450523 jednoznačno identifikuje knjigu

  urn:uuid:6e8bc430-9c3a-11d9-9669-0800200c9a66
- URL (Uniform Resource Locator) ili veb adresa, specificira ver resurs ali i njegovu lokaciju na mreži i protokol za njegovo dohvatanje. Primeri:

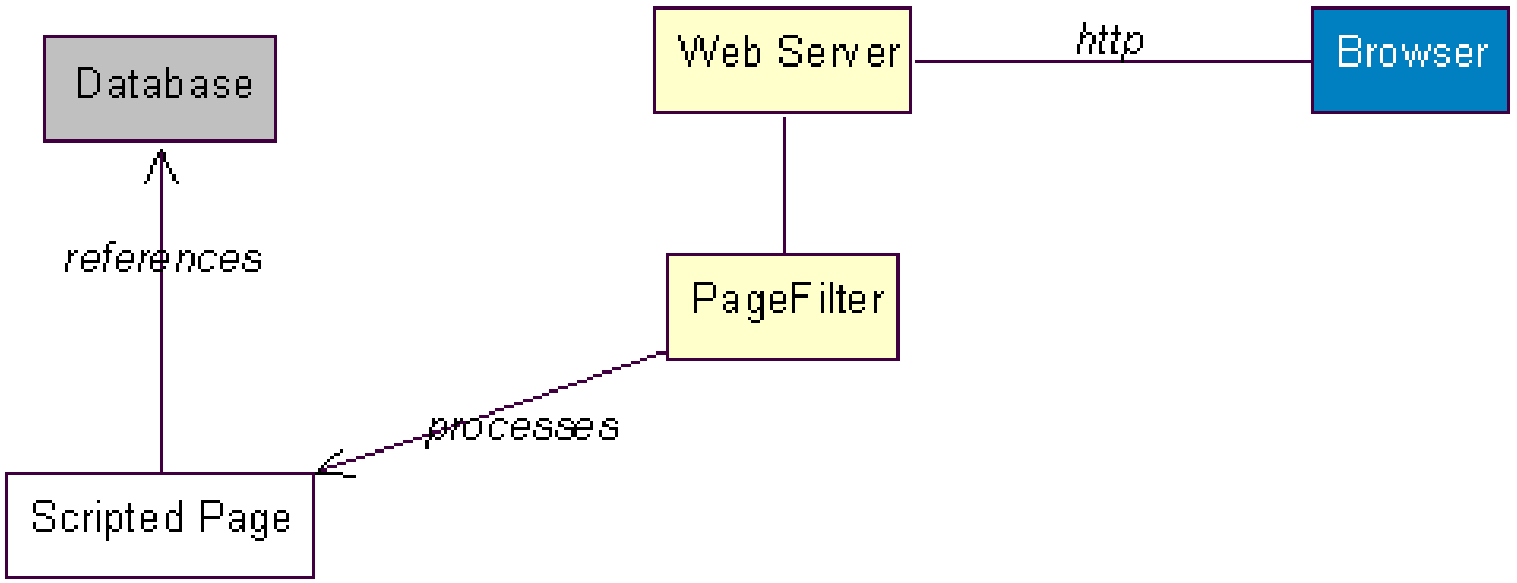
  http://www.ietf.org/rfc/rfc2396.txt

  ftp://ftp.is.co.za/rfc/rfc1808.txt

  mailto:pera@example.com

Dinamički veb sajt

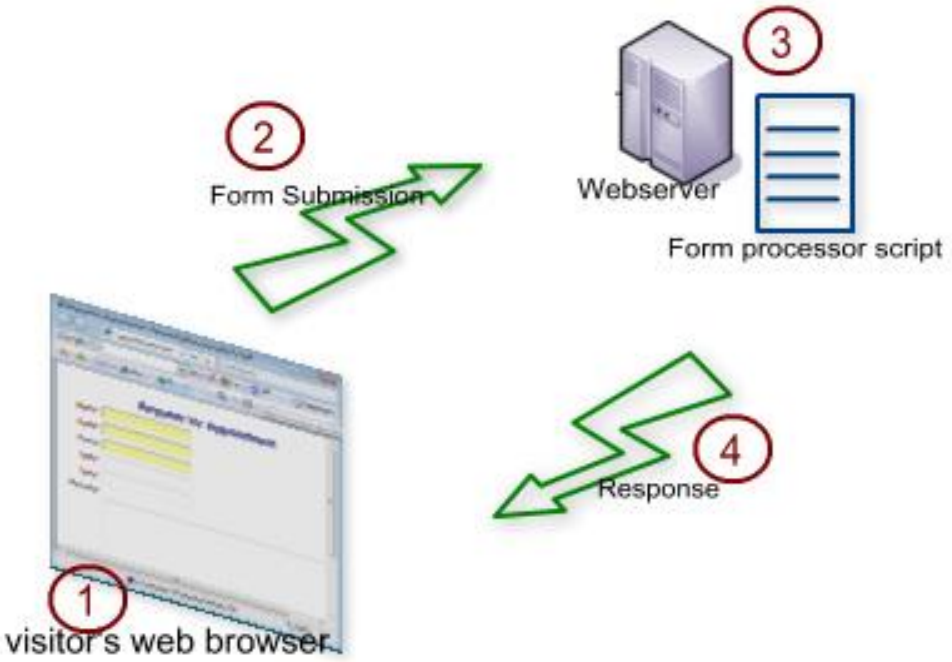
- U nekim situacijama sadržaj stranice nije statički određen (memorisan u fajlu), već se sklapa dinamički od informacija iz baze podataka (ili drugih repozitorijuma informacija) i formatira na osnovu niza instrukcija (skripta) koji se čuva u fajlu. Web server upotrebljava filter (intepreter) stranica da interpretira i izvrši skriptove. Web sajtovi koji primenjuju ovu strategiju nazivaju se dinamički sajtovi. Skriptovanje može biti i u pregledaču, na strani klijenta.



HTML Forme

- Najviše korišćeni mehanizam za sakupljanje ulaznih podataka od korisnika je putem HTML formi.
- HTML forma je kolekcija ulaznih polja koji se prikazuju (render-uju) kao web stranica. Osnovni ulazni elementi su: tekstualno polje (text box), tekstualna oblast (text area), polje za čekiranje (checkbox), grupa radio dugmadi (radio button group) i selekciona lista (selection list).
- Svi ulazni elementi na formi identifikovani su imenom ili identifikatorom. Svaka forma se asocira sa akcionom stranicom (tako što se navede URL te stranice). Akciona stranica služi da primi i procesira informacije sadržane u popunjenoj formi. Skoro uvek reč je o stranici na serveru koja sadrži skript za procesiranje.

