Projektovanje web aplikacija

UVOD

Arhitektura softvera

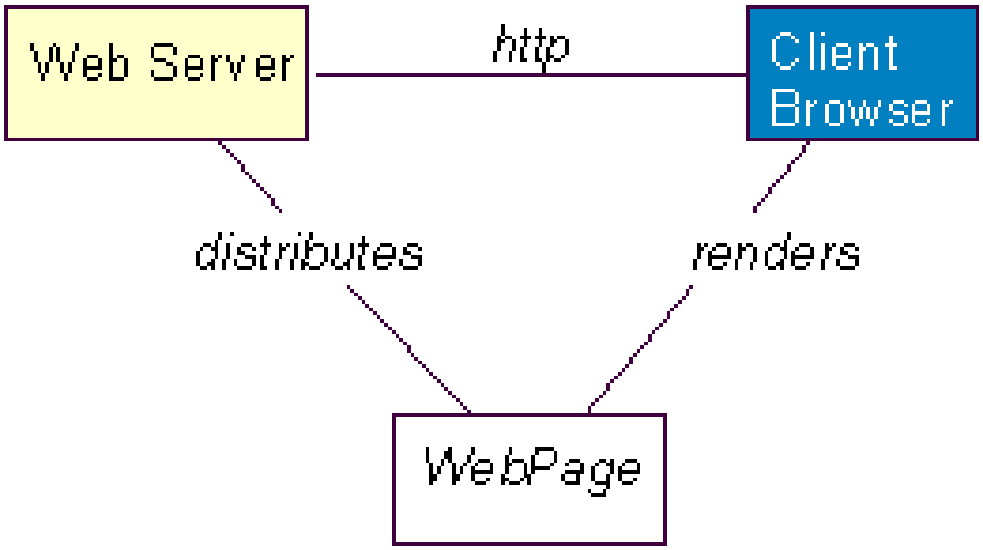
Arhitektura softvera:

- Glavne komponente sistema i način na koji međusobno interaguju;
- Predstavlja deljeno razumevanje dizajna sistema od strane programera
- Arhitektonske odluke u projektovanju sistema: (važne) odluke koje treba doneti što pre tokom razvoja sistema i koje je teško kasnije izmeniti

Arhitektura web aplikacije

Web sajt

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



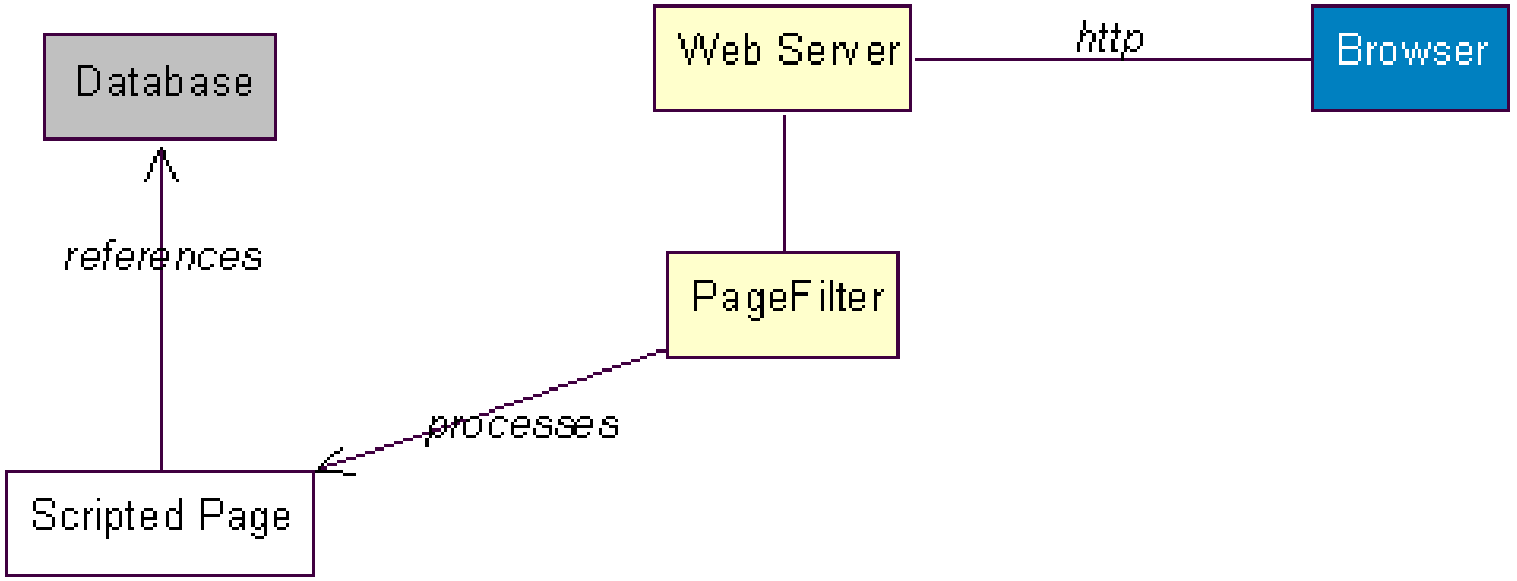
Arhitektura web aplikacije

- Informacije koje pruža web sajt tipično su zapamćene, u formatiranom obliku, u fajlovima.
- Klijent zahteva fajl po imenu i kada je neophodno pruža i informaciju o njegovoj celokupnoj putanji (adresi) . Ovi fajlovi se nazivaju stranicama i reprezentuju sadržaj web sajta.

Arhitektura web aplikacije

Dinamički web 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.



Arhitektura web aplikacije

Skriptovanje na serverskoj strani

- Krajnji rezultat serverskog procesiranja je:
- ažuriranje stanja na serveru (baza podataka, poslovni objekti), i
- priprema stranice formatirane u HTML-u(korisničkog interfejsa) za klijentski browser koji je postavio zahtev.
- Programski jezici/tehnologije: C/CGI, php, jsp, asp, Python, Ruby,…

Skriptovanje na klijentskoj strani

- Klijentski browser takođe može da izvršava programski skript na stranici (na javascript-u).
- Međutim, kada browser izvršava takav skript, on nema neposredan pristup serverskim resursima (baza podataka,...). Skriptovi koji se izvršavaju na klijentu tipično dopunjavaju izgled i ponašanje korisničkog interfejsa, a ne implementiraju poslovnu logiku aplikacije.

