Základy tvorby interaktívnych aplikácií

- Architektúra, MVC, MVP

- Ing. Jaroslav Erdelyi

- LS 2021-2022

Obsah

- Architektúra interaktívnych web-aplikácií
  - návrhový vzor MVC
  - návrhový vzor MVP
- Ukážky implementácie v JavaScript
  - základný variant
  - rozšírenie o viacero objektov a animáciu
  - štruktúra pre komplexnejšiu aplikáciu

Architektúra interaktívnych webových aplikácií

- Potreba určenia základnej štruktúry aplikácie
- Nie je to len rozdelenie kódu aplikácie
- Celková organizácia projektu
- Rôzne odporúčania a návrhové vzory
- SW a HW obmedzenia

Architektúra interaktívnych webových aplikácií

- Javascript <script>
- Kreslenie <canvas>
- onkeydown ...
- <audio>
- <audio>
- WebSocket
- Rieši browser

Hlavný cyklus

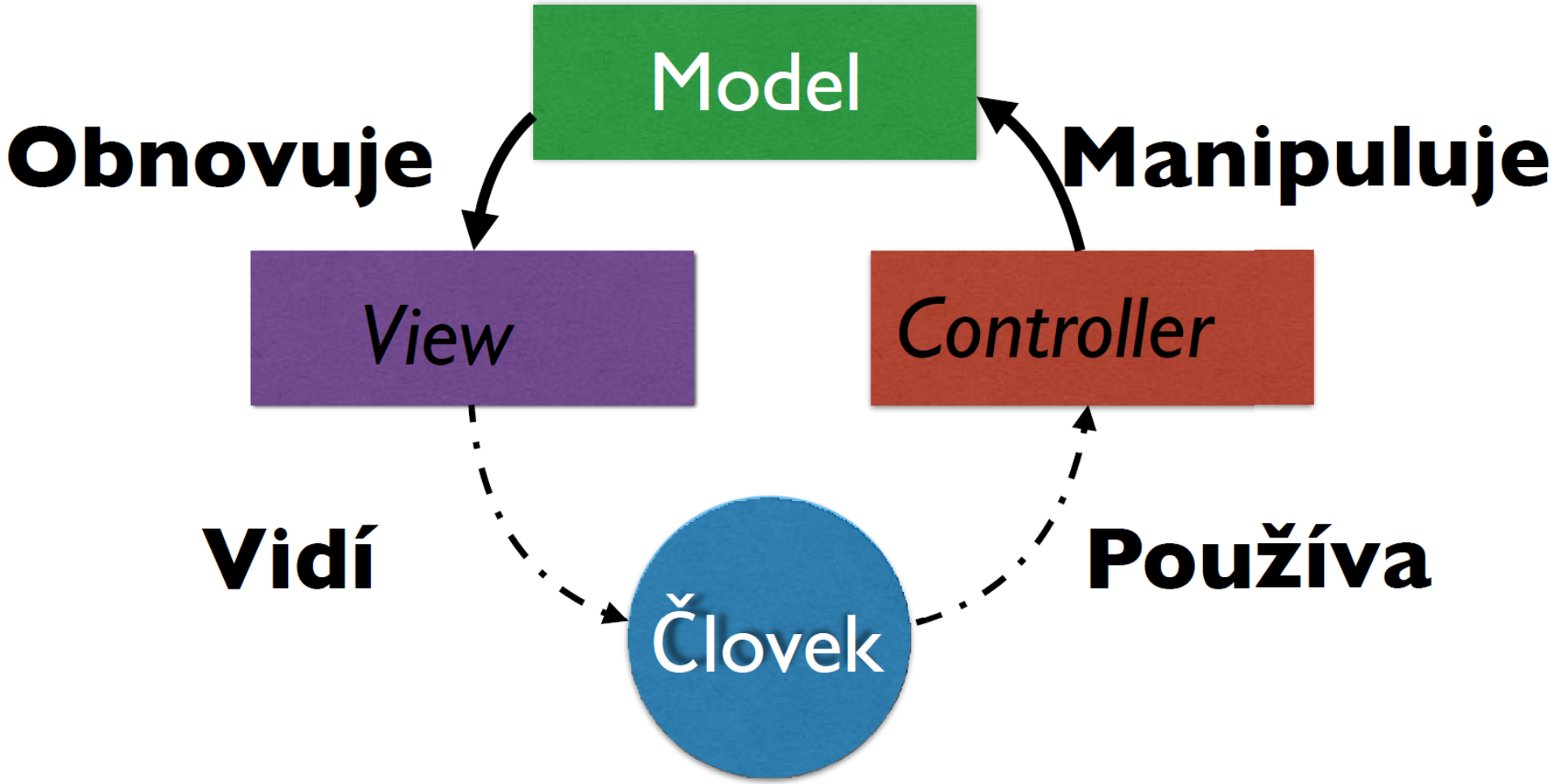
- Typicky hlavný cyklus v hre vyzerá nasledovne JavaScript

```
function mainLoop()
{
    MoveScene();
    DrawScene();
    PlaySounds();
    HandleInputs();
}
setInterval(mainLoop,1000/60);    // 60 fps
```

