Základy tvorby interaktívnych aplikácií

- Návrhový vzor Observer a Widgets

- Ing. Jaroslav Erdelyi

- LS 2021-2022

Obsah

- Vzor Observer
- Rozšírenie vzoru na všeobecne použitie
- Šírenie udalosti v aplikáciách
- Widgets a ich ukážková implementácia v JS
- Využitie dedenia

Vzor Obeserver

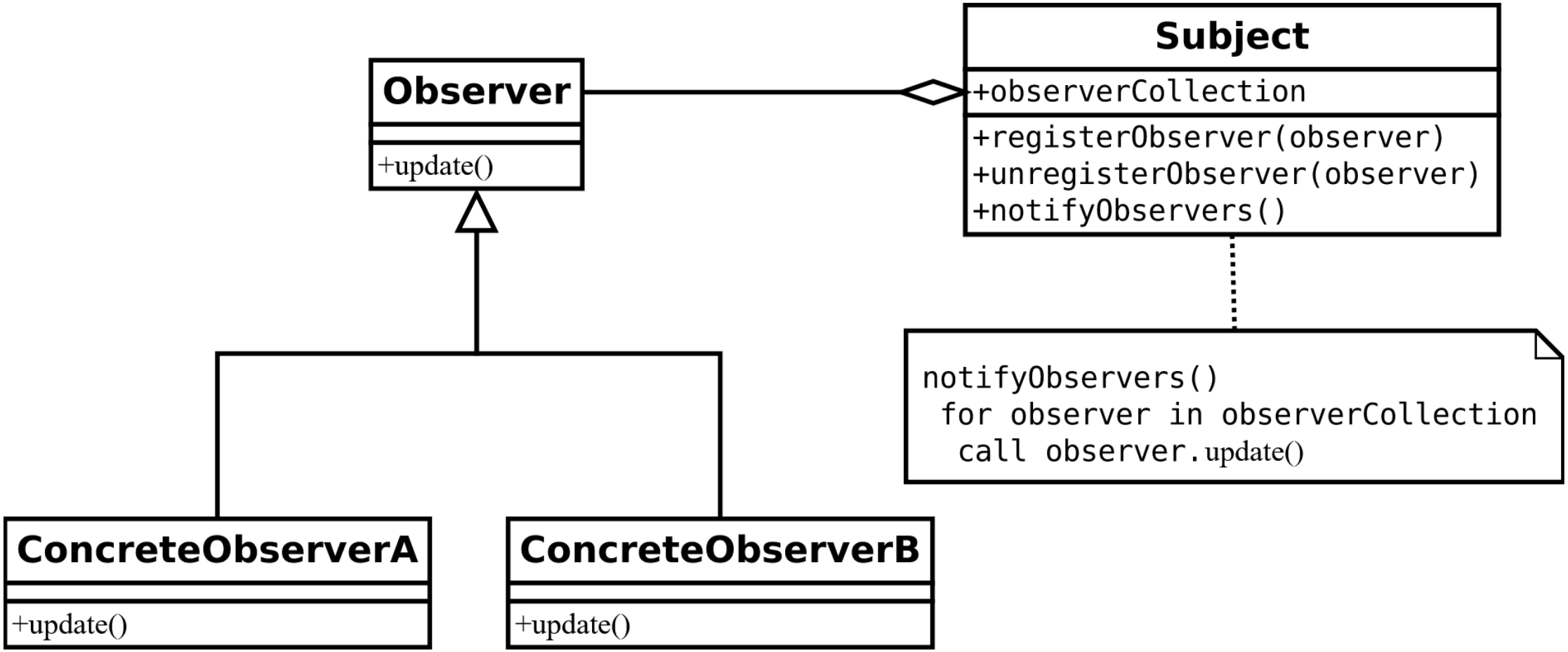
Vzor Observer

- Návrhový vzor *Observer*, *Observer Pattern*
- Rieši problém šírenia sprav a udalosti medzi objektami v aplikácii
- Kľúčová súčasť vzoru MVC a implementácie GUI aplikácii
- Súčasťou implementácie mnohých knižníc a systémov