Arhitektura web aplikacije

Forme

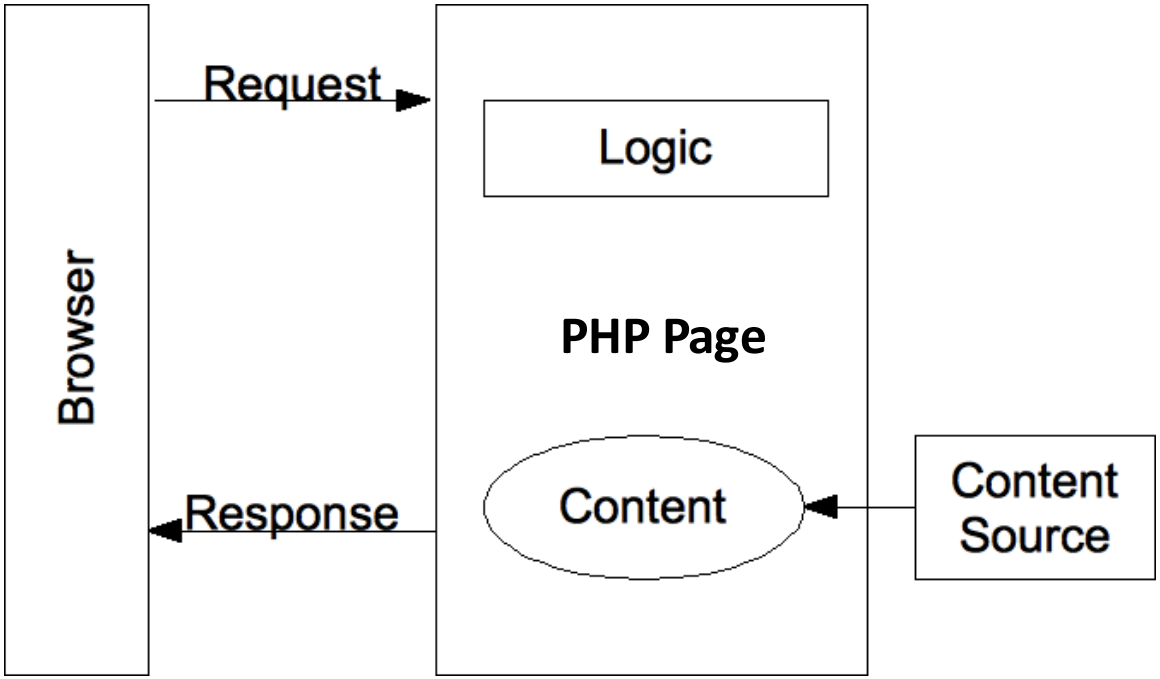
- Najviše korišćeni mehanizam za sakupljanje ulaznih podataka od korisnika je putem HTML formi.
- HTML forma je kolekcija ulaznih polja koji se prikazuju (render-uju) kao web stranica. Osnovni ulazni elementi su: tekstualno polje (text box), tekstualna oblast (text area), polje za čekiranje (checkbox), 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.
- 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.

Model 1 i model 2

- Termin definisan u Java zajednici, ali primenljiv i na ostale
- Aplikacije modela 1 su proceduralne aplikacije, lakše za izradu malih web sajtova
- Aplikacije modela 2 su aplikacije izrađene od više slojeva i/ili koje poštuju princip projektovanja poznat pod nazivom razdvajanje zaduženja (ili nadležnosti) (separation of concerns).
- Zaduženja su različiti aspekti funkcionalnosti softvera. Na primer, "poslovna logika" softvera je jedno, a interfejs kroz koji osoba koristi ovu logiku je nešto drugo.
- Razdvajanje zaduženja zahteva odvojen kod za svako od ovih zaduženja. Promenaa interfejsa ne treba da zahteva promenu poslovne logike i obrnuto.

Model 1

- U modelu 1, zahtev se prosleđuje PHP modulu i onda taj modul obrađuje sve vezano za zahtev, uključujući obradu zahteva, potvrđivanje podataka, izvršavanje poslovne logike i generisanje odgovora. Model 1 arhitektura se najčešće koristi u manjim, jednostavnim aplikacijama zbog svoje lakoće razvoja.

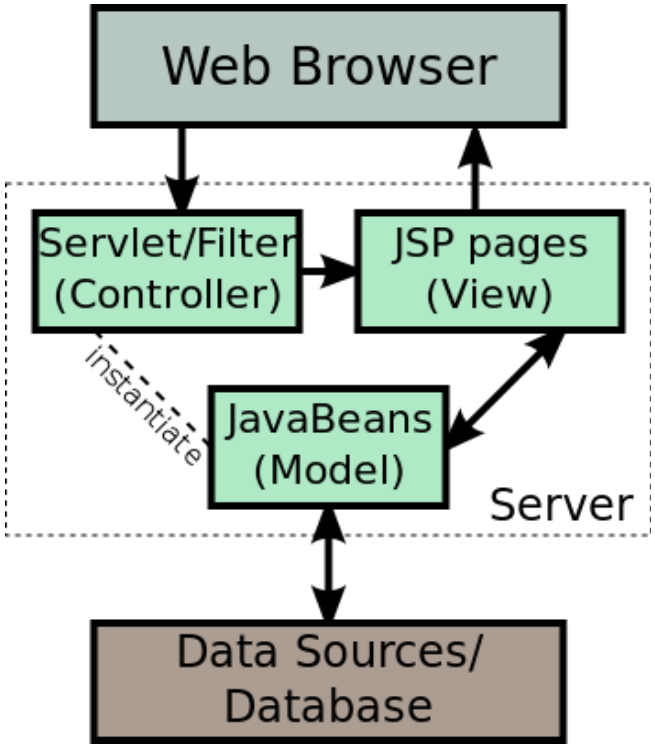


Model 1

- Iako konceptualno jednostavna, ova arhitektura nije pogodna za razvoj velikih aplikacija, jer se, neizbežno, veliki deo funkcionalnosti duplira u svakoj PHP strani.
- Takođe, Model 1 arhitektura nepotrebno povezuje poslovnu logiku i logiku prezentacije zahteva. Preplitanje poslovne logike sa prezentacijom čini teškim uvođenje novih "pogleda" ili pristupnih tačaka u aplikaciji.
  - Na primer, ako želimo da pored HTML prikaza aplikacija obezbedi i RESTFul servise.

Model 2

- Model 2 je zasnovan na projektnom obrascu koji razdvaja prikaz sadržaja od logike za dobijanje i manipulaciju sadržajem. Za razdvajanje logike i prikaza, najčešće se koristi Model-View-Controller (MVC) šablon.
- U modelu 2 aplikacije, zahtevi od klijentskog pretraživača se prosleđuju kontroleru.Kontroler obavlja svu potrebnu logiku da dobije tačan sadržaj za prikaz. Kontroler od tog sadržaja pravi zahtev (obično u obliku objekta) i odlučuje kom pogledu će proslediti taj zahtev. Pogled zatim prikazuje sadržaj dobijen od strane kontrolora.
- Model 2 preporučuje se za srednje i velike aplikacije.