Návrhové vzory

- Základným problémom je určiť aká časť kódu je zodpovedná za aké operácie
- Návrhové vzory popisujú vysoko-úrovňovú organizáciu, ktorá rieši bežne problémy
- V objektovo orientovanom programovaní je návrhový vzor základom, z ktorého sa vychádza

Vzor Model View Controler (MVC)



MVC

- **Model** – vnútorná reprezentácia dát aplikácie
- **View** - pohľad na model a jeho zobrazenie
- **Controller** – riadenie a spracovanie vstupov a zmien
- Zvyčajne sa tieto časti implementujú pomocou oddelených objektov alebo modulov

Model

- Väčšina programov má v prvom rade vykonávať prácu a nemusí dobre vyzerať
  - Existujú samozrejme výnimky
  - Užitočne aplikácie existovali dávno pred GUI
- Model je časťou, ktorá vykonáva riešenie, všetku prácu – je modelom riešenia problému
- Model by mal byť nezávislý od ostatných komponentov
  - Poskytuje však rozhranie (metódy, funkcie), ktoré možno použiť

Controller

- **Controller** určuje ako sa bude s **Modelom** pracovať
  - Často krát je **Controller** samotne GUI
- Je skoro vždy možne **Controller** a **Model** oddeliť
- Návrh **Controller-a** je však zvyčajne závislý od **Modelu**
- **Model** by sa nemal nikdy prispôsobovať **Controller-u**

View

- Používateľ očakáva, že bude *vidieť* stav aplikácie
- **View** poskytuje náhľad na to čo **Model** vykonáva
  - **View** je pasívny a neovplyvňuje **Model**
- **Model** je zväčša nezávislý od **View,** avšak poskytuje mu rozhranie (funkcie/metódy)
- **Pohľad** by *nemal zobrazovať* nič súvisiace s činnosťou **Controller**

View a Controller

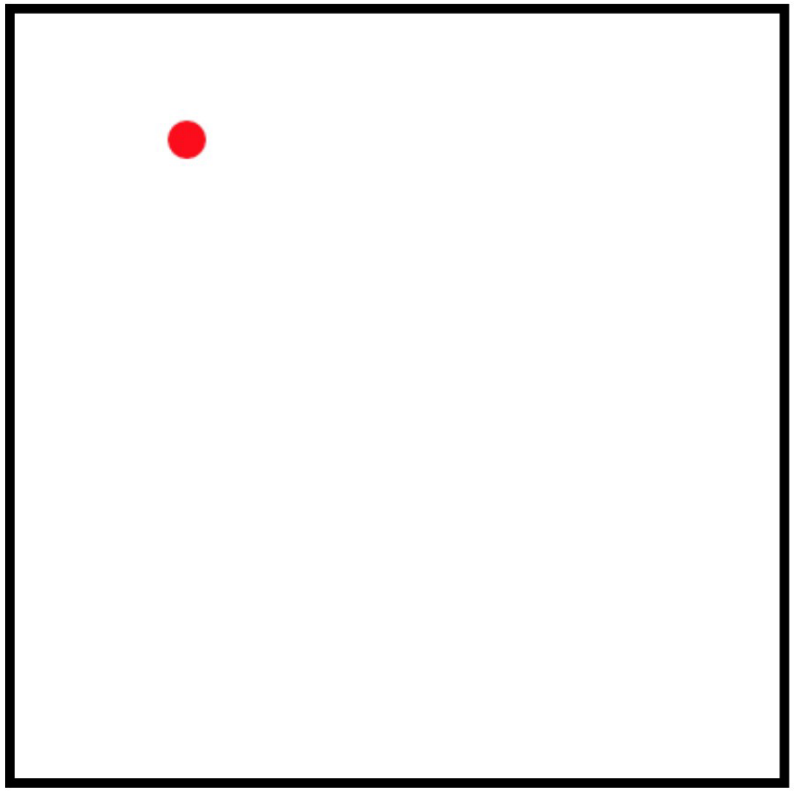
- Často krát je však užitočne spojiť **Controller** a **View**, hlavne u malých aplikácii
- Je to vhodne napríklad pri vysokom prepojení **Controller** s **View**
- **Model** však vždy zostava oddelený
- **Nikdy** nemiešajte GUI kód s **Modelom**

Oddelenie závislostí

- Nezávislosť časti kódu vedie k robustnosti, flexibilite pri zmenách a ľahšej udržiava teľnosti kódu
- Nezávislosť časti kódu je žiaduca vlastnosť
- Z predmetu *Princípy softvérového inžinierstva:*
  - *Súdržnosť* - ako silno je softvérový prvok zameraný
  - *Previazanosť* - ako silne je softvérový prvok spojený s inými prvkami

Oddelenie závislostí

- **Model** by nemal byť kontaminovaný ovládaním či kódom vykresľovania
- **View** by mal reprezentovať **Model** taký aký naozaj je bez skreslenia
- **Controller** by mal komunikovať s **Modelom** a **View** len za účelom ich manipulácie
- **Controller** môže napríklad nastaviť premenne, ktoré **View** a **Model** používa



Step

Tick: 4

Dot

- Ukážková aplikácia, demonštrácia MVC
- Stlačenie tlačidla *krok* vykoná posun bodu
- Bod sa odráža od hraníc obrazu
- Krok a pozícia sú zobrazene v texte
- Situácia je ilustrovaná obrazom
- Pozri implementáciu v 1_dot.html