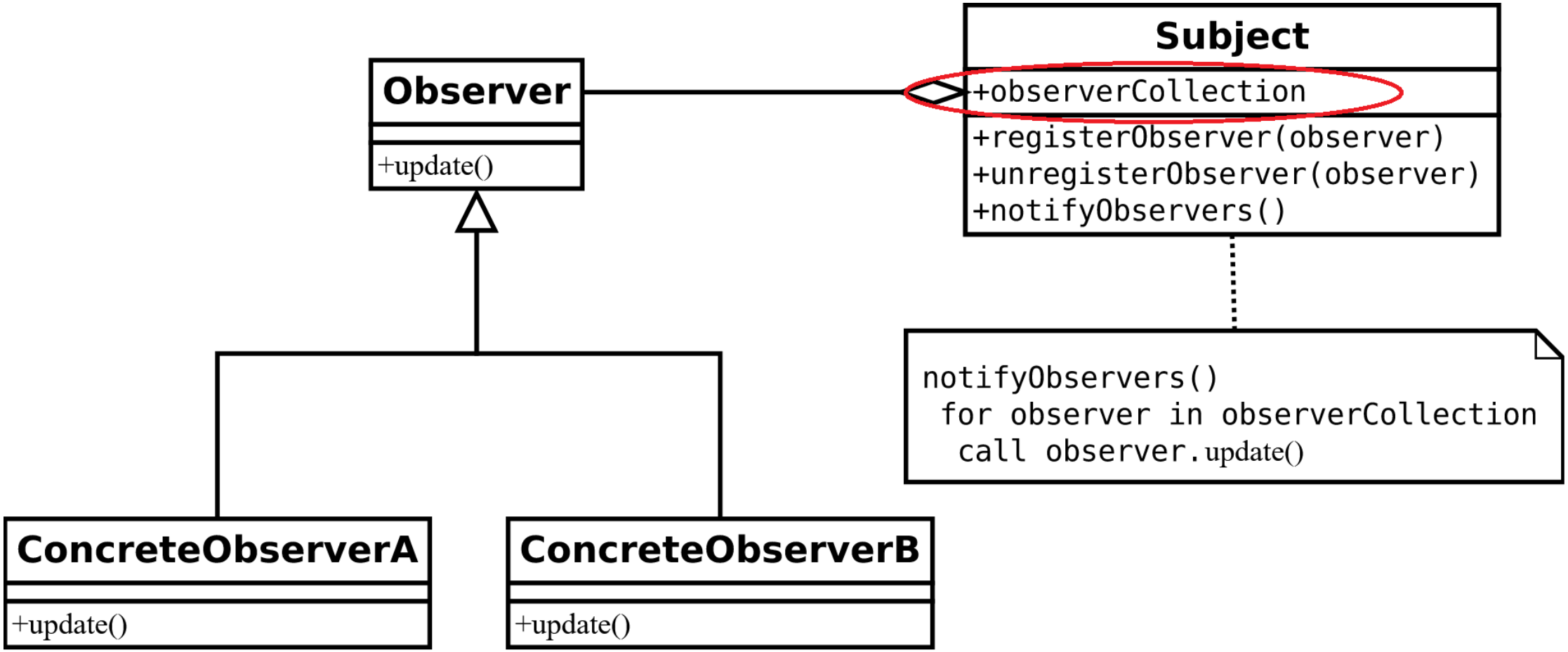
Vzor Observer

- Vzor pozostava z dvoch objektov
- *Subject* je objekt, ktory je pozorovany
- *Observer* je objekt, ktory pozoruje
- Pri vyskyte udalosti informuje *Subject* všetky *Observer* objekty, ktore ho pozoruju o zmene
- *Observer* reaguje na zmenu