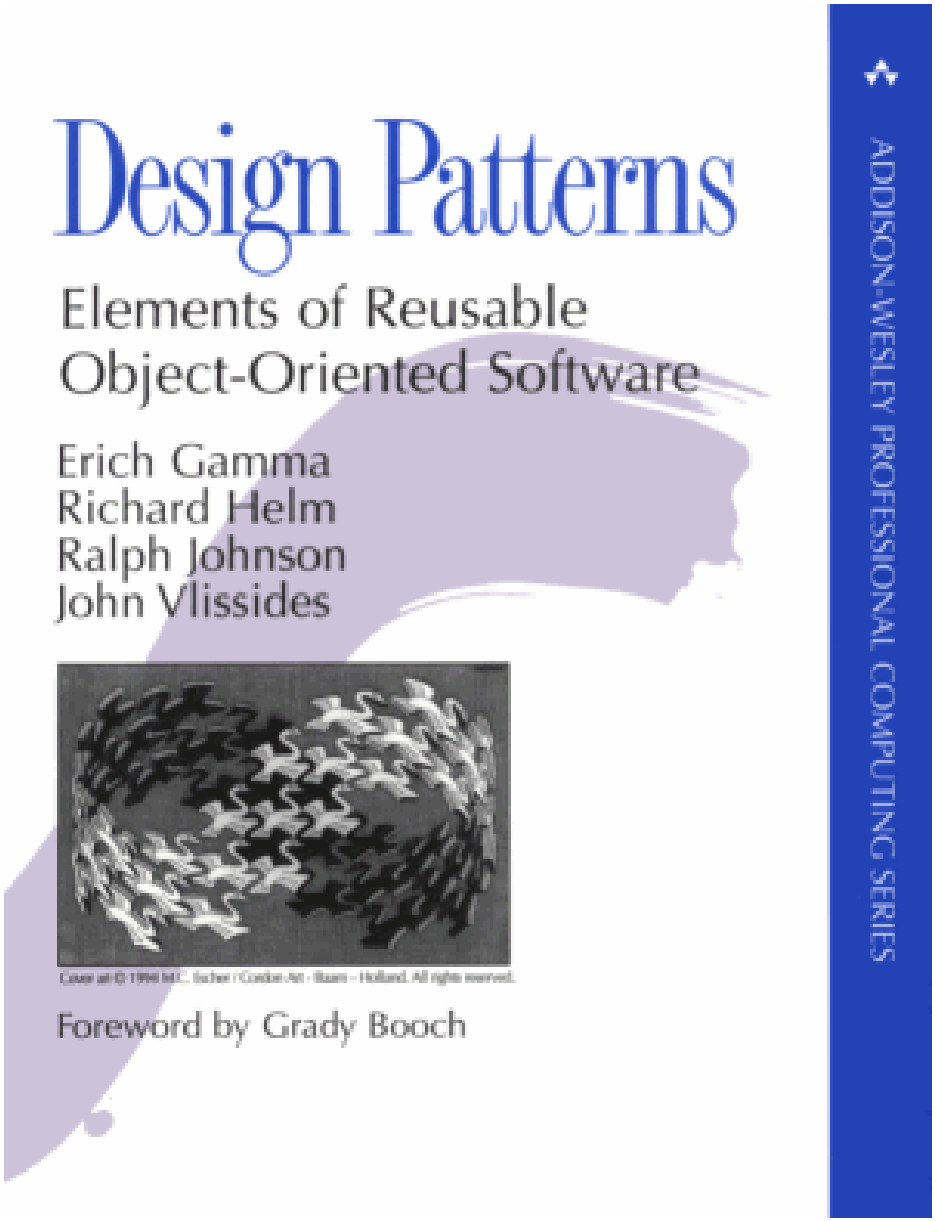


PROJEKTNI UZORCI

Projektni uzorci (Design patterns)

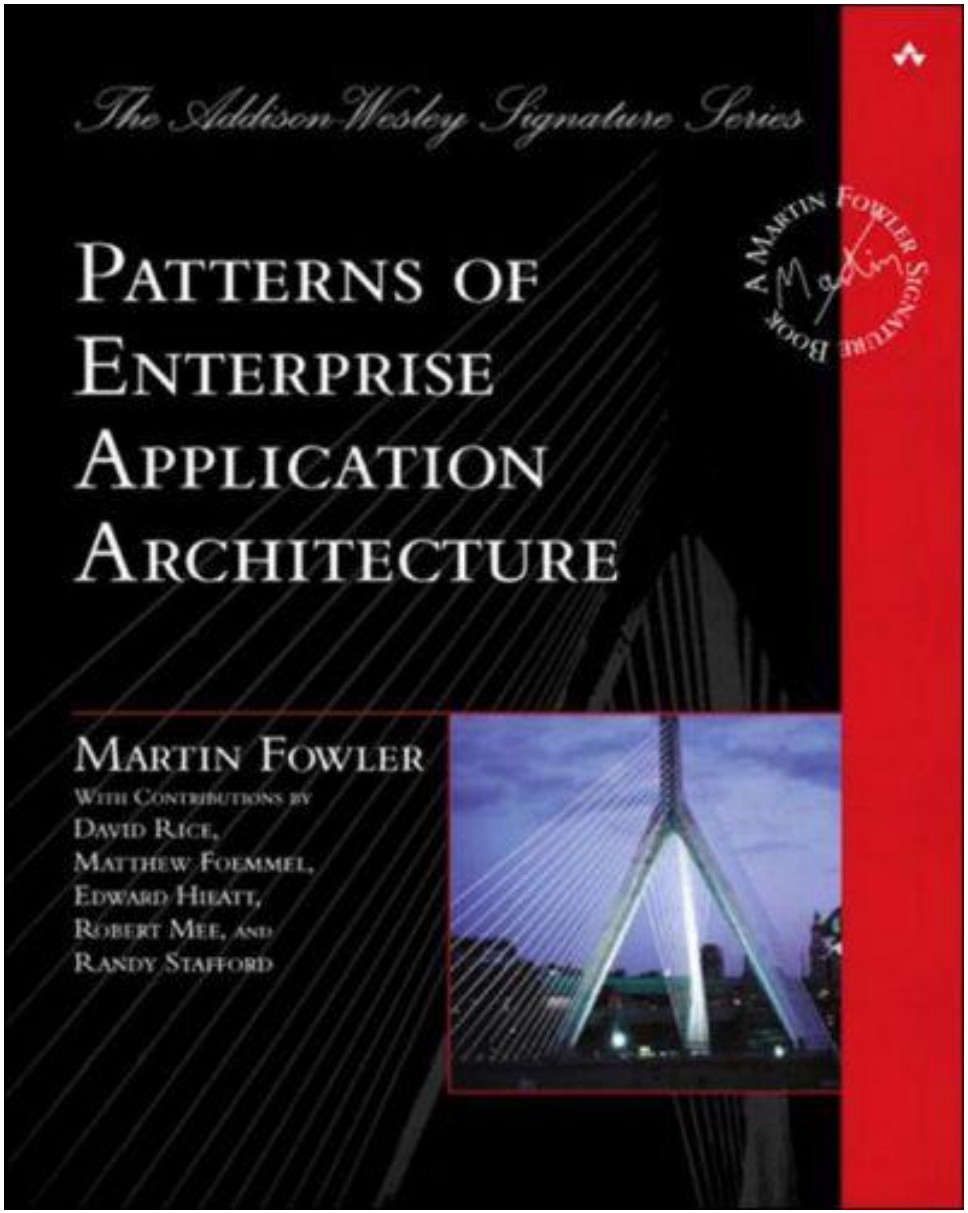
- *"Svaki uzorak opisuje **problem** koji se javlja iznova i iznova u našem okruženju, a zatim opisuje **suštinu rešenja** tog problema, na takav način da se takvo rešenje može **koristiti ponovo** milion puta, a da se nikada to ne uradi dva puta na isti način“*
  - Christopher Alexander, poznati arhitekta (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