Dot

- **Model** ovláda pohyb bodu
- V tomto prípade musí **Model** vedieť veľkosť **View**
  - Pretože potrebuje modelovať hranice od ktorých nastane odraz
- **Model** nevie nič o zvyšku aplikácie a jej ovládacích prvkoch
- **Controller** spracúva stlačenie tlačidla a aktualizuje **Model**
- **View** zobrazuje scénu a počet krokov

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bouncing Dot</title>
    <script>

    </script>
    <style>
      #canvas {
        border-style: solid;
        border-width: 5px;
      }
    </style>
  </head>

  <body>
    <p>
      <canvas id="canvas" height="400" width="400"></canvas>
    </p>
    <p>
      <button id="button">Step</button>
    </p>
    <p id="text">
      Please click Step!
    </p>
  </body>
</html>
```

Celá aplikácia

```
var canvas
var ctx
var tick = 0  // pocitanie krokov po stlaceni tlacitka

// MODEL
//

// View
//

// Controller
//

// Initialize the application
window.onload = function() { // Main
  // Set up global variables for easy access
  button = document.getElementById("button")   // objekt pre tlacitko
  text = document.getElementById("text")       // objekt pre text
  canvas = document.getElementById("canvas")   // objekt pre canvas
  ctx = canvas.getContext("2d")                // objekt pre contex
  button.onclick = step                        // stlacenie tlacitka zavola step()
}
```

Model

```
class Dot{
  x = 50;
  y = 50;
  dx = 10;
  dy = 4;

  // A method to move the object
  move() {
    if (this.x >= canvas.width || this.x <= 0) {
      this.dx *= -1
    }
    if (this.y >= canvas.height || this.y <= 0) {
      this.dy *= -1
    }

    // Update object position
    this.x = this.x + this.dx
    this.y = this.y + this.dy
  }
```

View

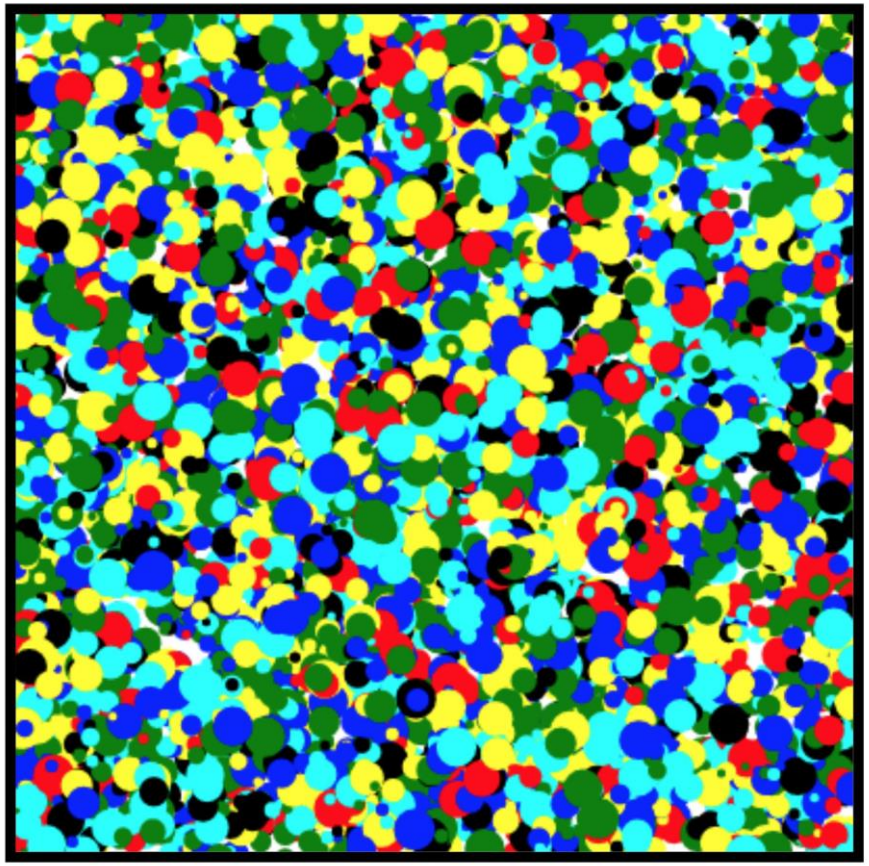
```
// Just takes care of drawing based on model
function drawDot() {
  // Clear the canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height)

  // Render a dot
  ctx.fillStyle = "red"
  ctx.beginPath()
  ctx.arc(dot.x, dot.y, 10, 0, Math.PI * 2)
  ctx.closePath()
  ctx.fill()
}
```

```
// Update html text
function setText() {
  text.innerHTML = "Tick: " + tick
}
```

Controller

```
// Controls what is being done in the application
// In this case it just orchestrates the model to move and view
to
// render in a loop
function step() {
  tick++
  dot.move()
  drawDot()
  setText()
}
```



Start

Tick: 24

Rozšírenie aplikácie Dot

- Zobrazenie viacerých objektov
- Kolekcia objektov, virtuálna scéna
- Každý objekt sa inicializuje na náhodnej pozícii
  - generovanie cez: ***Konštruktor***
- Objekty sa pohybujú
  - animácia 30fps
- Pozri implementáciu v 2_dots.htm

Rozšírenie aplikácie Dot

- Globálne premenne
  - časovač *timer*, počet krokov *tick*, pole farieb *colours,* pole objektov *dots*