HTML Forme

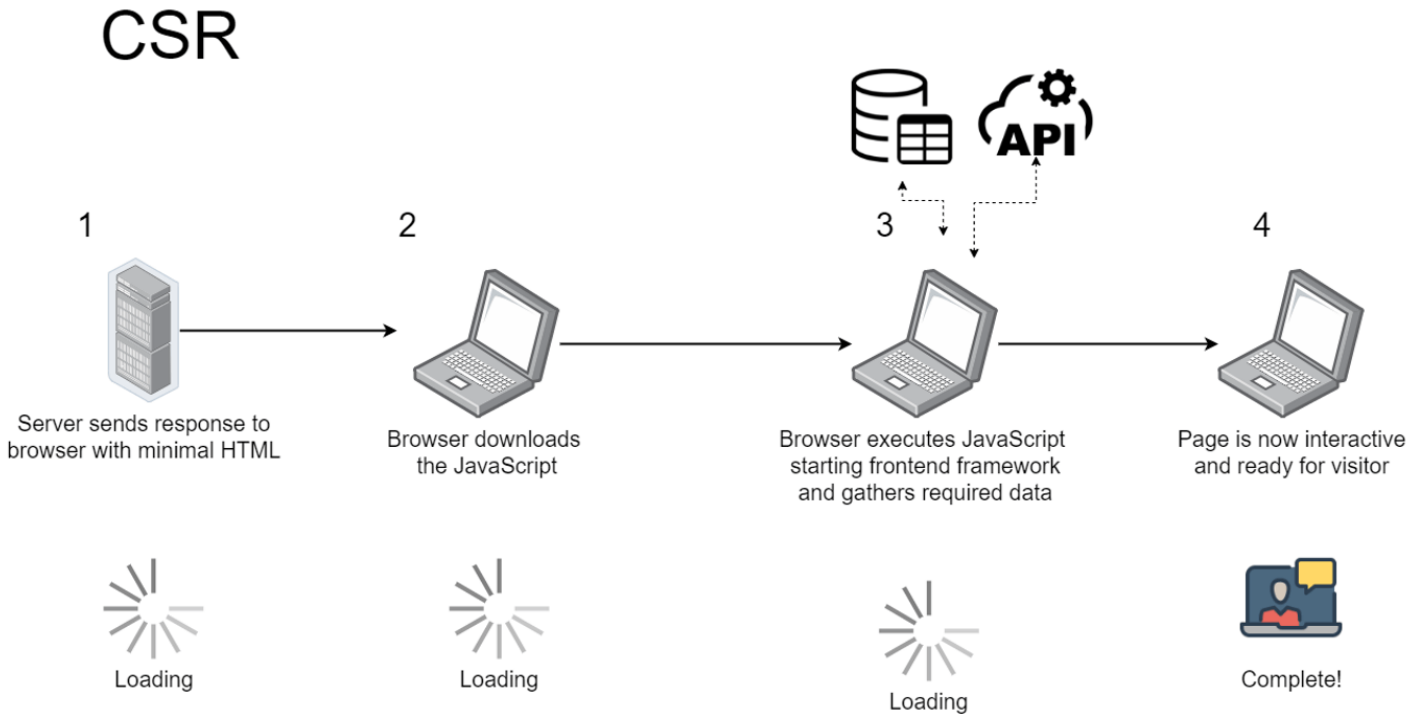


- Kada popuni formu, korisnik podnosi (submits) formu na server zahtevajući akcionu stranicu sa servera. Web server pronalazi tu stranicu i interpretira (izvršava) sadržani programski skript. Programski skript može da čita informacije koje su podnete sa forme.
- Koristeći Ajax protokol, klijentska strana može slati i primati podatke sa servera asinhrono (u pozadini) bez narušavanja izgleda i ponašanja postojeće strane.

TIPOVI ARHITEKTURA VEB APLIKACIJA

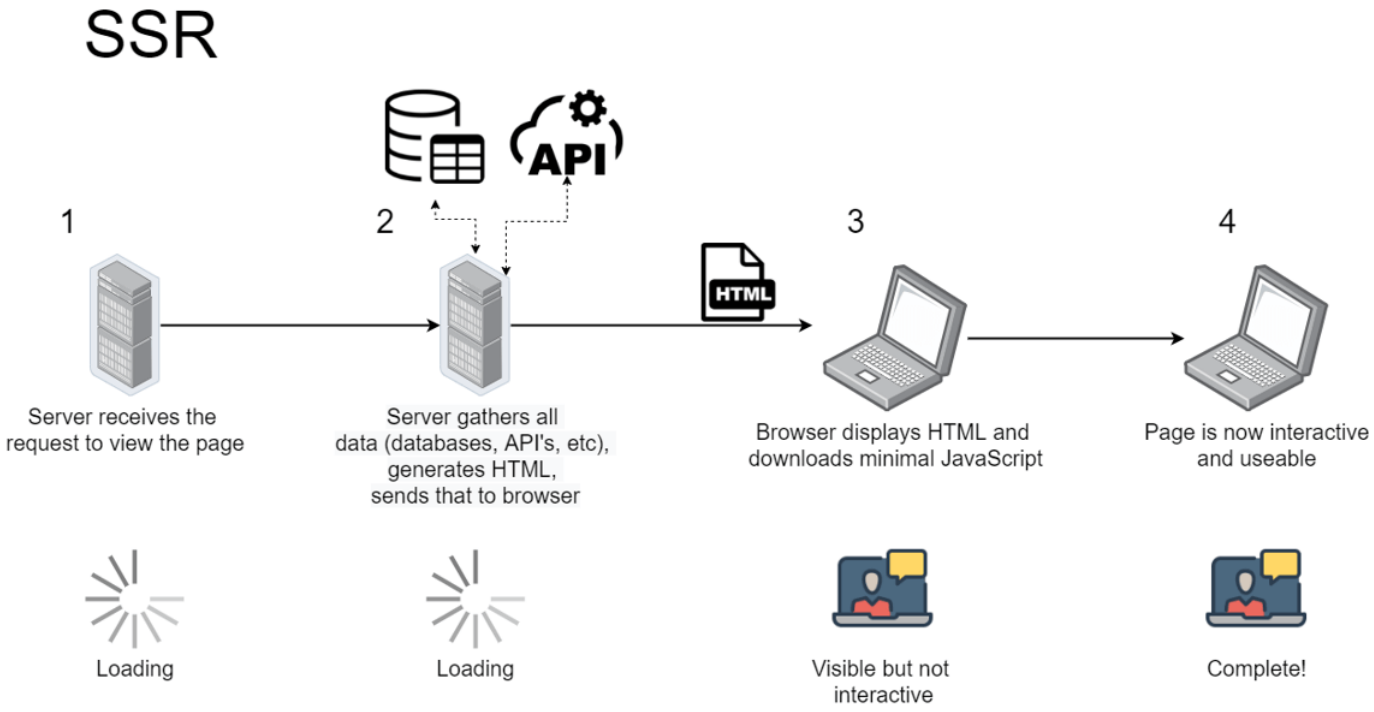
Client Side Rendered aplikacije

- CSR aplikacije, su veb aplikacije u kojima se većina logike aplikacije pokreće u veb pretraživaču klijenta. To znači da klijentov pretraživač preuzima jednu HTML datoteku, koja uključuje JavaScript kod koji je odgovoran za prikazivanje celog korisničkog interfejsa i preuzimanje podataka sa servera preko API-ja. Primeri popularnih okvira na strani klijenta za pravljenje veb aplikacija ovog tipa uključuju Angular, React i Vue.js.



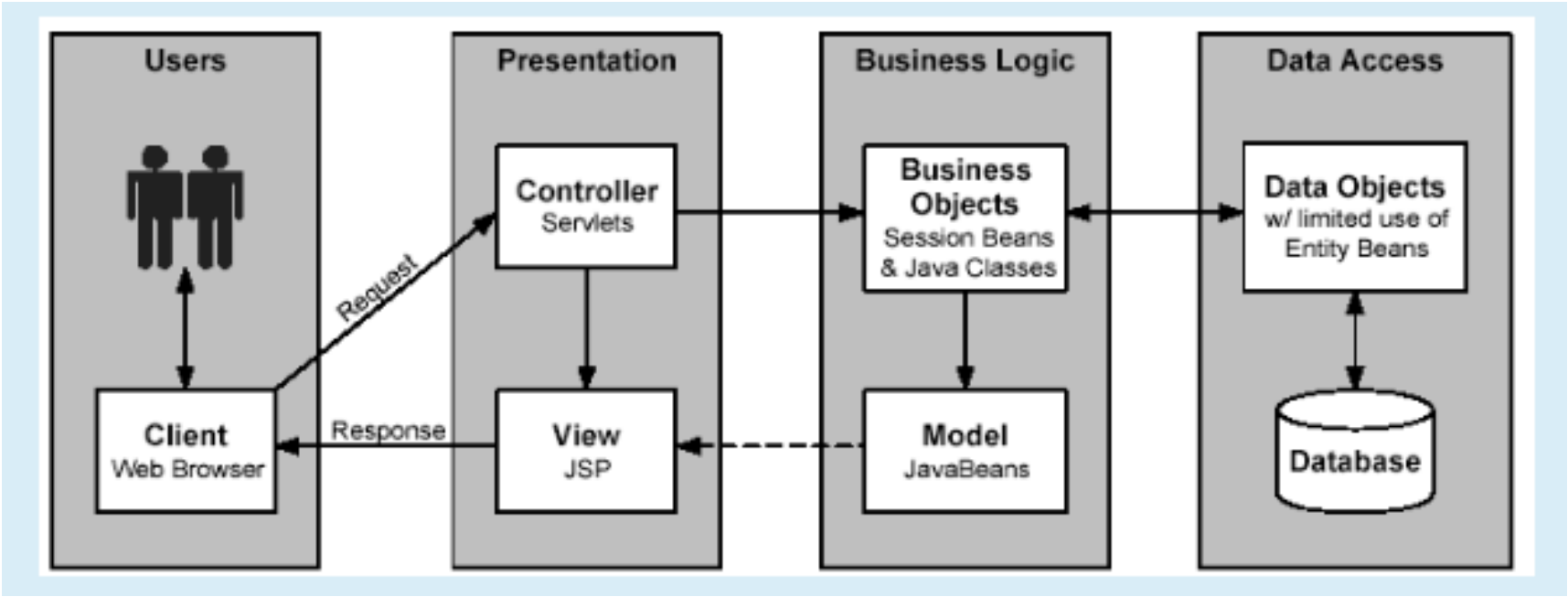
Server Side Rendered aplikacije

- SSR aplikacije, su veb aplikacije u kojima se većina logike aplikacije pokreće na serveru. To znači da server generiše HTML sadržaj dinamički na osnovu zahteva korisnika i šalje rezultujući HTML pretraživaču klijenta za prikaz. U ovom modelu, kod na strani servera može uključivati poslovnu logiku, pristup podacima i HTML prikazivanje. Primeri popularnih okvira na strani servera za pravljenje SSR veb aplikacija uključuju Django i Laravel.