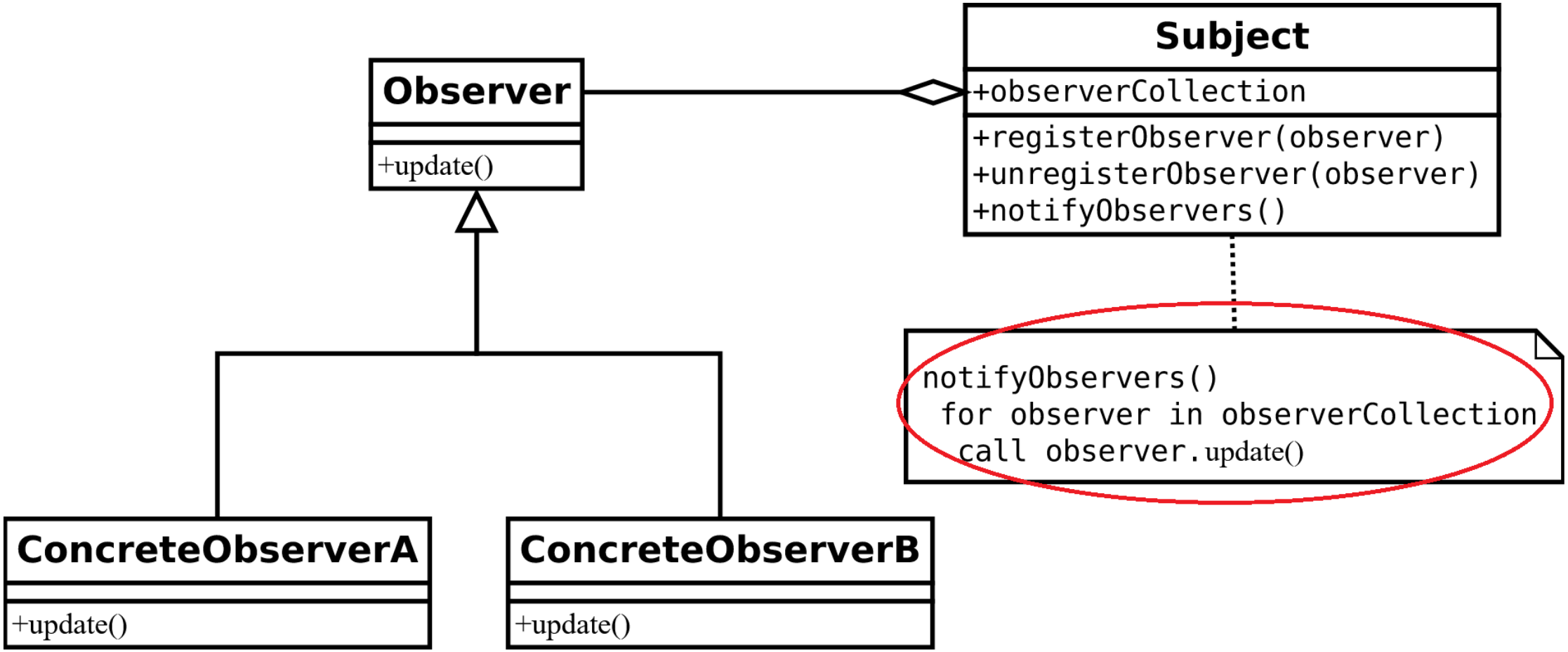
Vzor Observer - UML



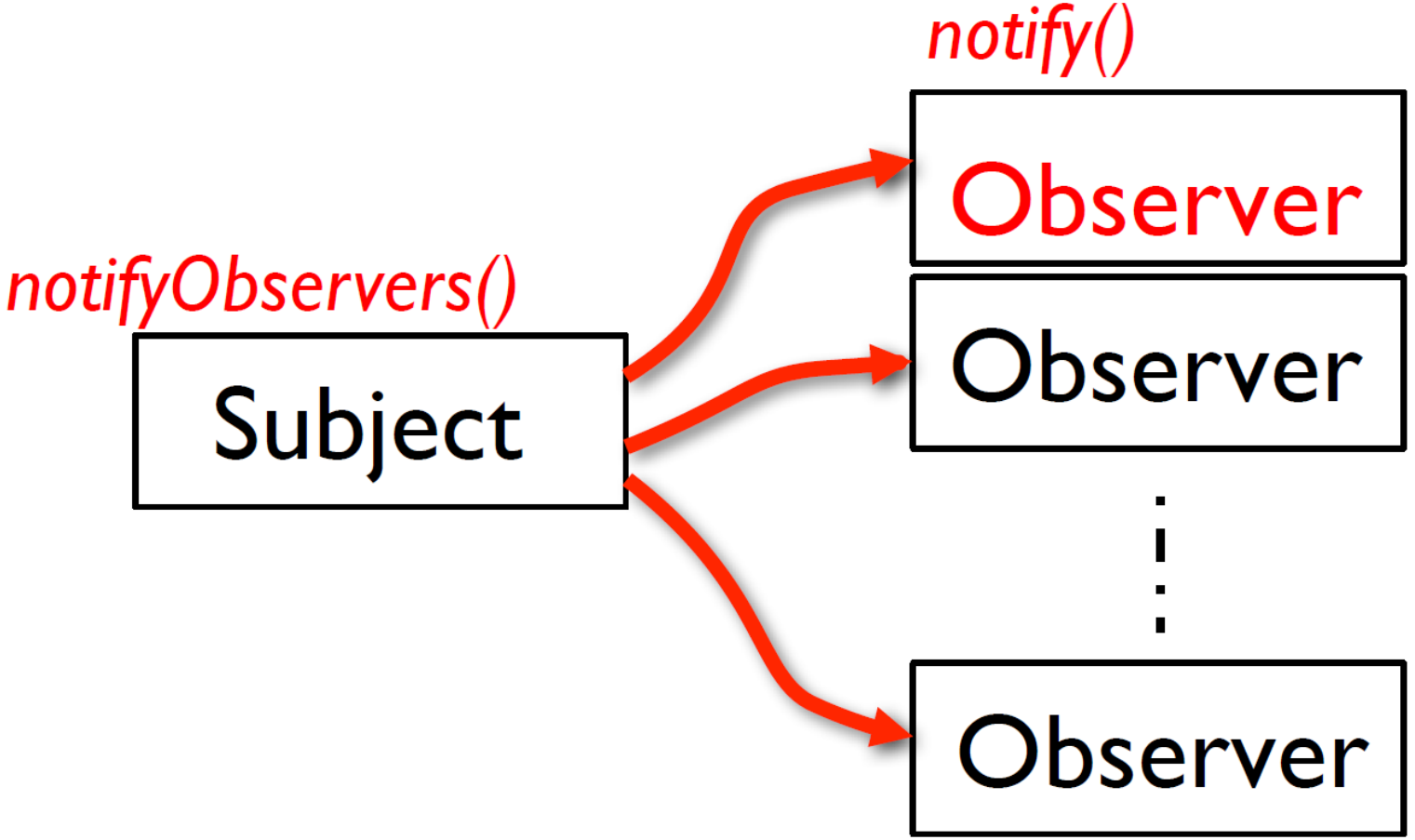
Vzor Observer - pozorovatelia



Vzor Observer



Vzor Observer



Subjekt

```
// Object Subject
function Subject() {
  // Storage for observers
  this.observerCollection = []
}
// Add an observer
Subject.prototype.registerObserver = function(observer) {
  this.observerCollection.push(observer)
}
// Remove an observer
Subject.prototype.unregisterObserver = function(observer) {
  var index = this.observerCollection.indexOf(observer)
  delete this.observerCollection[index]
}
// Notify all observers
Subject.prototype.notifyObservers = function() {
  for (var index in this.observerCollection) {
    var observer = this.observerCollection[index]
    observer.notify(this)
  }
}
```

Objekt

Ako funguje notify

```
// Observer object
function Observer() {}

// Notify method that Subject calls on notifyObservers
Observer.prototype.notify = function(subject) {
  console.log("Observer.notify from " + subject)
    // Do something
}
```

Rozšírenie

- Čo ak bude *Observer* zároveň tiež *Subject* pre ďalšie objekty?
- Objekty môžeme usporiadať do stromov, grafov
  - Správy by bolo možne šíriť celou štruktúrou
- Každý objekt však musí vedieť ako ma komunikovať
- Potrebujeme teda implementáciu jedinou **triedou**