```
var canvas
var ctx
var timer
var tick = 0
var colours = ["red", "green", "blue", "yellow", "cyan",
"black"]

// Model
//
// In this case the model will be a collection of objects
var dots = []
```

Inicializácia

- tlačidlo bude vykonávať funkciu *start()*
- v cykle vytvoríme inštancie *Dot* a vložíme do poľa

```
// Initialization
window.onload = function () {
    // Setup global variables
    button = document.getElementById("button")
    text = document.getElementById("text")
    canvas = document.getElementById("canvas")
    ctx = canvas.getContext("2d")
    button.onclick = start

    // Create 5000 dots
    for (i = 0; i < 5000; i++) {
        dots.push( Dot() )
    }
}
```

Model

```
class Dot{
    // We will make a new object newDot
    constructor(canvas){
        this.x = Math.random() * canvas.width
        this.y = Math.random() * canvas.height
        this.dx = Math.random() * 10 - 5
        this.dy = Math.random() * 10 - 5
        this.size = Math.random() * 8 + 2

        // Randomly select a colour
        var colour_index = Math.round(Math.random() * (colours.length - 1))
        this.colour = colours[colour_index]
    }
```

Model

```
move(canvas) {
    // Logic
    if (this.x >= canvas.width || this.x <= 0) {
        this.dx *= -1
    }
    if (this.y >= canvas.height || this.y <= 0) {
        this.dy *= -1
    }

    // Up  (property) Dot.x: number
    this.x = this.x + this.dx
    this.y = this.y + this.dy
}
```

View

```
// View
//
function draw() {
    // Clear canvas
    ctx.clearRect(0, 0, canvas.width, canvas.height)

    // Render each dot
    for (i in dots) {
        var dot = dots[i]
        ctx.fillStyle = dot.colour
        ctx.beginPath();
        ctx.arc(dot.x, dot.y, dot.size, 0, Math.PI * 2);
        ctx.closePath();
        ctx.fill();
    }
}
```

Controller

- funkcia *move()* zavolá nad každým objektom *Dots* metódu *move()*
- *step()* vykoná jeden krok simulácie (animácie)

```
// Move all dots
function move() {
    for (var i in dots) {
        dots[i].move()
    }
}

function step() {
    tick++
    move()
    draw()
    setText()
}
```

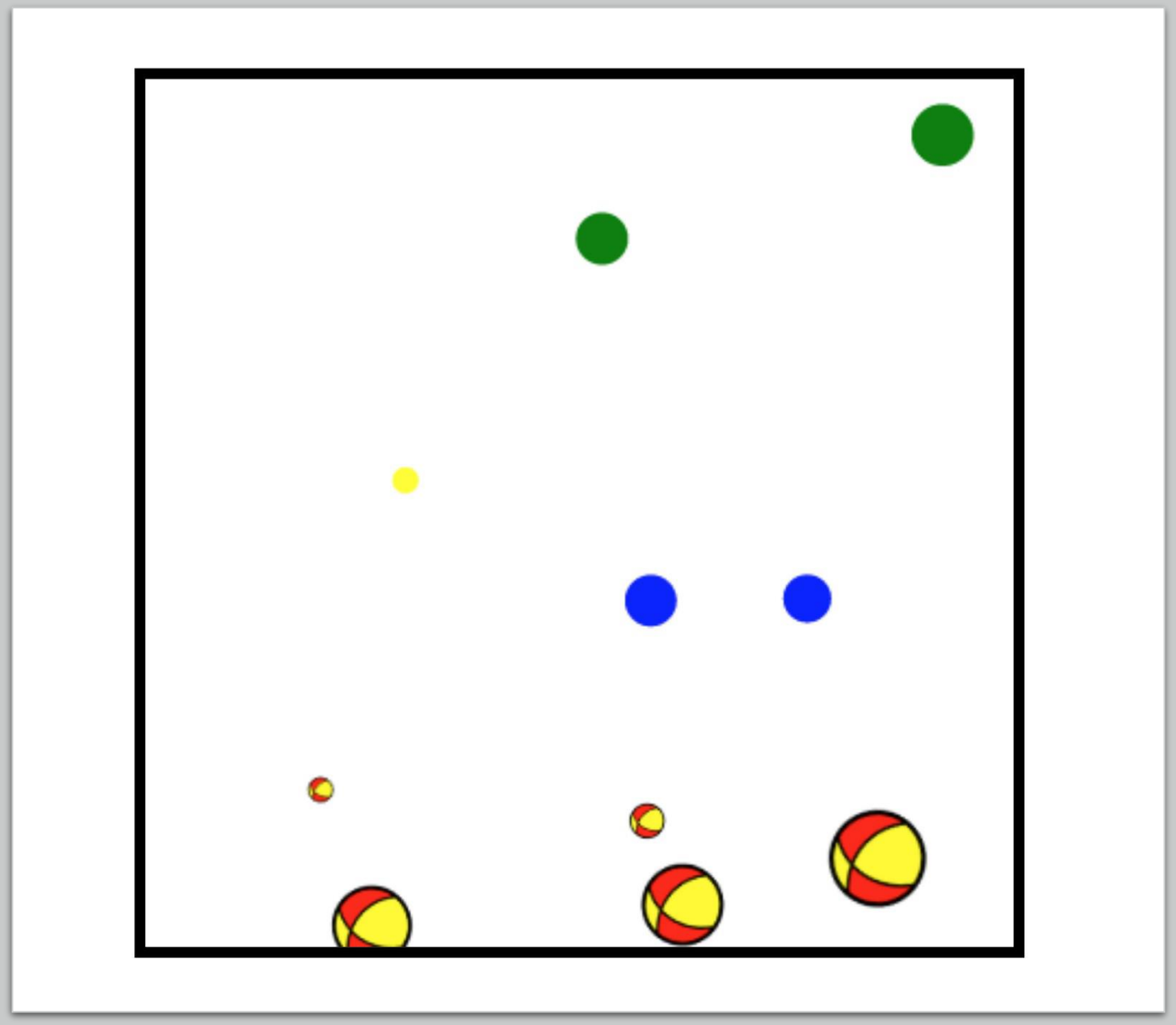
Controller

- *start()*
  - prepína medzi 2 stavmi tlačidla
  - maže alebo nastavuje časovač, ktorý potom v pravidelných intervaloch vola *step()*