Višeslojne veb aplikacije

- Višeslojne ili n-slojne aplikacije, su veb aplikacije u kojima je logika aplikacije podeljena na više slojeva (tiers, layers) odgovornih za različite aspekte funkcionisanja aplikacije.
- Svaki nivo se može razviti korišćenjem različitih tehnologija, u zavisnosti od specifičnih zahteva aplikacije i preferencija razvojnog tima. Ključno je osigurati da je svaki nivo odvojen od ostalih i da se komunikacija između nivoa obavlja preko dobro definisanih API-ja ili interfejsa.

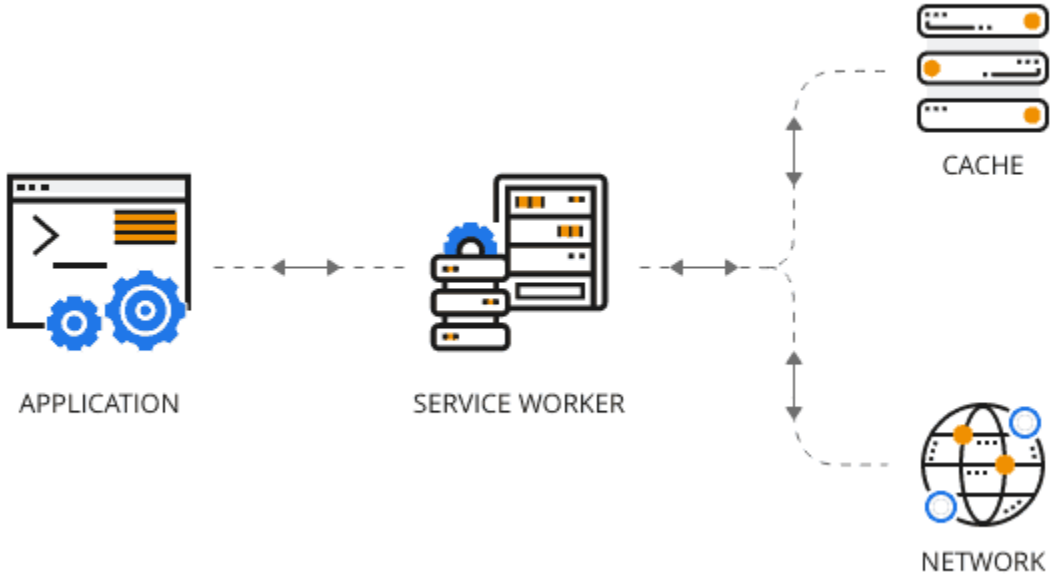


Višeslojne veb aplikacije

- Višeslojne ili n-slojne aplikacije: primer podele na slojeve:
  - Prezentacioni sloj odgovoran za korisnički interfejs i korisničko iskustvo aplikacije. Tehnologije: HTML, CSS, JavaScript, React.js, Angular.js, Vue.js.
  - Aplikativni sloj: odgovoran za sprovođenje poslovne logike i aplikativnih radnih tokova. Tehnologije: PHP, Python, Ruby, Java, .NET, Node.js.
  - Servisni sloj: odgovoran za skup aplikativnih servisa i APIja koje može koristiti prezenzacioni sloj ili eksterne aplikacije. Tehnologije: RESTful APIs, SOAP, GraphQL, Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure.
  - Sloj pristupa podacima: odgovoran za skladištenje i dohvatanje podataka. Tehnologije: MySQL, PostgreSQL, MongoDB, Oracle, Cassandra, Redis.
  - Infrastrukturni sloj: odgovoran za upravljanje infrastrukturom na kojoj počiva aplikacija, uključujuće servere, baze podataka, balansere opterećenja itd. Tehnologije: Docker, Kubernetes, Amazon Elastic Compute Cloud (EC2), Microsoft Azure Virtual Machines, Google Cloud Compute Engine.

Progresivne veb aplikacije

- Progresivne veb aplikacije koriste kombinaciju veb tehnologija kao što su HTML, CSS i JavaScript, zajedno sa funkcijama kao što su service workers, projektovane su da rade na korisničkom uredjaju i mogu se instalirati na korisnički uredjaj kao klasična aplikacija. Service workers su skripte koje se pokreću u pozadini i upravljaju funkcijama van mreže, push obaveštenjima i keširanjem. PVA takođe koriste tehnike responzivnog dizajna kako bi obezbedili dosledno korisničko iskustvo na različitim uređajima i veličinama ekrana. Jedna od prednosti PVA je ta što korisnicima nude funkcije čak i kada postoji ograničena ili nikakva internet konekcija. okviri za pravljenje PVA uključuju React, Angular i Vue.js



Serverless veb aplikacije

- Veb aplikacije bez servera (Serverless) su napravljene korišćenjem usluga zasnovanih na oblaku i ne zahtevaju da programer postavi ili njime upravlja bilo kakvu serversku infrastrukturu. Umesto toga, programeri pišu funkcije koje se pokreću u oblaku, a ove funkcije upravljaju logikom aplikacije.
- Arhitekture bez servera su pogodne za aplikacije koje imaju nepredvidive ili veoma promenljive obrasce saobraćaja, kao i za aplikacije koje zahtevaju visok nivo skalabilnosti i dostupnosti. Popularni okviri bez servera uključuju AVS Lambda, Google Cloud funkcije i Azure funkcije.

Monolitne i distribuirane veb aplikacije

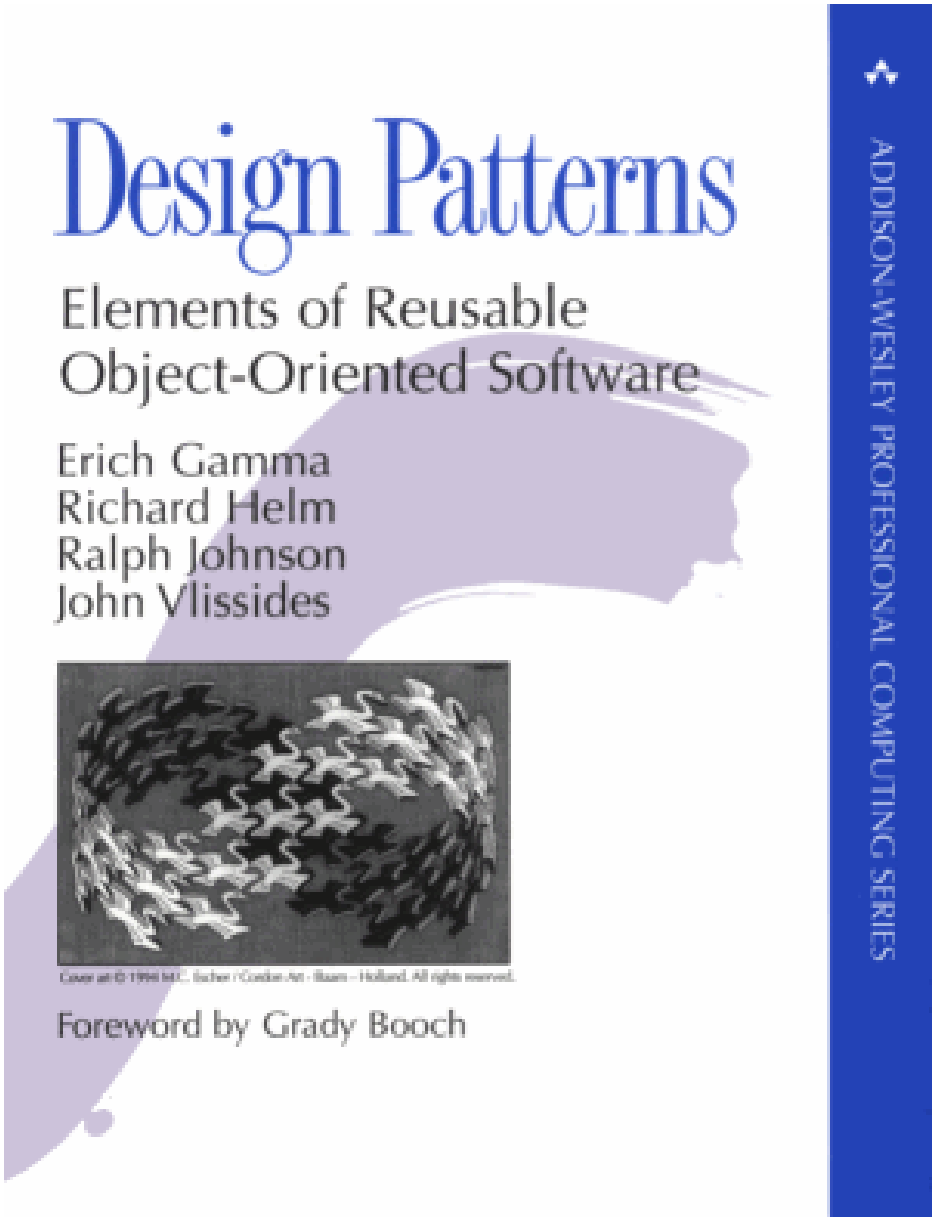
- Definicija: Monolitna arhitektura je pristup gde je cela veb aplikacija dizajnirana kao jedna, samostalna jedinica koja radi na jednom serveru ili grupi servera.
- Prednosti: Monolitna arhitektura se lako razvija i primenjuje. Jednostavnije je testirati i otklanjati greške, a u nekim slučajevima može biti efikasnije od distribuiranih arhitektura. Takođe je lakše obezbediti jer sve komponente rade na jednoj mašini.
- Nedostaci: Monolitna arhitektura može postati složena i glomazna kako aplikacija raste. Promene u jednom delu aplikacije mogu zahtevati ponovno raspoređivanje cele aplikacije. Takođe može biti teško horizontalno skalirati jer sve komponente rade na jednom serveru.
- Alternative: Kako veb aplikacije postaju složenije i zahtevaju veću skalabilnost i agilnost, programeri se sve više okreću mikroservisima i drugim distribuiranim arhitekturama kao alternativama monolitnoj arhitekturi.
- **Dizajnerski obrasci**: arhitektura veb aplikacije može da koristi različite obrasce dizajna, kao što su Model-Viev-Controller, Domain Driven Design, Event Driven Architecture, Inversion of Control