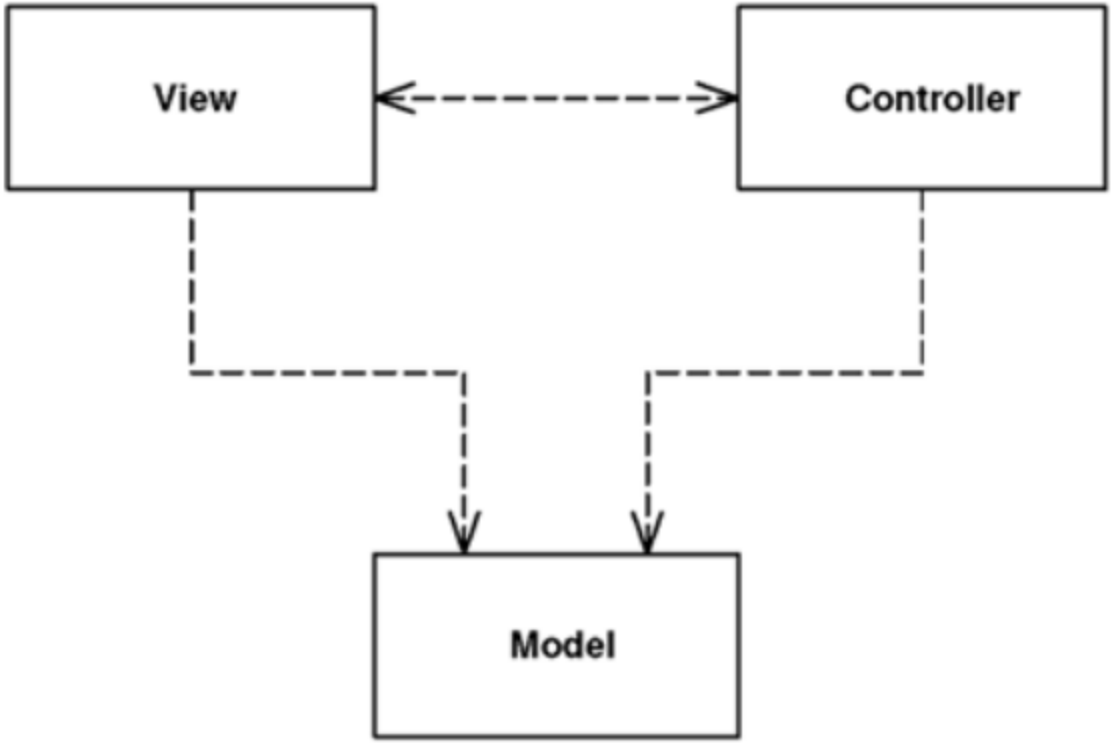


- PoEAA patterns
  - Razmotrićemo samo neke uzorke iz ove knjige, koji se često koriste u web programiranju
  - Za razliku od GoF uzoraka, koji su uglavnom na nivou detaljnog projektovanja, PoEAA se odnosi na nivo arhitekture

MODEL VIEW CONTROLLER

Model View Controller

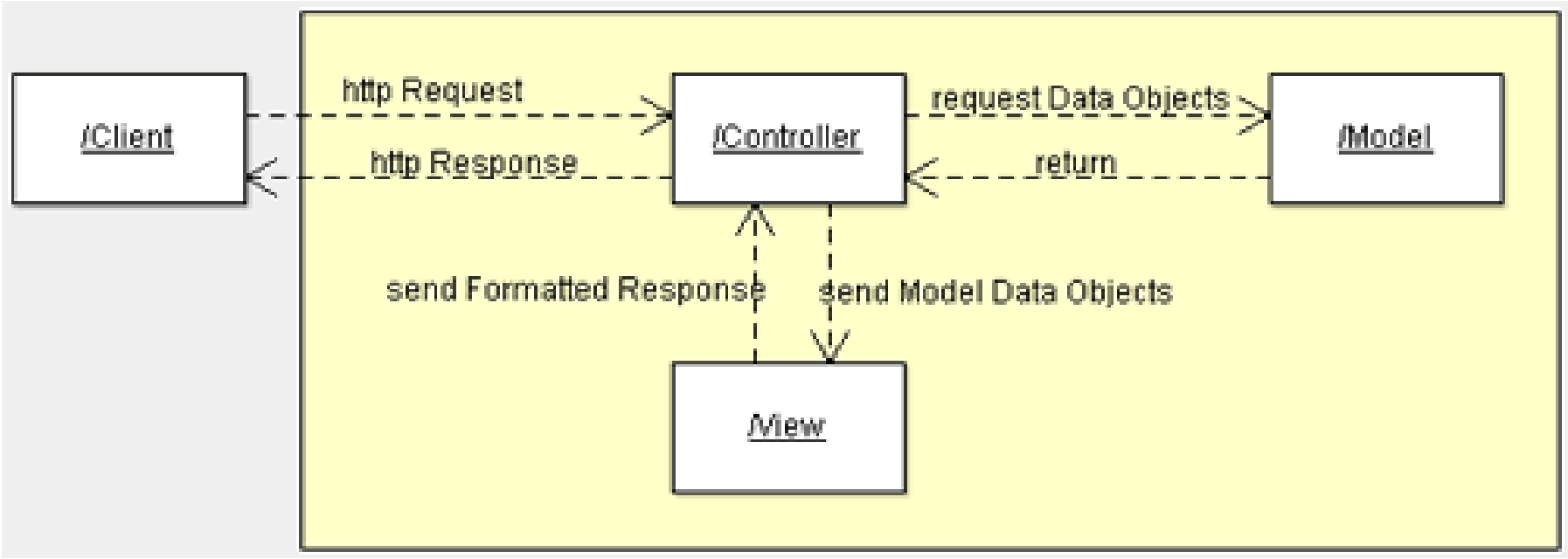
- *Deli interakciju sa korisničkim interfejsom u tri različite uloge.* Model prikaz kontroler je najviše korišćen obrazac za savremene web aplikacije. Korišćen je prvi put u Smalltalk-u kasnih 1970-ih, a zatim usvojen i popularizovan u Javi. Trenutno postoji više od desetak PHP web okvira na osnovu MVC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