```
// Starts and stops a timer that will call step() 30 times per
second
function start() {
    if (timer) {
        timer = clearInterval(timer)
        button.textContent = "Start"
    } else {
        timer = setInterval(step, 1000 / 30);
        button.textContent = "Stop"
    }
}
```

Odrážajúca sa lopta

- Rozdelenie kódu do súborov
  - vhodná štruktúra
  - inicializácia
  - hlavná slučka
- Viacero druhov objektov
  - nezávisle zobrazenie
  - pohyb
- Práca s časom
- Pozri implementáciu v adresári ***loptičky***



Základná štruktúra

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bouncing Balls</title>
    <meta charset="utf-8" />
    <link href="css/balls.css" rel="stylesheet" />
    <script src="js/ball.js"></script>
    <script src="js/dot.js"></script>
    <script src="js/main.js"></script>
  </head>
  <body>
    <p>
      <canvas id="canvas" height="400" width="400">No Canvas :(
      </canvas>
    </p>
    <img id="image" src="img/ball.png" hidden="true" />
  </body>
</html>
```

Globálne premenné a inicializácia

```
var canvas
var ctx
var time

// Model
var scene = []

// View / Controller / Main loop

// Initialization
window.onload = function() {
  canvas = document.getElementById("canvas")
  ctx = canvas.getContext("2d")

  for (i = 0; i < 5; i++) {         // Create 5 dots and store them in scene
    scene.push( new Dot() )
  }

  for (i = 0; i < 5; i++) {        // Create 5 balls and store them in scene
    scene.push( new Ball() )
  }
  time = Date.now()
  requestAnimationFrame(step)
}
```

Hlavná slučka

- počíta, koľko času ubehlo od predch. snímky: *dt*
- posunie a vykresli všetky objekty v scéne (simulácia)
- vyžiada si vytvorenie anim. rámca (vola samu seba)

```
function step() {
  console.log("Step")

  // Get time delta
  var now = Date.now()
  var dt = (now - time) / 100
  time = now

  move(dt)
  draw()

  requestAnimationFrame(step)
}
```

View a Controller

```
// View
function draw() {
  // Clear canvas
  ctx.fillStyle = "white"
  ctx.fillRect(0, 0, canvas.width, canvas.height)

  // Render all objects in scene
  for (i in scene) {
    scene[i].draw()
  }
}
```

```
// Controller
function move(dt) {

  // Move all objects in scene (pass delta time)
  for (var i in scene) {
    scene[i].move(dt)
  }
}
```

Objekt Ball

```
class Ball {
  // Initialization
  constructor() {
    this.image = document.getElementById("image")

    this.x = Math.random() * canvas.width
    this.y = Math.random() * canvas.height
    this.dx = Math.random() * 50 - 25
    this.dy = Math.random() * 50 - 25
    this.size = Math.random() + .3
    this.rotation = 0
  }

  // Movement logic
  move(dt) { ... }

  // Render self
  draw() { ... }
}
```

Pohyb objektu Ball

```
 // Movement logic
  move(dt) {
    console.log("hello")
    if (this.x > canvas.width) {
      this.x = canvas.width
      this.dx = -Math.abs(this.dx)
    }
    if (this.x < 0) { ... }
    if (this.y > canvas.height) {
      this.y = canvas.height
      this.dy = -Math.abs(this.dy) * 0.9 // Reduce movement
    }
    if (this.y < 0) { ... }

    // Movement - !!! each motion (speed, gravity vector) is multiplied with
dt !!!
    this.x += this.dx * dt
    this.y += this.dy * dt

    this.dy += 9.8 * dt      // Add gravitational force

    this.rotation += dt        // Rotate based on time
  }
```

Vykreslenie objektu Ball

- Transformácie
  - aplikujú sa v opačnom poradí
  - sa kumulujú, preto **každý** objekt by si mal najprv odložiť stav
    - *Context*-u, a po vykreslení kontext obnoviť

```
  // Render self - saving state allows to transform object
individually
  draw() {
    ctx.save()              // Save current state
    ctx.translate(this.x, this.y)
    ctx.rotate(this.rotation)
    ctx.scale(this.size, this.size)
    ctx.drawImage(this.image, -20, -20, 40, 40)
    ctx.restore()           // Restore state
  }
```