OBRASCI ZA ARHITEKTURU VEB APLIKACIJE

Projektni uzorci (Design patterns)

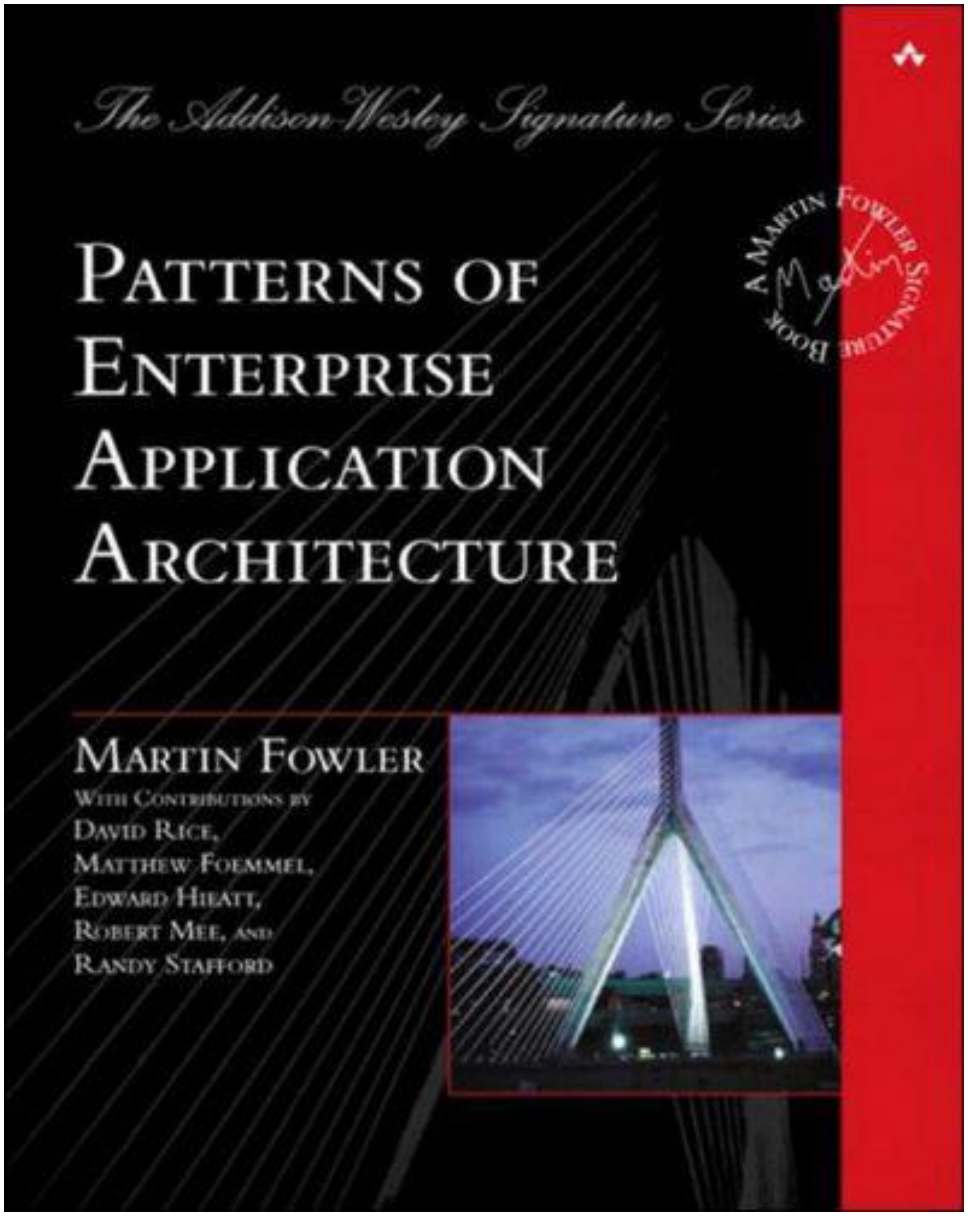
- *"Svaki uzorak opisuje **problem** koji se javlja iznova i iznova u našem okruženju, a zatim opisuje **suštinu rešenja** tog problema, na takav način da se takvo rešenje može **koristiti ponovo** milion puta, a da se nikada to ne uradi dva puta na isti način“*
  - Christopher Alexander, poznati arhitekta (za građevine)
- Prednosti korišćenja uzoraka u projektovanju:
  - Učenje na tuđem iskustvu
  - Uzorci nisu jezički zavisni
  - Izolovani elementi rešenja koji se mogu međusobno kombinovati
  - Uzorci čine rečnik za sporazumevanje među graditeljima sistema
  - Mnoge biblioteke i aplikativni okviri su konstruisani prema određenim projektnim uzorcima i poznavanje tih uzoraka omogućuje bolje razumevanje bibl. I okvira

Izvori informacija o projektnim uzorcima



- "Gang of Four" patterns
- klasični opštenamenski uzorci za projektovanje OO softvera
  - Ovo se proučava na kursu projektovanja softvera