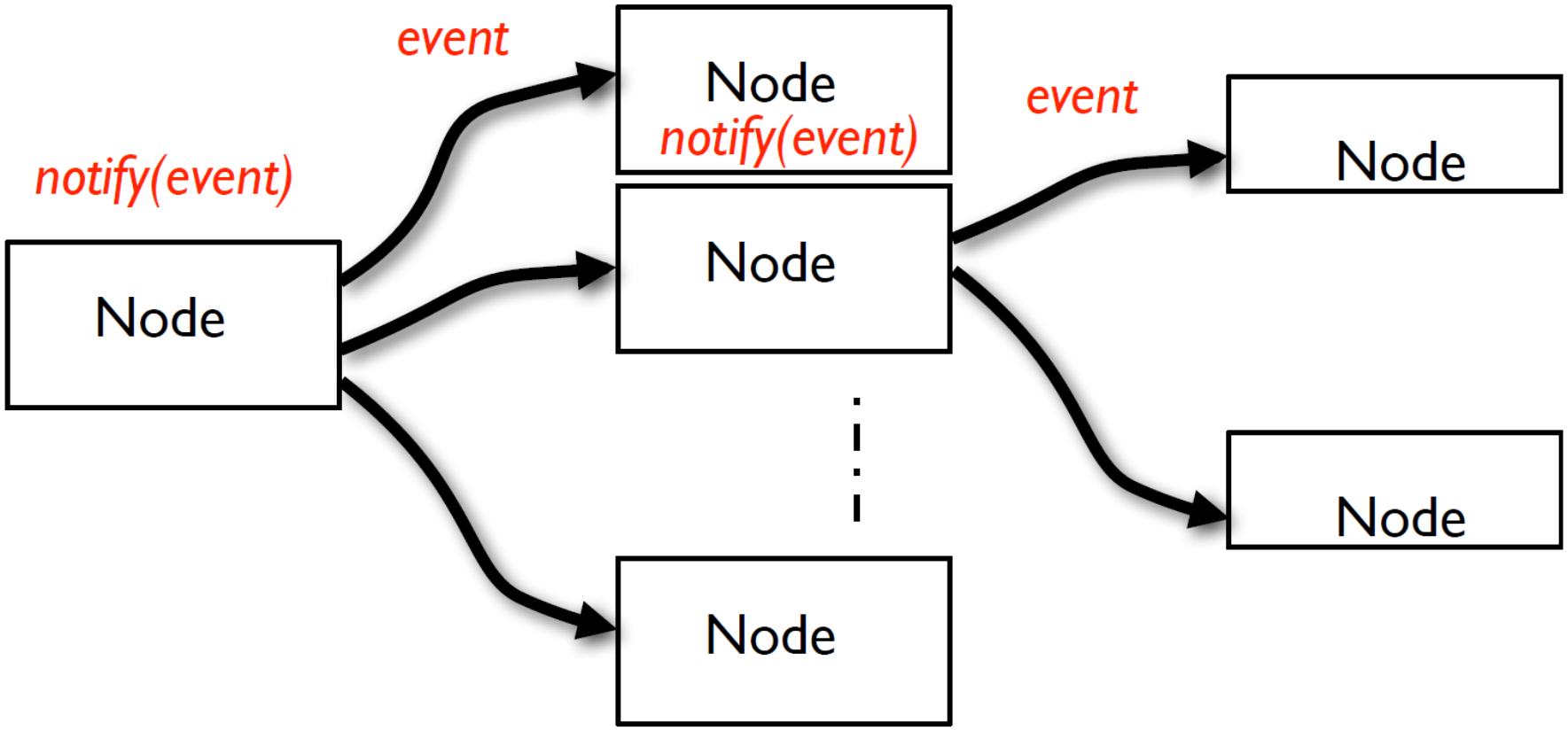
Node - uzol

- Definujeme uzol stromu objektom *Node*
- Uzol uchováva zoznam svojich pozorovateľov podobne ako *Subject*
- Uzol by mal ale vedieť poslať akúkoľvek správu, nie len *notify()*
- Použijeme **dedenie** vlastnosti z objektu *Node* pre ďalšie objekty

Node - uzol

- *nodes* – zoznam observerov
- *add()* - prida observer
- *remove()* – odoberie observer
- *notify()* - zavola metodu “event”

Šírenie správy



Node - uzol

- implementuje zároveň *Subject* aj *Observer*

```
class Node {
  constructor() { this.nodes = [] }
  // Add node
  add(node) { this.nodes.push(node) }
  // Remove node
  remove(node) {
    var index = this.nodes.indexOf(node)
    delete this.nodes[index]
  }
  // Pass message "event" to child nodes
  notify(event, argument) {
    for (var index in this.nodes) {
      var node = this.nodes[index]
      // If node has defined message method, then call the method
      if (typeof (node[event]) == "function")
        node[event](argument)
    }
  }
}
```

Trieda Hello

- dedí z triedy *Node*
- *hello()* - “oficiálny” interface triedy
  - vykoná *onHello()* a pošle “hello” správu svojim potomkom
  - *onHello()* - samotna implementacia “hello” metody

```
class Hello extends Node {
  constructor(name) {
    // Initialize node
    super();
    this.name = name
  }
  // Example messages
  hello(parent) {
    this.onHello(parent)
    // Pass message to children
    this.notify("hello", this)
  }
  // continues on next slide
```

```
window.onload = function() {
  console.log("\n>>> ROOT Message")
  // Add root node
  var root = new Hello("root")
  // Send hello message
  root.hello()

  console.log("\n>>>> Children Message")

  // Add two children to root
  var child1 = new Hello("child1")
  var child2 = new Hello("child2")
  root.add(child1)
  root.add(child2)
  // Send hello message
  root.hello()

  // continues on next slide
```