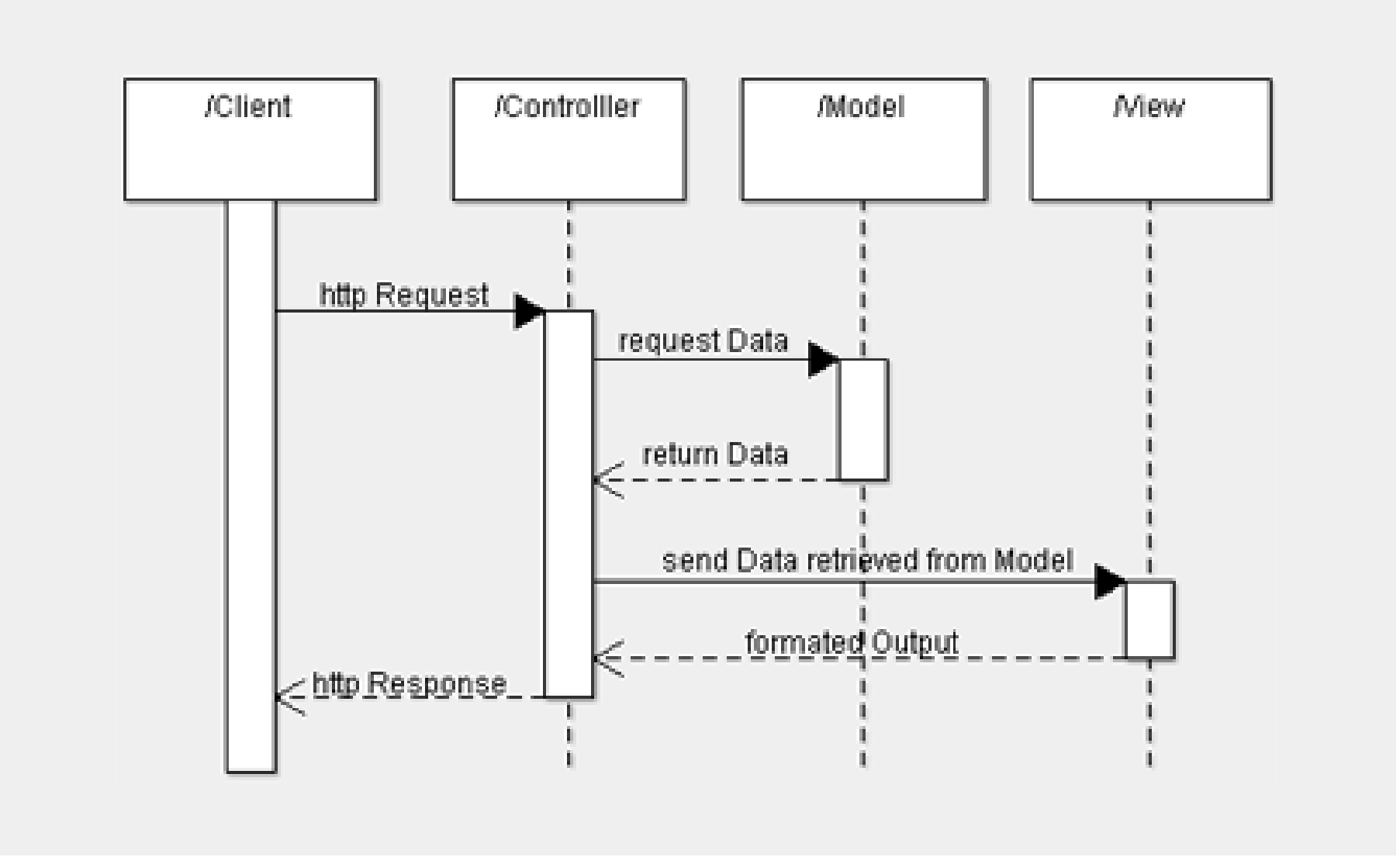
Model View Controller

- 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

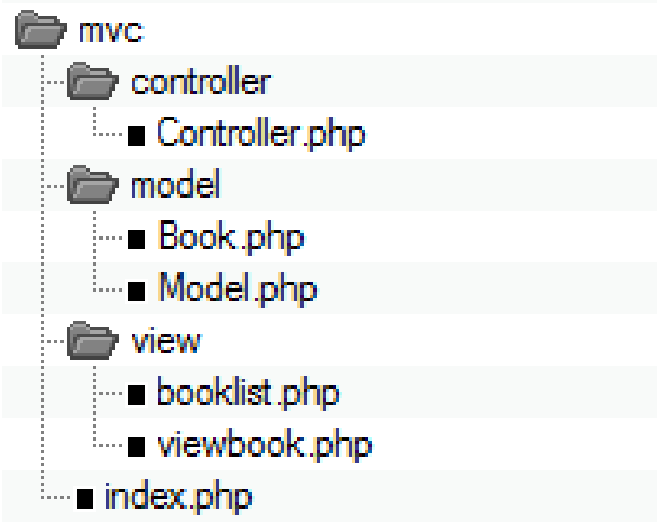


Dijagram sekvence MVC



Jednostavna PHP implementacija MVC

- Php primer ima jednostavnu strukturu, stavljajući svaku MVC komponentu u poseban folder:



Ulazna tačka aplikacije je index.php. Fajl index php delegiraće sve zahteve kontroleru:

```
// index.php file
   include_once("controller/Controller.php");

   $controller = new Controller();
   $controller->invoke();
```

Klasa Kontroler

- Konstruktor instancira klasu model i kada stigne zahtev od klijenta, kontroler odlučuje koji podaci se zahtevaju od modela. Onda poziva klasu model da preuzme te podatke. Nakon toga poziva odgovarajući prikaz prosleđujući mu podatke koji dolaze iz modela. Kod je izuzetno jednostavan. Kontroler ne zna ništa o bazi podataka ili o tome kako se stranica generiše.

Kontroler

```
include_once("model/Model.php");

  class Controller {
      public $model;

      public function __construct() {
          $this->model = new Model();
      }

      public function invoke() {
          if (!isset($_GET['book'])) {
              // no special book is requested, we'll show a list of all available books
              $books = $this->model->getBookList();
              include 'view/booklist.php';
          }
          else  {
              // show the requested book
              $book = $this->model->getBook($_GET['book']);
              include 'view/viewbook.php';
          }
      }
  }
```

Model

- Model predstavlja podatke i logiku aplikacije, što mnogi zovu poslovna logika. Odgovoran je za:

- skladištenje, brisanje, ažuriranje podataka aplikacije. Generalno to uključuje rad sa bazom podataka, ali je zamislivo i sprovođenje istih operacija pozivanjem spoljnih web servisa ili API-ja.

- enkapsulira logiku aplikacije. To je sloj koji treba da implementira svu logiku aplikacije. Najčešće greške su da se implementiraju operacije aplikativne logike unutar kontrolera ili prikaznog sloja.

Klase Model i Book

- U našem primeru model predstavljaju dve klase: "Model“ i “Book".
- “Book" klasa je entitetska. Ova klasa treba da bude izložena sloju Prikaza i predstavlja relevantan pogled na Model.
- U dobroj realizaciji MVC obrasca samo entitetske klase treba izlagati iz modela i one ne bi uopšte trebalo da sadrže poslovnu logiku.
- Njihova svrha je samo da čuvaju podatke.
- U zavisnosti od implementacije entitetski objekat može biti zamenjen podatkom u XML ili JSON formatu.
- U našem primeru možete primetiti kako model vraća ili određenu knjigu, ili listu svih dostupnih knjiga

Klasa Model

```
include_once("model/Book.php");

  class Model {
      public function getBookList() {
          // here goes some hardcoded values to simulate the database
          return array(
              "Jungle Bok" => new Book("Jungle Book", "R. Kipling", "A classic."),
              "Moonwalker" => new Book("Moonwalker", "J. Walker", ""),
              "PHP for Dummies" => new Book("PHP for Dummies", “N.N.", "")
          );
      }

      public function getBook($title) {
          // first get all the books and then we return the requested one.
          // in a real life scenario this will be done through a db select command
          $allBooks = $this->getBookList();
          return $allBooks[$title];
      }
  }
```

Entitetska klasa Book

```
class Book {
    public $title;
    public $author;
    public $description;

    public function __construct($title, $author, $description) {
        $this->title = $title;
        $this->author = $author;
        $this->description = $description;
    }
}
```

Implementacija Prikaza

- Prikaz (prezentacioni sloj) je odgovoran za formatiranje podatake dobijene iz modela u obliku koji je pristupačan korisniku. Podaci mogu doći u različitim formatima od modela: jednostavni objekti, xml strukture, JSON, ...
- Kontroler delegira podatke iz modela do određenog elementa prikaza, obično povezan sa glavnim entitetom u modelu.
- U našem primeru pogled sadrži samo dva fajla, jedan za prikazivanje jedne knjige, a drugi za prikazivanje liste knjiga.