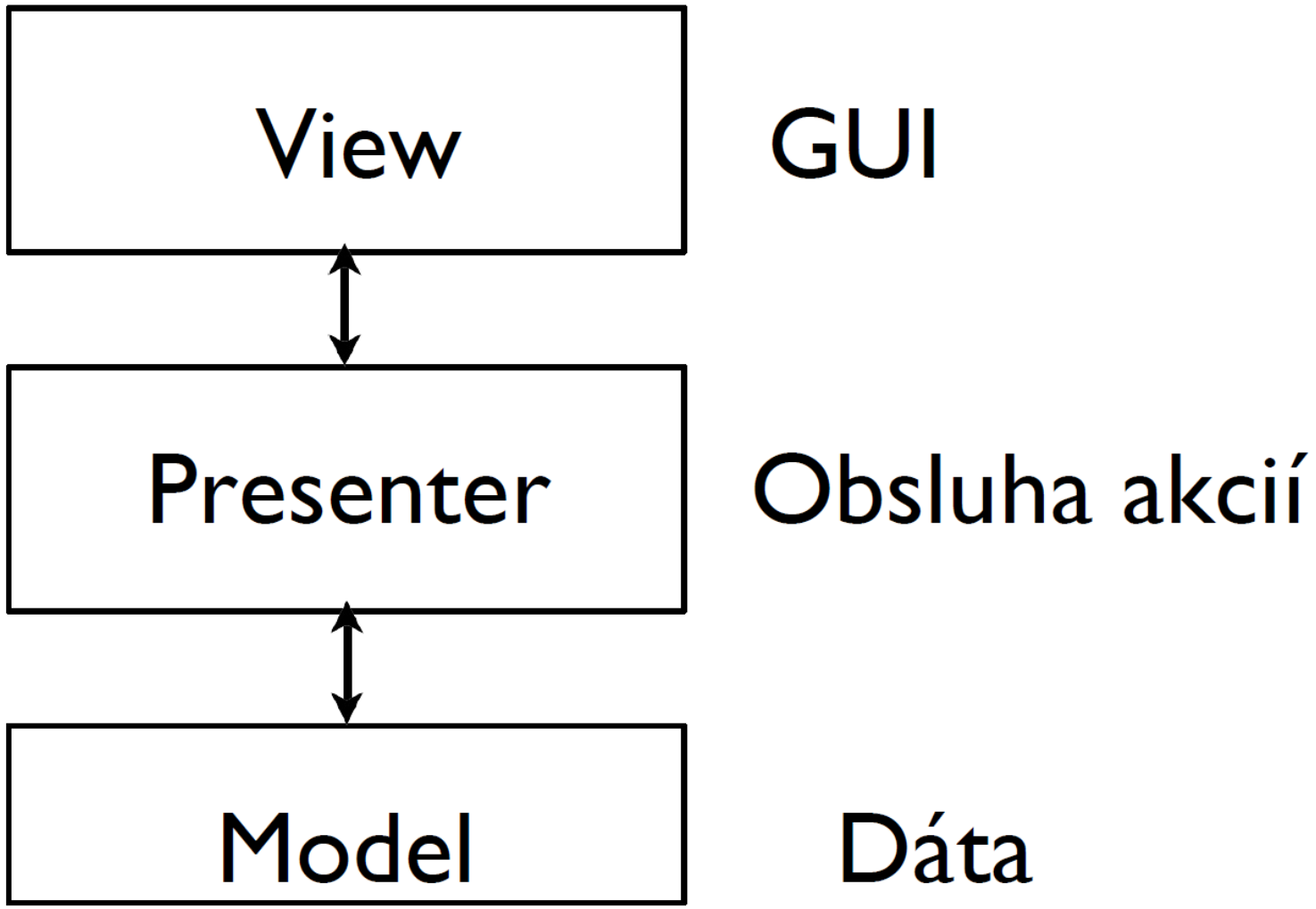
MVC

- Rekapitulácia
  - Typicky návrhový vzor uplatňovaný pri tvorbe interaktívnych aplikácii
  - Hlavná myšlienka je oddelenie kódu používateľského rozhrania, dát aplikácie a jej logiky