Izvori informacija o projektnim uzorcima



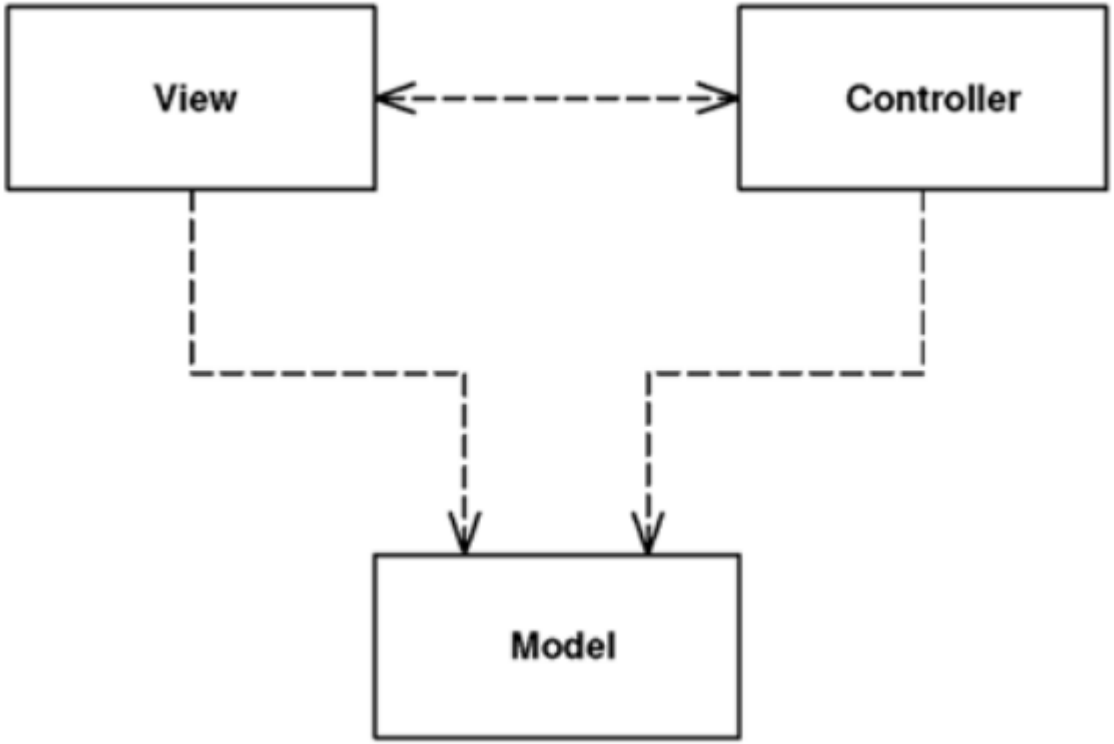
- PoEAA patterns
  - Razmotrićemo samo neke uzorke iz ove knjige, koji se često koriste u web programiranju
  - Za razliku od GoF uzoraka, koji su uglavnom na nivou detaljnog projektovanja, PoEAA se odnosi na nivo arhitekture
- Ažurnija verzija Fowlerovih obrazaca je na sajtu:

  https://martinfowler.com/architecture/

OBRAZAC MODEL POGLED KONTROLER (MVC)

Model View Controller

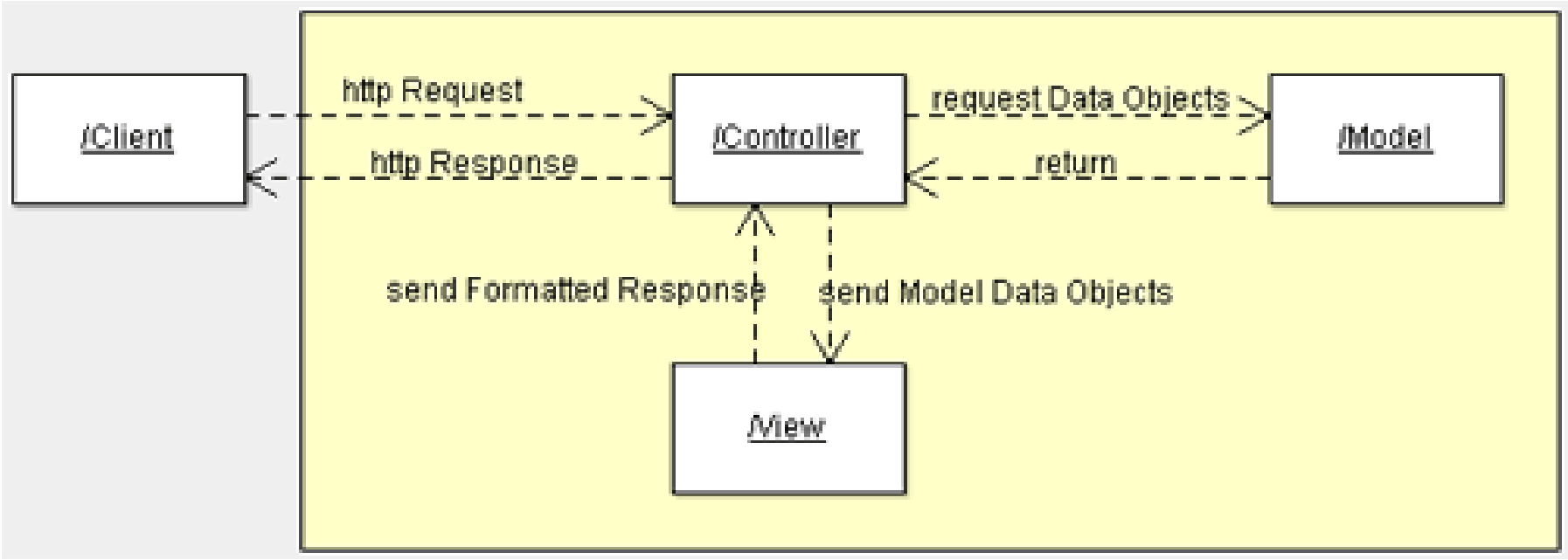
- *Deli interakciju sa korisničkim interfejsom u tri različite uloge.* Model prikaz kontroler je najviše korišćen obrazac za savremene web aplikacije. Korišćen je prvi put u Smalltalk-u kasnih 1970-ih, a zatim usvojen i popularizovan u Javi. Trenutno postoji više od desetak PHP web okvira na osnovu MVC obrasca



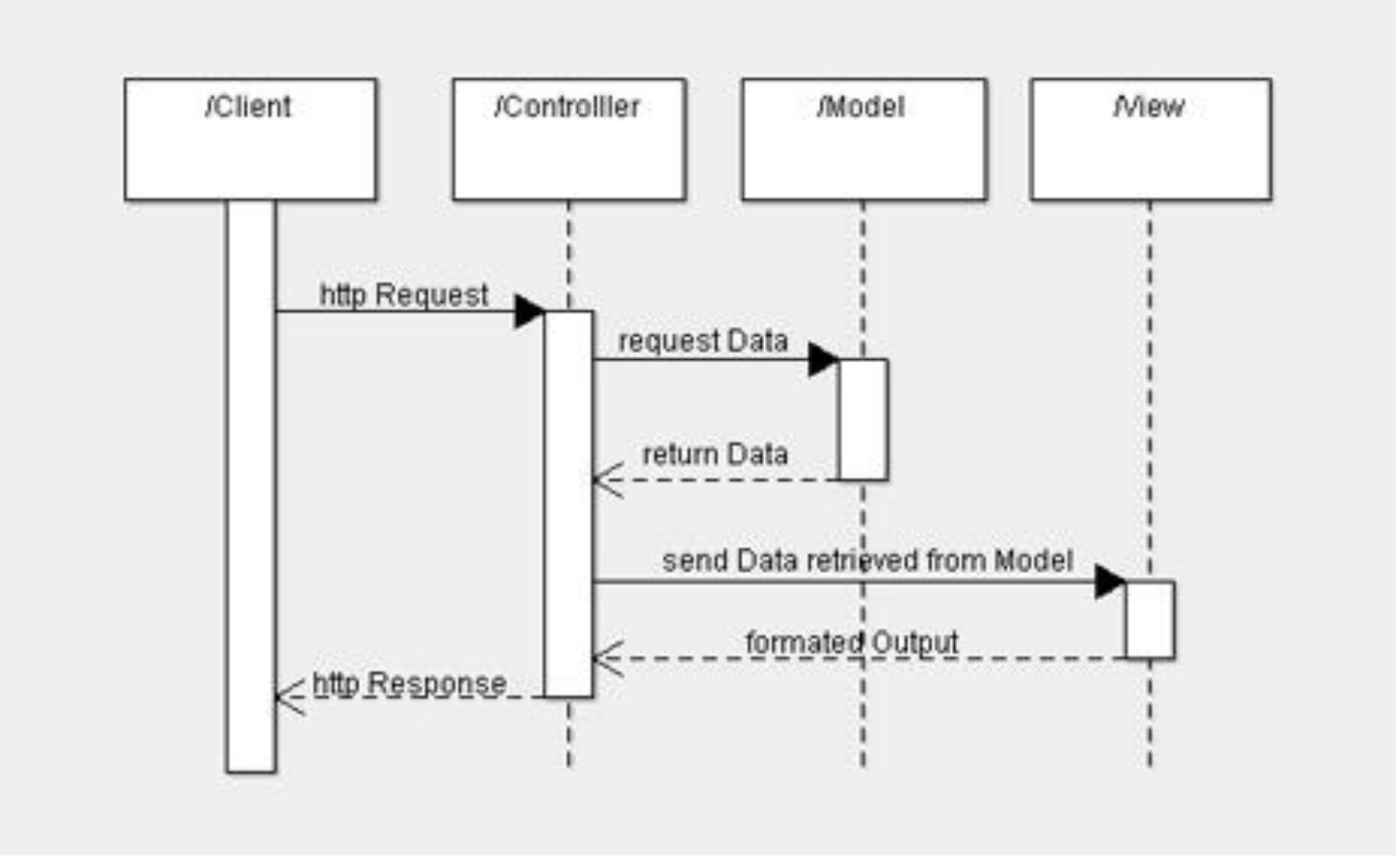
Model View Controller

- MVC uzorak razdvaja aplikaciju u 3 celine: Model, Prikaz i Kontroler:
- Model je odgovoran za upravljanje podacima, on čuva i vraća entitete korišćene od strane aplikacije, obično iz baze podataka, i sadrži logiku aplikacije.
- Prikaz (prezentacija) je odgovoran za prikazivanje podataka koje pruža model u određenom formatu. On ima sličnu upotrebu kao šabloni prisutni u nekim popularnim web aplikacijama, kao što su WordPress, Joomla, ...
- Kontroler upravlja modelom i prikazom da rade zajedno. Kontroler primi zahtev od klijenta, pokreće model za obavljanje traženih poslova i šalje podatke na Prikaz. Prikaz formatira podatke za prikaz korisniku, u HTML formatu.