Moduli viewbook.php i booklist.php

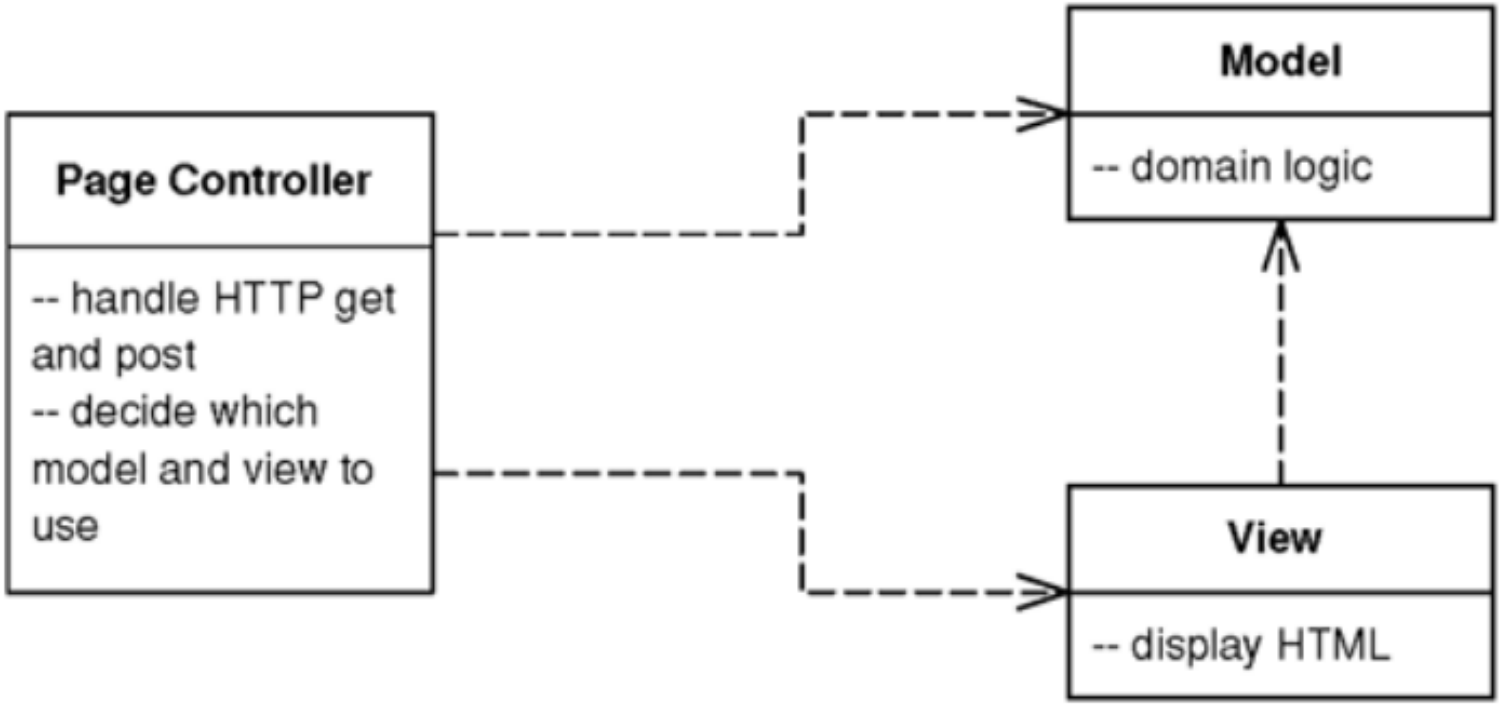
```
<html>
  <head></head>
<body>
<?php
      echo 'Title:' . $book->title . '<br/>';
      echo 'Author:' . $book->author .
    '<br/>';
        echo 'Description:' . $book->
    description . '<br/>';
?>
</body>
</html>
```

```
<html>
  <head></head>
  <body>
    <table>
     <tbody>
    <tr><td>Title</td>
   <td>Author</td>
    <td>Description</td>
    </tr></tbody>
      <?php
         foreach ($books as $title => $book)  {
            echo '<tr><td>
             <a href="index.php?book=‘ .
              $book->title.'">'.$book->title.
              '</a></td><td>'.$book->author.
               '</td><td>'.
               $book->description.'</td></tr>';
         }
      ?>
    </table>
  </body>
  </html>
```

UZORAK PAGE CONTROLLER

Kontroler stranice (Page Controller)

- Objekat koji obrađuje zahtev za određenom stranicom ili akcijom na Web sajtu.



Kako radi Page Controller

- Osnovna ideja iza kontrolera stranice je da imamo po jedan modul na Web serveru kao kontroler za svaku stranu na sajtu, preciznije, kontrolori se vezuju za svaku akciju, koja se može kliknuti na link ili dugme.
- Osnovne odgovornosti stranice kontrolera su:
  - Dekodirati URL i izdvojiti sve podatke iz formi da se sakupe svi podaci za akciju.
  - Kreiranje i pozivanje svih objekata modela radi obrade podataka. Svi relevantni podaci iz HTML zahteva treba da budu prosleđeni modelu, tako da objekti ne treba nikakvu vezu sa HTML zahtevom.
  - Određivanje koji pogled treba da prikaže stranicu rezultata i prosleđivanje informacija modela tom pogledu.

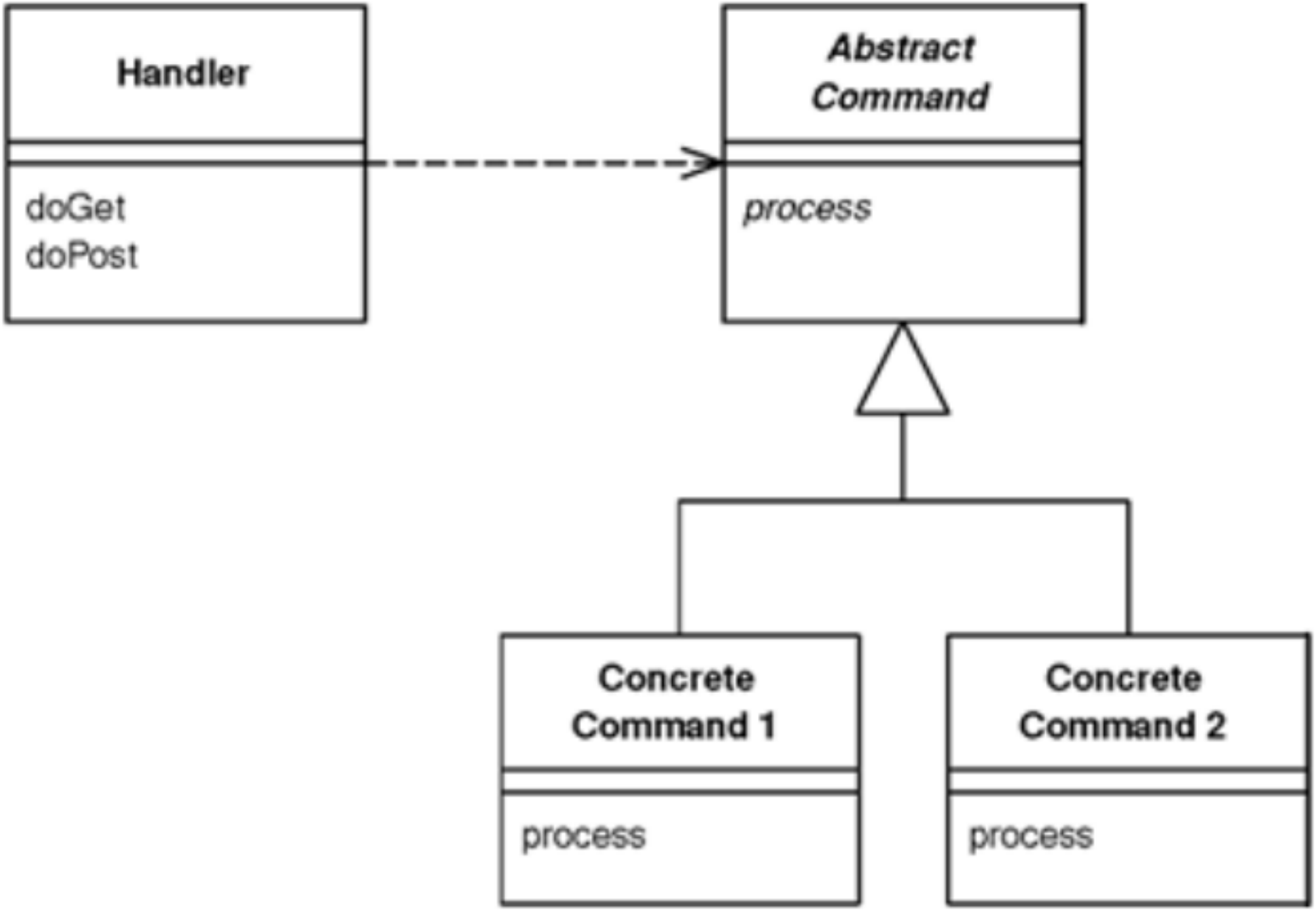
Kada koristiti Page Controller

- Ključna odluka poenta je da li da koristite kontroler stranice ili Front Controller
- Page kontroler je mnogo manje složen nego Front kontroler i radi posebno dobro u situacijama gde je većina kontrolera logike prilično jednostavna.