Ukážka použitia

```
  console.log("\n>>>> Grandchildren Message")

  // Create grandchildren
  var grandChild = new Hello("grandChild")
  // Create special object and redefine its onhello()
  var blackSheep = new Hello("blackSheep")
  blackSheep.onHello = function(parent) {
    console.log("I don't want to tell.")
  }
  // Add grandchildren to child+
  child1.add(grandChild)
  child1.add(blackSheep)

  // Send hello message from tree root
  root.hello()
}
```

Ukážka použitia

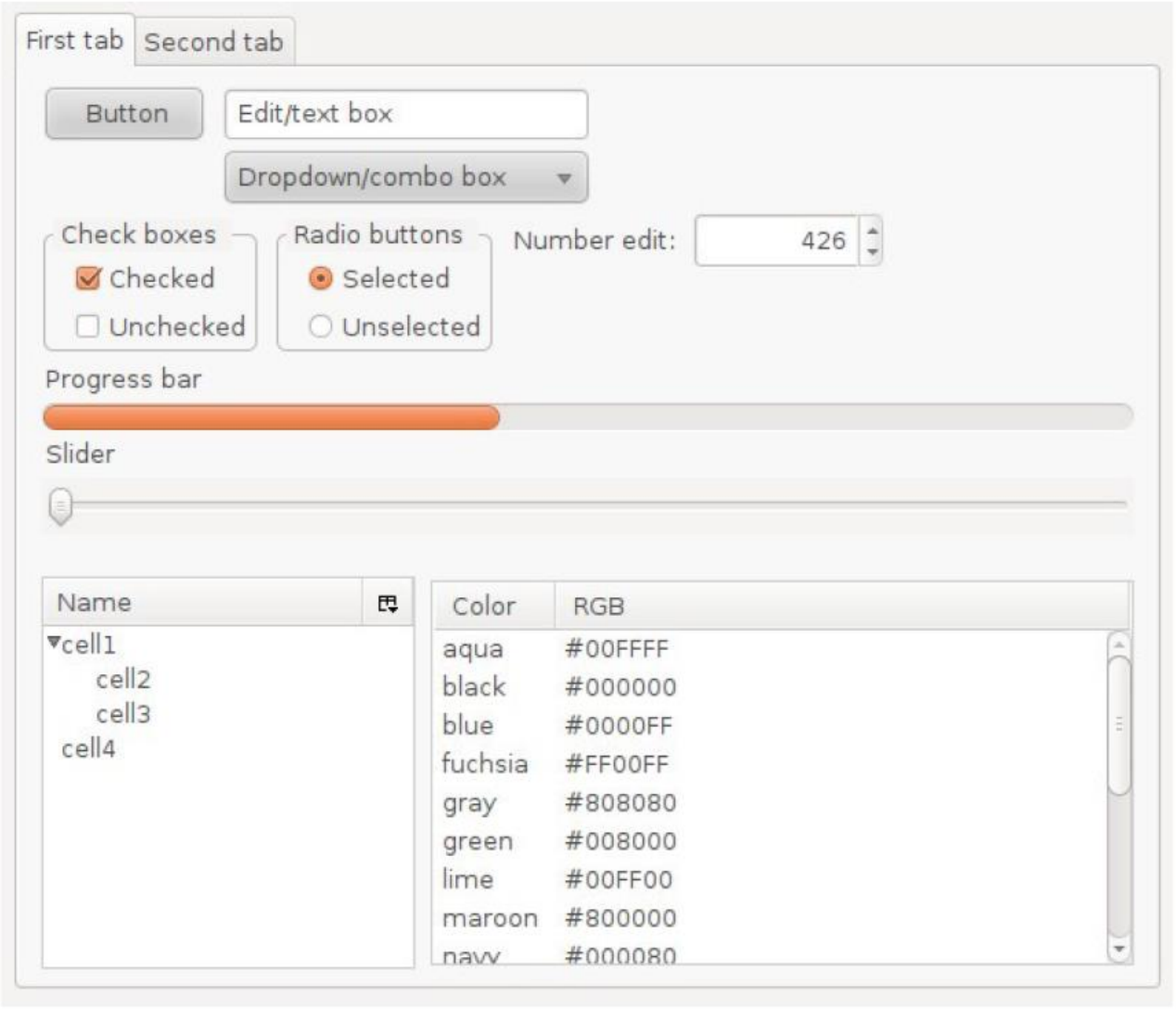
Šírenie správ v stromovej štruktúre

- Vyhodne najma pre systemy kde sa počet a usporiadanie objektov meni (dynamicke hry)
- Strom je vhodna štruktura na organizaciu a reprezentáciu priestoru
- V princípe každá GUI knižnica organizuje ovladacie prvky do stromovej štruktury
- Na baze posielania sprav môžeme vybudovať grafické rozhranie aplikácie