Dijagram kolaboracije MVC



Dijagram sekvence MVC



Jednostavna implementacija MVC

- Evo primera Python aplikacije koja implementira MVC obrazac za jednostavnu studentsku veb aplikaciju koristeći CGI bez upotrebe okvira, sa klasama Model, Viev i Controller u zasebnim modulima:

student_model.py

```
class StudentModel:
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no
```

student_view.py

```
class StudentView:
  def get_student_details(self):
    print("Content-type: text/html\n")
    print("<html><body>")
    print("<form method='POST' action='/cgi-bin/student_cgi.py'>")
    print("Name: <input type='text' name='name'><br>")
    print("Roll Number: <input type='text' name='roll_no'><br>")
    print("<input type='hidden' name='action' value='submit'>")
    print("<input type='submit' value='Submit'>")
    print("</form></body></html>")

  def show_student_details(self, model):
    print("Content-type: text/html\n")
    print(f"<html><body>Student Details: Name - {model.name}, Roll Number - {model.roll_no}</body></html>")
```

Jednostavna implementacija MVC

- StudentViev klasa upravlja prikazivanjem HTML forme za unošenje detalja o učeniku, a StudentController je odgovoran za slanje forme i ažuriranje StudentModela u skladu sa tim.
- **student_controller.py**

```
from student_model import StudentModel
from student_view import StudentView
import cgi

class StudentController:
    def __init__(self):
        self.model = StudentModel("", "")
        self.view = StudentView()

    def get_student_details(self):
        self.view.get_student_details()

    def set_student_details(self, form):
        name = form.getvalue('name')
        roll_no = form.getvalue('roll_no')
        self.model = StudentModel(name, roll_no)
        self.view.show_student_details(self.model)
```

Jednostavna implementacija MVC

- Glavna ulazna tačka je u student_cgi.pi, koja inicijalizuje kontroler i usmerava zahteve na osnovu parametra "action" u CGI zahtevu.
- **student_cgi.py**

```
#!/usr/bin/env python3

from student_controller import StudentController
import cgi

def main():
    form = cgi.FieldStorage()
    if "action" in form:
        action = form.getvalue("action")
    else:
        action = "details"

    controller = StudentController()

    if action == "details":
        controller.get_student_details()
    elif action == "submit":
        controller.set_student_details(form)


if __name__ == "__main__":
    main()
```

Kako pokrenuti python cgi aplikaciju?

1. Podrazumeva se da na Windows-u postoji instaliran python.
2. Raspakovati arhivu sa fajlovima iz primera u neki folder npr. c:\temp. Skriptovi se nalaze u c:\temp\cgi-bin folderu.
3. Otvoriti Windows command prompt I pozicionirati se u c:\temp.
4. Pokrenuti interni python server na localhost-u komandom:
   `python -m http.server --cgi --bind localhost`
5. Pristupiti skriptu putem veb pregledača na stranici
   `http://localhost:8000/cgi-bin/student_cgi.py`.
6. Ako je sve u redu, izlaz iz skripta (forma za unos imena i broj indeksa I dugme Submit) treba da se prikaže u pregledaču. Ako pregledač nudi da snimi fajl student_cgi.py onda je došlo do greške u izvršavanju skripta, error log može da se pregleda u prozoru gde je pokrenut server. Server se zaustavlja sa ctrl-C.

OBRAZAC ARHITEKTURA VOĐENA DOGAĐAJIMA

Arhitektura vođena događajima

- Event driven architecture (EDA) je obrazac arhitekture softvera koji naglašava kreiranje, otkrivanje, konzumaciju i reakciju na događaje koji se dešavaju unutar sistema. U EDA, događaji su centralni za arhitekturu, a komponente sistema interaguju tako što proizvode i konzumiraju događaje.
- Reagovanje na događaje u aplikaciji može biti sinhrono (da se ceo događaj obradi pre nego što se pređe na obradu sledećeg), ali pravi dobici vide se kada se pravi asinhrono reagovanje na događaje, jer tada aplikcija može da radi druge stvari dok na primer čeka na ulaz/izlaz.

Sinhrono i asinhrono programiranje

- U ljudskoj komunikacji telefonski razgovor je primer sinhronog komuniciranja, jer se obrada i odziv na ulazne informacije dešava odmah po prijemu bez čekanja.
- Elektronska pošta je primer asinhrone komunikacije, jer primalac pošte će pročitati poruku i odgovoriti na nju kada mu to bude zgodno.
- Slično ovome, u asinhronom programiranju, postoji više taskova, gde više taskova može da obavlja različiti posao i ne moraju da čekaju jedan drugog na završetak posla. Pri tome postoji dogovoreni mehanizam gde jedan task može da obavesti drugi da je posao završen i da je rezultat, ako postoji, dostupan.

Multitasking, višenitna i višeprocesna obrada

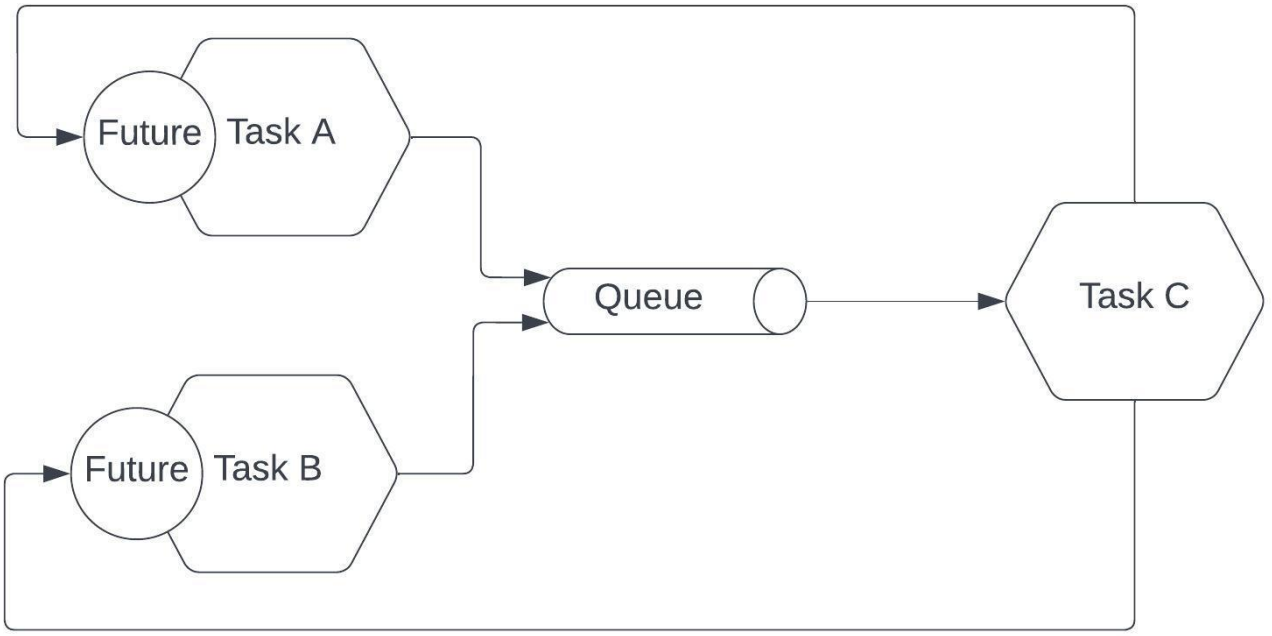
- Postoje različiti načini za realizaciju asinhrone obrade događaja u veb programiranju:
  - Kod kooperativnog multitaskinga (korutine) programer ima punu kontrolu nad kreiranjem taskova i njihovim raspoređivanjem (scheduling). Ceo program se i dalje izvršava u jednoj niti procesora. Ovaj način pogodan je kada je procesiranje I/O bound i postoji veliki broj sporih I/O zahteva. Za python biblioteka je asyncio, za PHP to su generatori (ili PHP 8 fibers).
  - Kod višenitnog programiranja program može da kreira više niti u okviru jednog procesa pod kontrolom operativnog sistema. Pogodno je za procesiranje I/O bound aplikacija kada ima manji broj brzih I/O zahteva, Python ima ograničenje da unutar jednog procesa samo jedna nit može da izvršava interpreter u jednom trenutku. Python za ovo ima biblioteku threading, PHP je imao pthreads, a od verzije 8 parallel. PHP ima odvojen interpreter unutar svake niti, ali to zahteva više resursa i ima sporiji start.