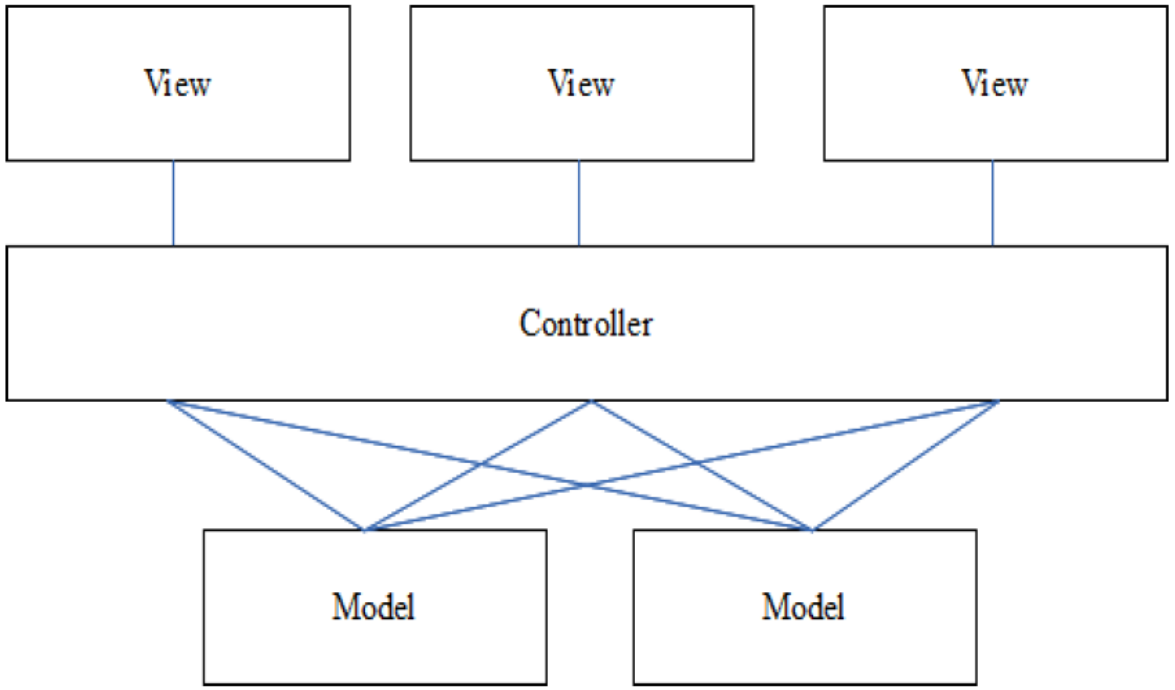
Model View Presenter - MVP

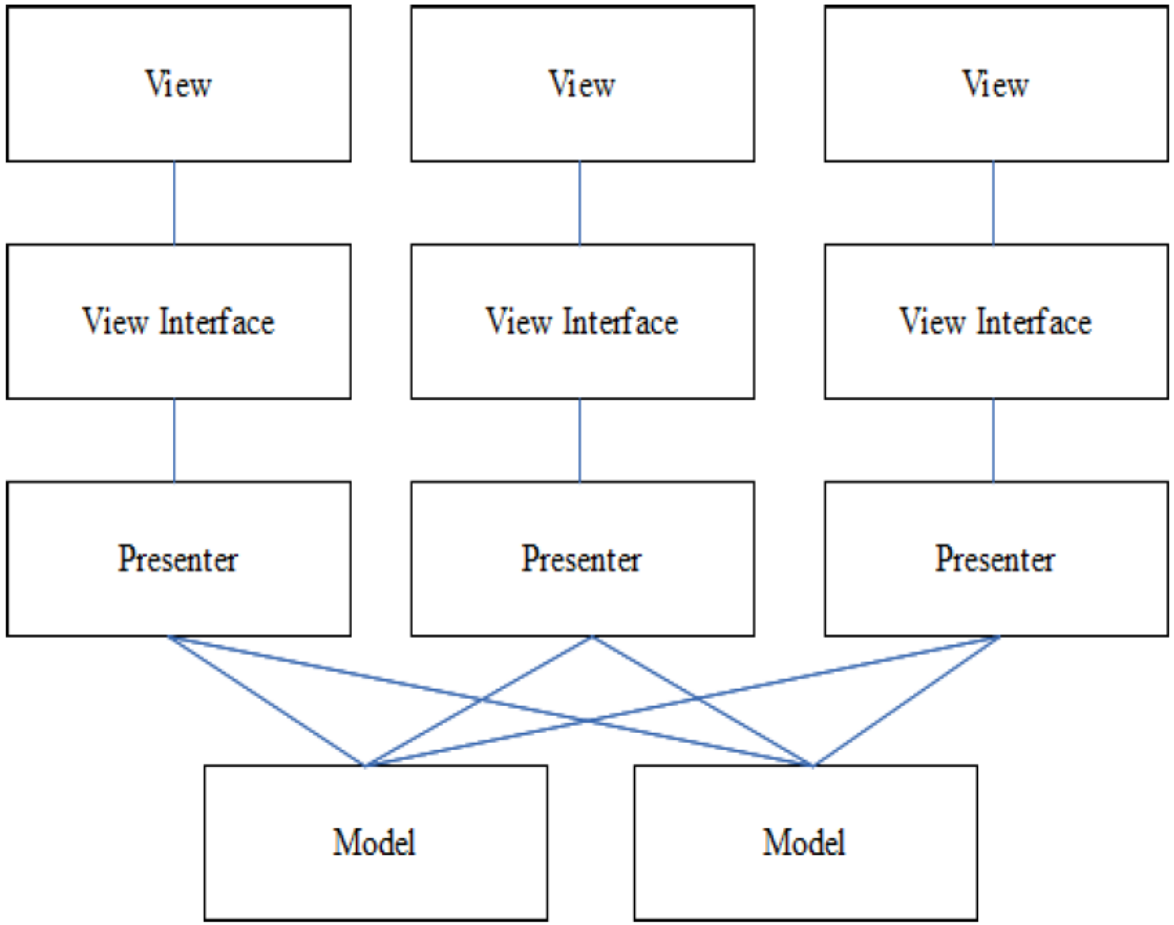
- Variácia myšlienky použitej v MVC
- Zamerane na moderne používateľské rozhrania a web aplikácie
- View predstavuje GUI aplikácie a jeho logika a implementácia je daná, napríklad HTML dokumentom
- Nie je nutne implementovať kód spracovania interakcie od používateľa, stačí obslúžiť len jeho akcie
- Presenter je zodpovedný za aplikačnú logiku a predstavuje jadro aplikácie, poskytuje používateľovi ovládanie cez View