UZORAK FRONT CONTROLLER

Front Controller

- *Kontroller koji obrađuje sve zahteve Web sajta*

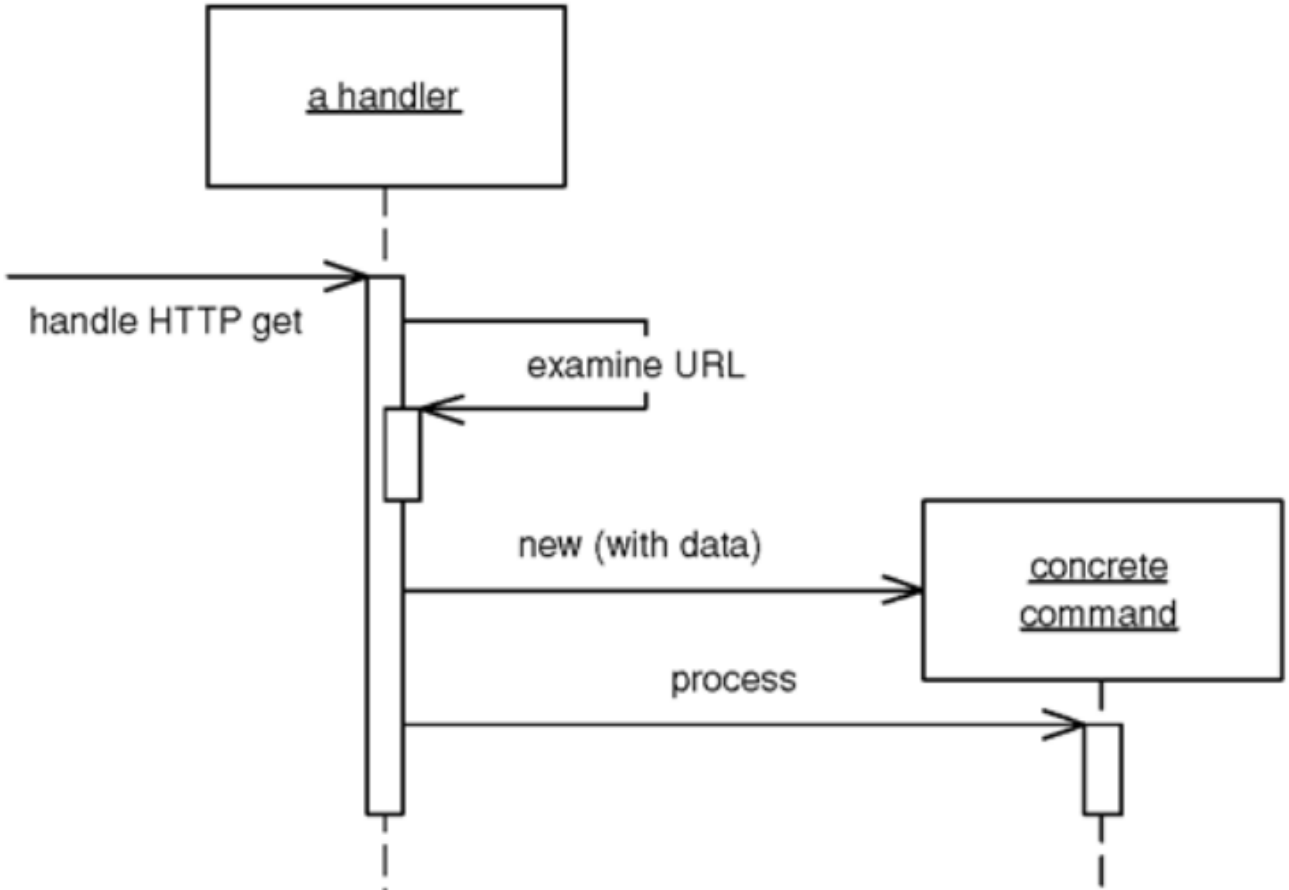


Namena Front Controllera

- U složenom Veb lokaciji postoji mnogo sličnih stvari koje treba da se urade prilikom obrade zahteva. Ove stvari uključuju bezbednost, internacionalizacije, kao i pružanje posebnih prikaza za pojedine korisnike. Ako je ponašanje ulaznog kontrolera raštrkano u više objekata, veliki deo ovog ponašanja može završiti dupliran. Takođe, teško je promeniti ponašanje u vreme izvršavanja.
- Prednji kontroler objedinjuje obradu svih zahteva usmeravanjem zahteva preko jednog jedinog objekta rukovaoca.
- Ovaj objekat može da izvrši zajedničko ponašanje, koje se može menjati u runtime dekoraterima. Rukovalac onda prosleđuje do komandnog objekta za ponašanje prema određenom zahtevu.

Kako radi Front Controller

- Prednji kontroler upravlja svim pozivima ka Web sajtu i obično je strukturiran u dva dela: Web handler i komandna hijerarhija. Web rukovalac je objekat koji zapravo prima post ili get zahteve od Web servera. On povlači samo toliko informacija iz URL i zahteva da odluči kakvu akciju da pokrene, a zatim delegira komandi da sprovede akciju.



Kako radi front controller

- Web hendler može da odluči koju komandu da pokrene bilo statički ili dinamički.Statička verzija uključuje raščlanjivanje URLa i korišćenje uslovne logike, dinamička verzija obično podrazumeva uzimanje standardni deo URLu i korišćenje dinamičkog instanciranja da stvori klasu komande.
- Dinamički slučaj omogućava dodavanje novih komandi bez menjanja Web handlera.
- Ovaj obrazac može se povezati sa Dekoratorom (GOF obrazac) i Filterom presretačem (presretanje Filter) (J2EE obrazac) da se obavi pred-i-postprocesiranje zahteva (autentifikacija, pravljenje loga, indentifikacija lokalnih podešavanja).

Kada koristiti Front Controller

- Prednost Front kontrolera je da omogućava da se izdvoji kod koji se inače duplira na strani kontrolera.
- Aplikacija se lakše prebacuje na drugi web server jer je jedino Front kontroler komponenta koja treba da se podesi da prima zahteve od klijenta.

Jednostavna statička implementacija FCa

```
< ?php
//A file with all the common configuration, like database
require_once('lib/config.php')

//Your header, the same regardless of the page
require_once('parts/header.php')

//This is the content area that will change based on the URL.
$url = substr($_SERVER['REQUEST_URI'], strlen(URLROOT));
switch($url){
      case (''): require_once('pages/root.php'); break;
      case ('foo'): require_once('pages/foo.php'); break;
      case ('bar'): require_once('pages/bar.php'); break;
      default: //the default is an error!
            require_once('pages/error.php'); break;
 }

//Your footer, the same regardless of the page
require_once('parts/footer.php')
?>
```

Dinamička implementacija FCa

```
class Controller {
    private function __construct() {}
    static function run() {
        $instance = new Controller();
        $instance->handleRequest();
    }
    function handleRequest() {
        $request = new Request(); // objekat
        $cmd_r = new CommandResolver();
        $cmd = $cmd_r->getCommand();
        $cmd->execute( $_REQUEST);
    }
}
```

U index.php fajlu aplikacije treba da stoji samo:

```
require( "Controller.php" );
Controller::run();
```

Dinamička implementacija FCa

- CommandResolver koristi Refleksiju da učita i instancira odgovarajuću klasu čije ime saznaje iz dobijenog URIja, čime izbegava switch konstrukciju (skica rešenja):

```
getCommand() {
    $cmd= substr($_SERVER['REQUEST_URI'], strlen(URLROOT));
    $filepath = “commands/“ . $cmd . “.php”;
    $classname = “{$cmd}";
    @require_once("$filepath");
    $cmd_class = new ReflectionClass($classname);
    return $cmd_class->newInstance();
}
```

Dinamička implementacija FCa

```
abstract class Command {

    final public function execute(){
        $this->doExecute();
    }

    abstract public function doExecute();
}

class concrete1 extends Command {
    public function __construct(){}

    public function doExecute(){
        echo "con1";
    }
}

class concrete2 extends Command {
    public function doExecute(){
        echo "con2";
    }
}
```

UZORAK TEMPLATE VIEW

Template View

- *Stvara sadržaj u HTMLu ubacivanjem markera u HTML stranu.*
- U idealnom slučaju, Tempate View treba da bude deklarativan. Obično se implementira kao PHP skripta (iako se može koristiti templejt generator kao Smarti).
- *PHP kao jezik koji podržava šablone čak i nudi neke alternativne konstrukte koji se mogu koristiti u Prikaznom šablonu da pojednostave strukturu koda (konstrukt bez zagrada):*

```
<p> Vrednost je: <?=$variable?></p>
<?php foreach ($list as $element): ?>
        <a href="<?=$element?>"><?=$element?></a>
<?php endforeach; ?>
<?php if ($uslov) { ?> Uslovljen tekst <?php } ?>
```

- PHP i drugi server-side programski jezici prvobitno su počeli da koriste < i > oznake, jer ih WYSIWYG editori ignorišu i dopusti dizajneru pristup i editovanje prikaznog šablona bez petljanja u kodu.