Widget

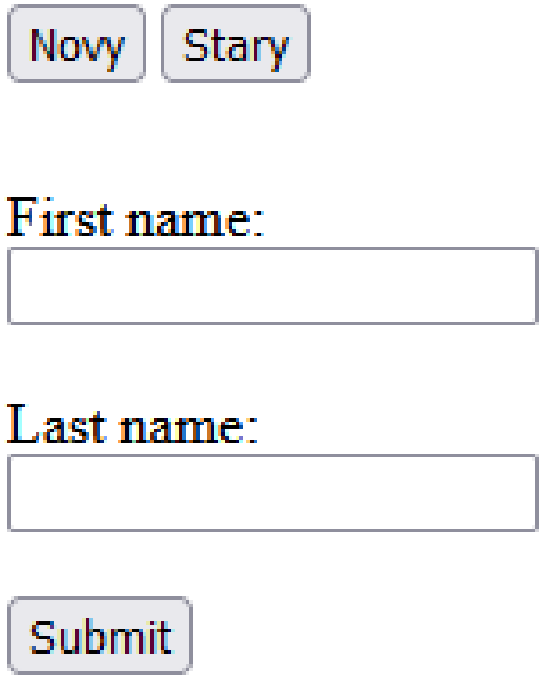
- Objekt reprezentujuci graficky element rozhrania aplikácie
- Typicky napr. button, textfield, menu atd.
- Všetky widgety majú spoločnú časť správania
- Spracúvajú prichádzajúce správy (click, move, key)
- Organizovane do stromov pre vytvorenie GUI

Widget


First tab
Second tab
Button
Edit/text box
Dropdown/combo box
Check boxes
Checked
Unchecked
Radio buttons
Selected
Unselected
Number edit:
426
Progress bar
Slider
Name
cell1
cell2
cell3
cell4
Color
RGB
aqua #00FFFF
black #000000
blue #0000FF
fuchsia #FF00FF
gray #808080
green #008000
lime #00FF00
maroon #800000
navy #000080

Widget - príklad

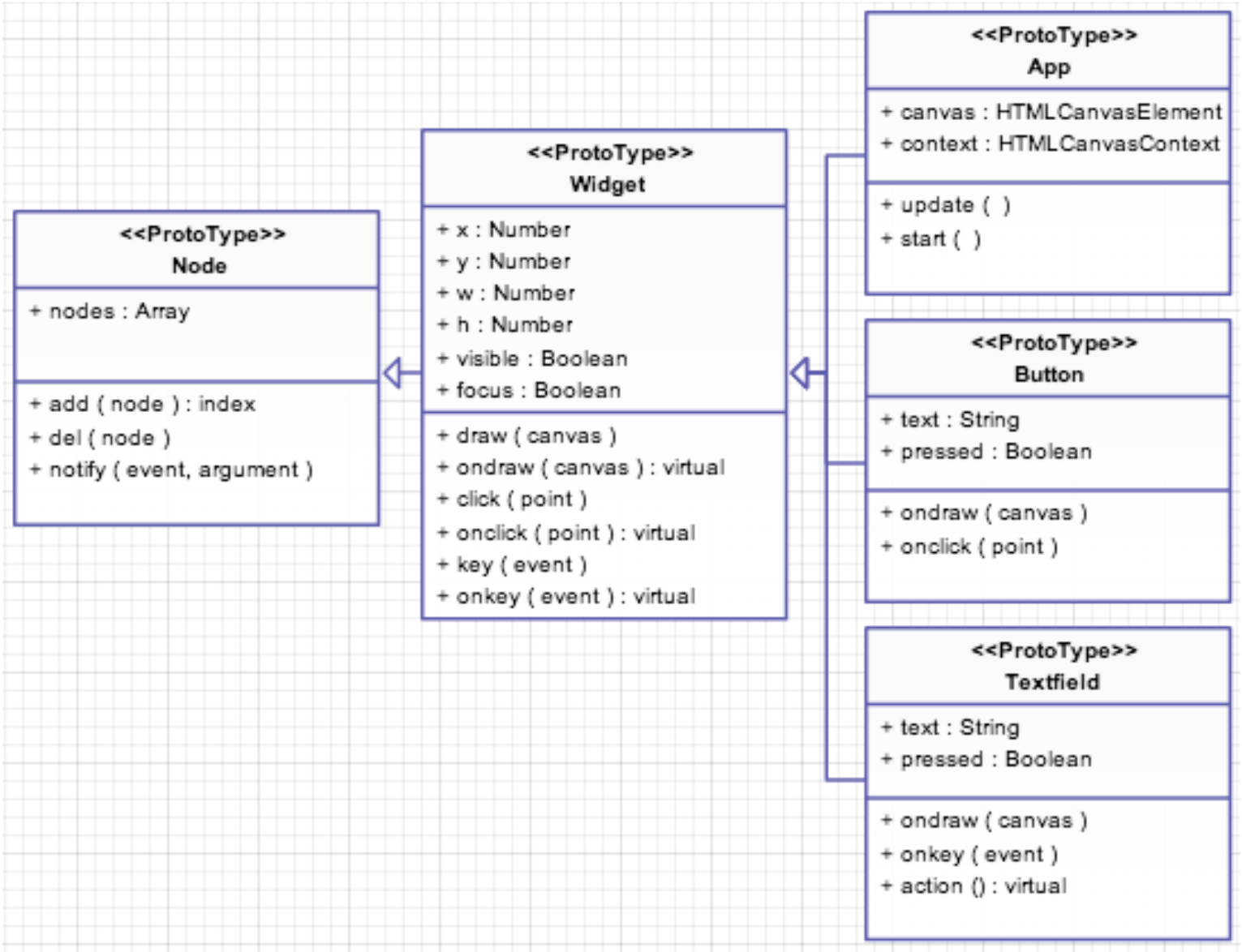
- Demonštruje ako možno aplikovať zasielanie správ na implementáciu GUI knižnice
- Budeme vychádzať z predošlého príkladu a implementácie objektu Node
- Implementujeme jednoduchú GUI knižnicu pre HTML5 canvas
- Knižnica bude obsahovať len ovládacie prvky:
- Button,Textfield a ich prototyp Widget