MVP


View
GUI
Presenter
Obsluha akcií
Model
Dáta

MVC vs MVP

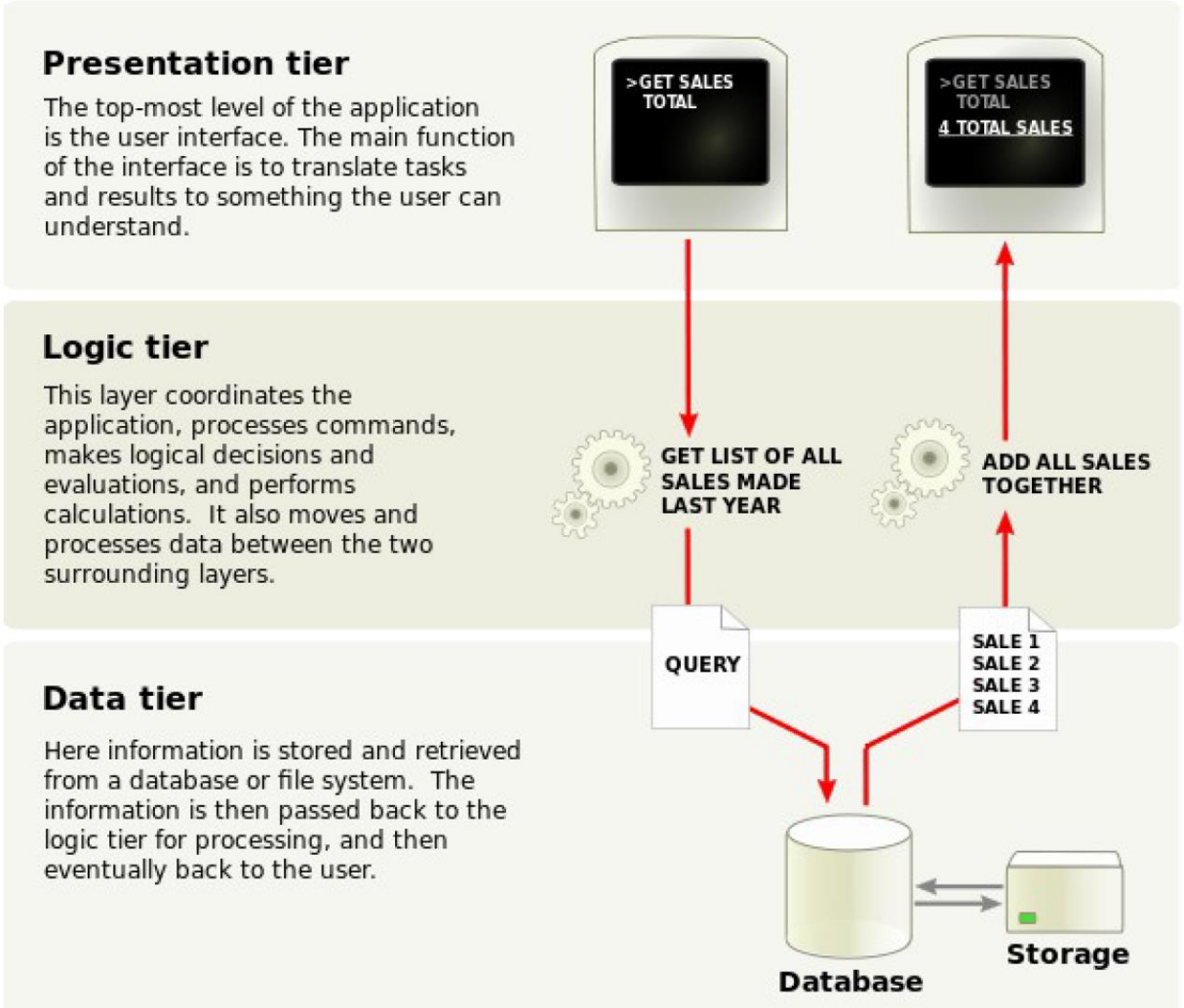

View
View
View
Controller
Model
Model


View
View
View
View Interface
View Interface
View Interface
Presenter
Presenter
Presenter
Model
Model

Troj-vrstvová architektúra

- MVP je vzor pre návrh interakcie, avšak možno ho zovšeobecniť pre návrh interaktívnych aplikácii
- Trojvrstvový návrh aplikácii typicky pre Web a Klient-server aplikácie
- Typicky oddeľujeme prezentačnú vrstvu od logiky aplikácie a jej dát

Troj-vrstvová architektúra



Zhrnutie

- Čo sú návrhové vzory
- Návrhový vzor MVC
  - Model – reprezentuje problémovú oblasť
  - View – zabezpečuje zobrazenie Modelu
  - Controler – rieši spracovanie vstupov a riadi aplikáciu
- Príklady využitia MVC v implementácii
  - základ pre Vaše riešenie
- Porovnanie MVC s MVP