Multitasking, višenitna i višeprocesna obrada

- Postoje različiti načini za realizaciju asinhrone obrade događaja u veb programiranju:
  - Kod višeprocesnog programiranja program može da kreira više odvojenih procesa i može se postići istinski paralelizam obrade ako sistem ima više procesora. Ovaj pristup je pogodan za procesiranje procesorsko zahtevnih aplikacija. Python biblioteka je multiprocessing, PHP ima funkciju proc_open i bilioteke za komunikaciono/sinhronizacione primitive (semafori,...) na nivou os. Procesi su mnogo cpu zahtevniji za kreiranje od niti i takođe kod višenitnog i višeprocesnog programiranja mora se voditi računa o uzajamnom isključivanju, deadlocku i svim začkoljicama pravog paralelnog programiranja.
  - Generalno postoje mnoge biblioteke koje su jezički nezavisne a koje olakšavaju asinhrono veb programiranje pokretano događajima, npr. Message brokeri (rabbitmq), itd.

Primer asinhrone aplikacije realizovane koperativnim multitaskingom

- U ovom primeru, python aplikacija koristi kooperativni multitasking koristeći asyncio. Kreiraju se tri taska: A, B i C. Postoji jedan red čekanja implementiran sa asyncio.Queue. Taskovi A i B stavljaju objekte tipa Future u red čekanja, a zatim čekaju da task C odredi vrednosti tim objektima. Task C preuzima Future objekte iz reda čekanja, i zatim im daje vrednosti nakon slučajnog broja sekundi. Tada svi taskovi prestaju sa izvršavanjem.



Primer asinhrone aplikacije realizovane koperativnim multitaskingom

```
import asyncio
import random

async def task_a(q):
    # Create a future object and put it in the queue
    future = asyncio.Future()
    await q.put(future)
    print(f"Request sent from task_a to task_c")
    # Wait for the future to be set with a value by task_c
    value = await future
    # Do something with the value
    print(f"Received value from task_c in task_a: {value}")

async def task_b(q):
    # Create a future object and put it in the queue
    future = asyncio.Future()
    await q.put(future)
    print(f"Request sent from task_b to task_c")
    # Wait for the future to be set with a value by task_c
    value = await future
    # Do something with the value
    print(f"Received value from task_c in task_b: {value}")
   # signalize task c to terminate by sending it None value
    await q.put(None)
```

Primer asinhrone aplikacije realizovane koperativnim multitaskingom

```
async def task_c(q):
    # Wait for future objects in the queue and give them values
    while True:
        future = await q.get()
        if future is None: break
        # Wait for a random number of seconds between 1 and 5
        await asyncio.sleep(random.randint(1, 5))
        # Generate a random value and set it on the future
        value = random.randint(1, 100)
        future.set_result(value)

async def main():
    # Create the queue
    q = asyncio.Queue()
    # Create tasks for task_a, task_b, and task_c
    tasks = [
        task_a(q),
        task_b(q),
        task_c(q)
    ]
    # Wait for all tasks to complete
    await asyncio.gather(*tasks)

# Run the main(), has an implicit event loop (task scheduling)
asyncio.run(main())
```

Primer asinhrone aplikacije realizovane koperativnim multitaskingom

- Async/await model programiranja koristi se u mnogim programskim jezicima za realizaciju korutina, to jest, kooperativnog multitaskinga (ali generalno u okviru jedne niti procesora).
- Sa async se definišu korutine. Dok kod obične funkcije postoji jedna tačka ulaza i jedna tačka izlaza, u korutinu kontrola može ući, izaći i nastaviti izvršavanje u puno različitih tačaka.
- Izvršavanje korutine se može pokrenuti samo iz druge korutine korišćenjem ključne reči await *ime_korutine.* Await ujedno predstavlja prekid izvršavanja korutine i predavanje kontrole drugim korutinama (drugi način je završetak izvršavanja sa return). Kada se uslov na koji await čeka ispuni (npr. vrednost izraćuna) korutina će u nekom trenutku nastaviti izvršavanje od dela iza await.
- Glavni program je isto korutina koja se pokreće sa asyncio.run(main()). Ovime se u “pozadini” tj. Python interpreteru startuje i “petlja događaja” koja raspoređuje kontrolu na korutine i bira koja će sledeća da počne ili nastavi izvršavanje.
- Samo korišćenje imena korutine (npr. u tasks) ne znači da ona počinje da se izvršava (kao kod običnih funkcija), nego se vraća coroutine objekat. U glavnom programu taskovi A,B,C pokreću se konkurentno korišćenjem gather poziva. koji ujedno potom čeka i da se svi taskovi završe.
- Gather kao i druge stvari koje se pokreću sa await, vraća Future, specijalni tip podataka koji predstavlja vrednost koja još nije sračunata. Kada se vrednost sračuna, kod koji je čekao na njega postaje spreman za izvršavanje, čim ga scheduler u petlji događaja izabere za izvršavanje (Analogija sa Observerom u Javi).

Primer asinhrone aplikacije realizovane koperativnim multitaskingom - zaključak

- Ovo je samo jednostavan primer, ali pokazuje kako možemo dizajnirati sistem oko događaja. Ovaj primer ima monolitnu arhitekturu, ali EDA se vrlo uspešno primenjuje u mikroservisnim arhitekturama kada je u pitanju multiprocesna realizacija.
- U aplikaciji u stvarnom svetu može biti mnogo više događaja i rukovalaca događajima. Aplikacija takođe može imati više izvora događaja, kao što su veb front-end, mobilna aplikacija ili neki back end servisi.
- Korišćenjem pristupa zasnovanog na događajima, možemo da upravljamo svim ovim događajima i izvorima na skalabilan i fleksibilan način, bez potrebe da čvrsto povezujemo različite delove sistema. Takođe možemo da izgradimo dodatne funkcije i usluge na bazi sistema događaja, kao što su analiza u realnom vremenu, revizija ili obaveštenja.
- Obrazac arhitekture vođene događajima je moćan alat za izgradnju složenih sistema koji zahtevaju obradu događaja u realnom vremenu i može nam pomoći da izbegnemo neke od izazova tradicionalnih request-response arhitektura.

UZORAK INVERZIJA KONTROLE

Inversion of Control (IoC)

- *To je obrazac dizajna koji ima za cilj da invertuje kontrolu nad objektima i njihovim zavisnostima. Umesto da objekti stvaraju svoje zavisnosti (objekte od kojih zavise), mi ubrizgavamo zavisnosti u objekte (engl. Dependency Injection). Ovo omogućava fleksibilniji kod pogodniji za testiranje.*

Primer

- Greeter klasa će koristiti NameService da dobije ime osobe koju želi da pozdravi. U ovom primeru, klasa NameService ima jednostavnu metodu za vraćanje imena “Pera".

```
class NameService:
    def get_nam"e(self) -> str:
        return "Pera"

class Greeter:
    def __init__(self):
        self.name_service = NameService()

    def greet(self) -> str:
        name = self.name_service.get_name()
        return f"Hello, {name}!"
```

- Ako jednostavno instanciramo klasu NameService u konstruktoru Greeter, Greeter zavisi od NameService i mora da zna kako da konfiguriše ovu klasu. Narušen je princip inverzije zavisnosti (soliD).

Ubacivanje zavisnosti

- Ako želimo da odvojimo Greeter od NameService-a da bismo npr. koristili neku drugu klasu sa istim metodom get_name, možemo promeniti Greeter klasu tako da ima konstruktor koji prihvata NameService objekat:

```
class NameService:
    def get_name(self) -> str:
        return "Pera"


class Greeter:
    def __init__(self, name_service: NameService):
        self.name_service = name_service

    def greet(self) -> str:
        name = self.name_service.get_name()
        return f"Hello, {name}!"
```

Instanciranje objekata

- Sada postaje komplikovanije, jer više nije dovoljno samo greeter = Greeter() nego je potrebno pamtiti sve zavisnosti klase.
- Rešenje je da se konfigurisanje objekata izvuče na jedno centralizovano mesto, takozvani kontejner sa inverzijom kontrole (inversion of control container).
- Ideja je da se "obrne kontrola" u sistemu uklanjanjem kontrole nad konfigurisanjem iz samih objekata i sprovoditi je u potpunosti odvojeno od njih.
- Definiše se jedna klasa (IOC Container) koja vodi računa o svim zavisnostima među svim drugim klasama.