Kada upotrebiti Template View

- Snaga Prikaznog šablona je da vam omogućava komponovanje sadržaja stranice gledajući strukturu stranice. Ovo je lakše za većinu ljudi rade i da uče. Posebno lepo podržava ideju da grafički dizajner uobličava stranicu dok programer rade na poslovnoj logici.

- Potencijalni rizici: Stavljanje komplikovane logike u stranu, čime ona postaje teška za održavanje, naročito neprogramerima.

Kada upotrebiti Template View

- *U MVC uzorku, podaci se ne dovlače u prikazni skript pristupajući direktno modelu, jer kontrolerski sloj treba da bude odgovoran za to. Niti prikazni skript sadrži ponašanje aplikacije.*

- *Umesto toga, ne-statičke podatke korišćene u uzorku Prikazni šablon prosleđuje kontroler.*

- *Kao rezultat toga, prikaz nema zavisnosti prema drugim klasama sem onih koje su mu prosleđene (to su obične strukture podataka ili model klase poput entiteta).*

UZORAK UBACIVANJE ZAVISNOSTI

Ubacivanje zavisnosti (Dependency Injection)

- *Korišćenjem šablona ubacivanje zavisnosti, klijentski objekat koji koristi servis (drugi objekat) ne mora da zna sve njegove detalje, a korišćeni objekat može se zameniti objektom sličnih karakteristika bez izmena u klijentu (na primer, u svrhu testiranja)*
- *Na ovaj način ostvaruje se slabija sprega između objekata, što je poželjan princip u dizajnu*
- *Ubacivanje je eksplicitno prosleđivanje zavisnosti (servisa) zavisnom objektu (klijentu). Ova zavisnost se pamti kao deo stanja klijenta. [Suprotno tome bi bilo da klijent sam kreira ili traži servis].*

Primer

- *Kod bez korišćenja ovog šablona:*

```
class Photo {
    /* @var PDO The connection to the database */
    protected $db;

    /* Construct. */
    public function __construct() {
        $this->db = DB::getInstance();
    }
}
```

- *Klasa Photo ima zavisnost od konekcije ka bazi podataka. Nedostatak je što je vrsta konekcije fiksirana u kodu i mora se menjati kod ako se koristi druge klase za perzistenciju podataka (ili lažna-mock baza za testiranje). Zašto bi Photo uopšte vodio računa o tome?*
- *Narušen je princip razdvajanja nadležnosti (SoC).*

Načini ubacivanja zavisnosti

- *Ubacivanje kroz konstruktor*
- *Ubacivanje putem setter metoda*
- *Ubacivanje putem implementacije interfejsa (koji sadrži setter metode za zavisnost)*

Ubacivanje zavisnosti setter metodama

```
class Photo {
    /* @var PDO The connection to the database */
    protected $db;

    public function __construct() {}
    /* Sets the database connection
     * @param PDO $dbConn The connection to the database.*/
    public function setDB($dbConn) {
        $this->db = $dbConn;
    }
}

$photo = new Photo;
$photo->setDB($dbConn);
```

- *Potrebno je da klijent obezbedi po jedan setX metod za svaku zavisnost X*

Ubacivanje zavisnosti u konstruktoru

```
class Photo {
    /* @var PDO The connection to the database */
    protected $db;

    /* Construct.
      @param PDO $db_conn The database connection */
    public function __construct($dbConn) {
        $this->db = $dbConn;
    }
}

$photo = new Photo($dbConn);
```

- *Potrebno je da klijent ima po jedan parametar u svom konstruktoru za svaku zavisnost*

Instanciranje objekata klasa kod DI šablona

- Sada postaje komplikovanije, jer više nije dovoljno samo $photo = new Photo; nego je potrebno pamtiti sve zavisnosti klase, na primer:

```
$photo = new Photo;
$photo->setDB($dbConn);
$photo->setConfig($config);
$photo->setResponse($response);
```

- Rešenje je da se konfigurisanje objekata izvuče na jedno centralizovano mesto, takozvani kontejner sa inverzijom kontrole (inversion of control container).

Kontejner sa inverzijom kontrole

- 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=Inversion of Control Container) koja vodi računa o svim zavisnostima među svim drugim klasama.

Kontejner sa inverzijom kontrole

- U prvoj varijanti, ova klasa ima po jedan metod za svaku klasu koja instancira objekat klase i postavlja sve zavisnosti za njega:

```
class IoC {
    /* @var PDO The connection to the database */
   protected $db;

    /* Create a new instance of Photo and set dependencies. */
   public static newPhoto() {
      $photo = new Photo;
      $photo->setDB(static::$db);
      // $photo->setConfig();
      // $photo->setResponse();
      return $photo;
   }
}
```

- Klijent samo zove newPhoto metod bez potrebe da vodi računa o zavisnostima

```
$photo = IoC::newPhoto();
```

IoC kontejner kao generički registar

- U drugoj varijanti kontejner je fleksibilniji i ne menja mu se kod za svaku klasu koju treba da konfiguriše:

```
class IoC {
    protected static $registry = array();

    /* Add a new resolver to the registry
      array.
      @param  string $name The id
      @param  object $resolve Closure
      that creates instance
      @return void */
    public static function
      register($name, Closure $resolve) {
        static::$registry[$name] =
      $resolve;
    }

    /* Create the instance
      @param  string $name The id
      @return mixed */
    public static function resolve($name) {
        if ( static::registered($name) ){
            $name = static::$registry[$name];
            return $name();
        }
        throw new Exception('Not registered.');
    }

    /** Determine whether the id is registerd
      @param  string $name The id
      @return bool Whether to id exists or not
      */
    public static function registered($name) {
        return array_key_exists($name,
      static::$registry);
    }
}
```

IoC kontejner kao generički registar

- Svaka klasa se prvo **registruje** pri kontejneru, tj. daje joj se identifikator i odgovarajuća lambda (neimenovana) funkcija koja zna da kreira instancu objekta i razreši njegove zavisnosti.

```
// Add `photo` to the registry array, along with a resolver
IoC::register('photo', function() {
    $photo = new Photo;
    $photo->setDB('...');
    $photo->setConfig('...');
    return $photo;
});
```

IoC kontejner kao generički registar

- Klasa se instancira pozivanjem resolve metoda u kontejneru sa odgovarajućim identifikatorom. Resolver pravi novi objekat i postavlja sve njegove zavisnosti

```
// Fetch new photo instance with dependencies set
$photo = IoC::resolve('photo');
```

IoC kontejner kao generički registar

- Mnoge PHP biblioteke (frameworks) podržavaju DI šablon. U Symfony-ju, na primer, moguće je klase registrovati i definisati njihove zavisnosti u spoljnom tekstualnom konfiguracionom fajlu
- Primer fajla u YAML formatu za registrovanje klase Photo:

```
services:
    photo:
    class: Photo
    arguments:  [‘db string’, ‘config’]
```