Widget – príklad implementácie

- Kvoli zjednodušeniu komunikacie je vhodne aby všetky objekty **dedili vlastnosti** z *Node*
- Správanie sa objektov *Button* a *Textfield* je do určitej miery rovnaké a preto ho zovšeobecnime do objektu *Widget*
- *Widget* bude zároveň plniť funkciu zapuzdrenia
- Budeme posielať nasled. správy: *draw, click, key*
- Objekt *Widget* by mal stanoviť pravidla šírenia sa základných sprav

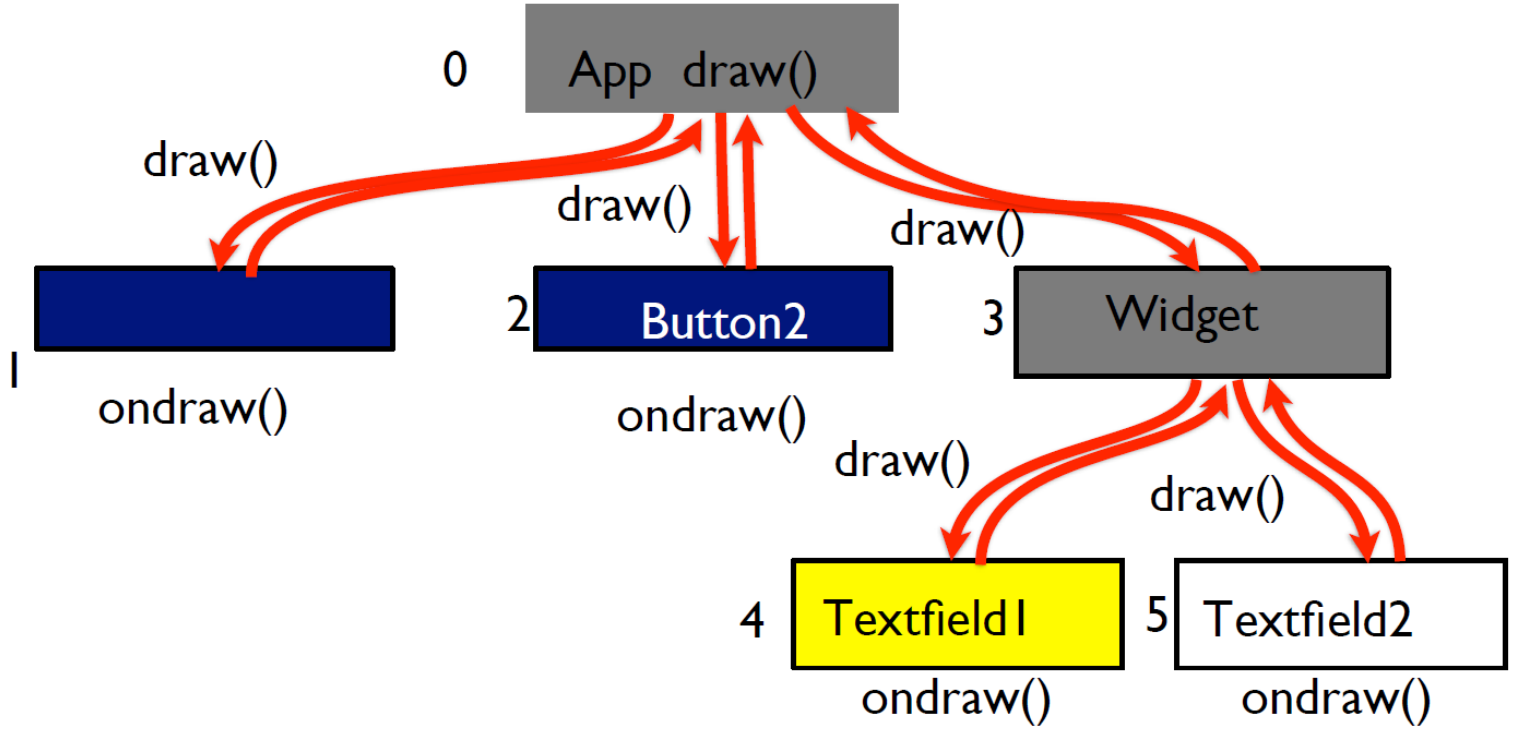
Dedenie vlastností



Widget - príklad

- Implementácia bude vyžadovať šírenie nasledujúcich sprav
- *draw* - sprava o vykreslení
- *click* - sprava o kliknutí myšou
- *key* - sprava o stlačení klávesy
- Všetky správy budú implementovane pomocou volania metód JavaScript objektu

Šírenie správa



```
<!DOCTYPE HTML>
<html>

  <head>
    <title>Widgets example</title>
    <script src="node.js"></script>
    <script src="widget.js"></script>
    <script src="button.js"></script>
    <script src="textfield.js"></script>
    <script src="app.js"></script>
    <script src="main.js"></script>
  </head>

  <body>
    <p>
      <canvas id="canvas" height="800" width="600"></canvas>
    </p>
  </body>

</html>
```