Kontejner sa inverzijom kontrole

- Koristimo biblioteku dependency_injector da ubacimo objekat NameService u klasu Greeter. (pip install dependency_injector)
- Definišemo klasu kontejnera koja deklariše objekat NameService kao Singleton i Greeter objekat kao Factory koja koristi objekat NameService. Singletor i Factory određuju način instanciranja odgovarajućih objekata.
- Zatim kreiramo instancu kontejnera i koristimo je za kreiranje Greeter objekta.
- Kada pozovemo greet() metod Greeter objekta, on će koristiti NameService objekat ubačen u njega od strane kontejnera.

```
from dependency_injector import containers, providers

class Container(containers.DeclarativeContainer):
    name_service = providers.Singleton(NameService)
    greeter = providers.Factory(Greeter, name_service)

container = Container()

greeter = container.greeter()
print(greeter.greet())
```

Kontejner sa inverzijom kontrole

- **Provajderi** kreiraju objekte i ubrizgavaju im potrebne zavisnosti, to jest, konfigurišu ih. Odgovorni su za životni ciklus objekata.
- **Factory provajder** kreira nove instance objekta pri svakom pozivu (pogodan za objekte kratkog životnog ciklusa).
- **Singleton provajder** obezbeđuje jedan objekat. On pamti prvi kreirani objekat i vraća ga kod ostalih poziva.
- **Kontejneri** su kolekcije provajdera. **DeclarativeContainer** je stil definicije kontejnera zasnovan na klasi. Korisnik kreira podklasu deklarativnog kontejnera, stavlja provajdere kao atribute i kreira instancu kontejnera.
- Registracija zavisnosti se obavlja dodavanjem atributa u deklarativni kontejner. Svaki atribut je neki provajder ili drugi deklarativni kontejner.
- Razrešavanje zavisnosti (engl. resolution) obavlja se kada se pristupi atributu deklarativnog kontejnera. Tada se poziva odgovarajući provajder da obezbedi instancu klase za taj provajder odgovara.
- Ubacivanje zavisnosti u ovom primeru ide kroz konstruktor objekta kada ga odgovarajući provajder instancira

OBRAZAC DIZAJN VOĐEN DOMENOM

Domain Driven Design

- Dizajn vođen domenom (DDD) je obrazac dizajna koji se fokusira na izgradnju softvera koji odražava domen iz stvarnog sveta kojem služi. To uključuje razumevanje domena i njegovo modelovanje u kodu, stvaranje zajedničkog jezika između programera i zainteresovanih strana i korišćenje tog modela za pokretanje dizajna softvera.
- Obrazac se dakle odnosi kako na strukturu same aplikacije, tako i na metodologiju projektovanja aplikacije
- Referenca: Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, ISBN: 0321125215
- Preporučeni sažetak ove knjige: https://www.infoq.com/minibooks/domain-driven-design-quickly/

Primer koji ilustruje DDD obrazac

- Pretpostavimo da projektujemo jednostavnu aplikaciju za e-trgovinu koja omogućava korisnicima da kupuju i prodaju proizvode.
- Prvi korak u primeni DDD je da se identifikuju osnovni koncepti domena, kao što su proizvodi, kupci, porudžbine i plaćanja.
- Zatim možemo kreirati klase koje modeliraju ove koncepte, zajedno sa njihovim atributima i ponašanjima.

```
class Order:
    def __init__(self, product_name, quantity, total):
        self.name = name
        self.quantity = quantity
        self.total = total

class Customer:
    def __init__(self, name: str):
        self.name = name
        self.orders = []

    def place_order(self, product_name, quantity, order_total):
        self.orders.append(Order(product_name, quantity, order_total))
```

Primer koji ilustruje DDD obrazac

- Na primer, možemo imati klasu proizvoda sa atributima kao što su naziv, opis, cena i količina, kao i metode kao što su get_price i decrement_quantity.

```
class Product:
    def __init__(self, name, description, price, quantity):
        self.name = name
        self.description = description
        self.price = price
        self.quantity = quantity

    def get_price(self):
        return self.price

    def decrement_quantity(self, amount):
        self.quantity -= amount
```

Primer koji ilustruje DDD obrazac

- Zatim možemo kreirati repozitorijume za upravljanje perzistencijom i dohvatanjem ovih objekata. Na primer, imamo ProductRepository koji može da dodaje, uklanja i preuzima proizvode iz baze podataka. Isto i za mušterije.

```
class ProductRepository:
 def __init__(self):
     self.products = []

 def add_product(self, product):
     self.products.append(product,

 def remove_product(self, product
     self.products.remove(product,

 def get_product_by_name(self, na
     for product in self.products
         if product.name == name:
             return product
     return None
```

```
class CustomerRepository:
  def __init__(self):
      self.customers = []

  def add_customer(self, customer):
      self.customers.append(customer)

  def remove_customer(self, customer):
      self.customers.remove(customer)

  def get_customer_by_name(self, name):
      for customer in self.customers:
          if customer.name == name:
              return customer
      return None
```

Primer koji ilustruje DDD obrazac

- Takođe možemo definisati servise koji orkestriraju interakcije između ovih objekata. Na primer, možemo da imamo OrderService koji omogućava klijentima da naručuju proizvode:

```
class OrderService:
    def __init__(self, product_repository):
        self.product_repository = product_repository

    def place_order(self, customer, product_name, quantity):
        product= self.product_repository.get_product_by_name(product_name)
        if product is None:
          raise ValueError(f"No product with name '{product_name}' found")
        if product.quantity < quantity:
          raise ValueError(f"Only {product.quantity} units of ’
                             {product_name}' available")
        order_total = product.get_price() * quantity
        customer.place_order(product, quantity, order_total)
        product.decrement_quantity(quantity)
```

Primer koji ilustruje DDD obrazac

```
product_repository = ProductRepository()
product_repository.add_product(Product("Apple", "A juicy red fruit", 1.99, 10))
product_repository.add_product(Product("Banana", "A yellow fruit", 0.99, 5))

customer_repository = CustomerRepository()
customer_repository.add_customer(Customer("Pera"))
customer_repository.add_customer(Customer("Mika"))

order_service = OrderService(product_repository)

print("Welcome to the e-commerce application")

while True:
    name = input("What is your name? ")
    customer = customer_repository.get_customer_by_name(name)
    if customer : break

while True:
    print("Available products:")
    for product in product_repository.products:
        print(f"- {product.name}: {product.description} ({product.quantity}
available) for ${product.price}")
    product_name = input("What product would you like to buy? ")
    quantity = int(input("How many units would you like to buy? "))
    try:
        order_service.place_order(customer, product_name, quantity)
        print(f"Order placed for {quantity} units of {product_name}")
    except ValueError as e:
        print(f"Error: {e}")
```

Konačno, sloj korisničkog interfejsa (konzolna aplikacija u ovom primeru) koji omogućava korisnicima interakciju sa aplikacijom.

Primer koji ilustruje DDD obrazac

- U ovom primeru, koristili smo dizajn vođen domenom da modelujemo domen e-trgovine na način koji je razumljiv i kojim se može upravljati.
- Klase Product, Customer i Order predstavljaju osnovne koncepte domena, dok klase ProductRepositoriy i CustomerRepository upravljaju njihovom perzistencijom.
- Klasa OrderService obezbeđuje poslovnu logiku za postavljanje porudžbine, koja uključuje proveru dostupnosti proizvoda i smanjenje njegove količine ako je dostupan.
- Konačno, konzolna aplikacija obezbeđuje korisnički interfejs za klijente za interakciju sa aplikacijom, uz pomoć klase Customer.
- Fokusirajući se na osnovne koncepte domena, možemo kreirati zajednički jezik između programera i zainteresovanih strana i koristiti taj jezik za pokretanje dizajna softvera. Naravno, ovo je samo jednostavan primer, a aplikacije u stvarnom svetu uključuju složenije domene i poslovnu logiku. Međutim, principi dizajna vođenog domenom mogu se primeniti na bilo koji domen i mogu pomoći u stvaranju softvera koji je razumljiv i pogodan za održavanje.