Trieda Node

```
// Object Node
class Node {
  constructor() {
    this.nodes = []      // Nodes to message
  }
  add(node) { this.nodes.push(node) }

  // Remove node
  remove(node) {
    var index = this.nodes.indexOf(node)
    delete this.nodes[index]
  }

  // Notify nodes of event
  notify(event, argument) {
    for (var index in this.nodes) {
      var node = this.nodes[index]
      if (typeof (node[event]) == "function")
        node[event](argument)
    }
  }
}
```

Trieda Widget

```
// Simple Widget implementation
class Widget extends Node {
  constructor(x, y, w, h) {
    // Construct Node
    super();

    this.x = x
    this.y = y
    this.w = w
    this.h = h
    this.rotation = 0
    this.visible = true
    this.focus = false
    this.border = true
  }
// continues on next slide
```

Trieda Widget pokrač.

```
// Drawing widgets using canvas
 draw(context) {
   if (!this.visible) return

   // Each widget contained in its parent
   context.save()

   context.translate(this.x, this.y)
   context.rotate(this.rotation)
   context.beginPath()
   context.rect(0, 0, this.w, this.h)
   context.clip()

   // Draw border
   if (this.border) { ... }

   // Draw
   this.ondraw(context)
   // Send draw event to other Widgets
   this.notify("draw", context)

   context.restore()
 }
 // Widget specific drawing
 ondraw(context) {}

// continues on next slide
```

Trieda Widget pokrač.

```
  // Click handling
  click(point) {
    if (!this.visible) return

    // Point needs to be converted to local coordinates
    var localPoint = {
      x: point.x - this.x,
      y: point.y - this.y
    }

    // Check localPoint is inside Widget boundary
    if (0 < localPoint.x && localPoint.x < this.w)
      if (0 < localPoint.y && localPoint.y < this.h) {
        this.focus = true
        // Call onclick function
        this.onclick(localPoint)
      } else
        this.focus = false

    // Send click event to other Widgets
    this.notify("click", localPoint)
  }
  // Widget specific click
  onclick(point) {}

// continues on next slide
```

Trieda Widget pokrač.

- Od triedy Widget bude dediť triedy *Button* a *Textfield*
- Tieto triedy rozšírime o metódu *action()*

```
// Keyboard handling
 key(key) {
   if (!this.visible) return

   if (this.focus) this.onkey(key)

   // Send key message to other Widgets
   this.notify("key", key)
 }

 onkey(key) {}
}  // class end
```

Trieda Button

```
// Simple Button implementation
class Button extends Widget {
  constructor(text, x, y, w, h) {
    // Construct Widget
    super(x, y, w, h)
    // Button specific
    this.text = text
    this.pressed = false
  }
  // Redefine ondraw function
  ondraw(context) {
    context.fillStyle = "blue"
    if (this.pressed)
      context.fillStyle = "green"
    context.fillRect(0, 0, this.w, this.h)
    context.font = "20px Arial";
    context.fillStyle = "white"
    context.textAlign = 'center';
    context.fillText(this.text, this.w / 2, this.h / 2);
  }
  // Redefine onclick function
  onclick(event) {
    this.pressed = !this.pressed
    if (this.action) return this.action()
  }
  // By default do nothing
  action() {}
}
```

Trieda Textfield

```
class Textfield extends Widget {
  constructor(text, x, y, w, h) {
    super(x, y, w, h) // Construct Widget
    this.text = text   // Textfield specific
  }

  ondraw(context) { ... }

  // Handle keyboard
  onkey(event) {
    var key = event.which
    switch (key) {
      case 8:   // backspace
        this.text = this.text.substring(0, this.text.length - 1);
        break;
      case 13:  // enter
        this.action()
        break;
      default:
        this.text += String.fromCharCode(key)
    }
  }

  // By default do nothing
  action() {}
}
```

Trieda App

- Reprezentuje Controller
- Zabezpečuje kreslenie, spracovanie udalostí a mainLoop

```
class App extends Widget {
  constructor(element) {
    var canvas = window.document.getElementById(element)
    var context = canvas.getContext("2d")
    super(0, 0, canvas.width, canvas.height)
    this.canvas = canvas
    this.context = context
  }
  // Redefine draw
  ondraw(context) {
    context.fillStyle = "gray"
    context.fillRect(0, 0, this.w, this.h)
  }
  // Redraw everything
  update() {
    this.draw(this.context)
  }
// continues on next slide
```

Trieda App pokrač.

```
// Initialize application handlers
 start() {
   var app = this

   // Register mouse handler
   window.onclick = function (event) {
     var point = { x: event.layerX, y: event.layerY, }
     // Send click message
     app.click(point)
   }

   // Register keyb handler
   window.onkeydown = function (event) {
     // Prevent browser from handling backspace key press
     event.cancelBubble = true
     if (event.stopPropagation) event.stopPropagation()
     // Send key message
     app.key(event)
     return false
   }

   // Update 30time per second
   setInterval(function () {
     app.update()
   }, 1000 / 30)
 }
} // class end
```

Súbor *main.js*

- vytvoríme si objekt app z triedy App
- pridáme aplikácie 2 tlačidla (s vlast. impl. metódy action())

```
var app // for easier debug

// Just start our application and add some widgets to it
window.onload = function() {
  app = new App("canvas")

  // Add buttons
  var button1 = new Button("Button1", 10, 10, 100, 50)
  button1.action = function() {
    alert("Button 1")
  }
  app.add(button1)

  var button2 = new Button("Button2", 150, 10, 100, 50)
  button2.action = function() {
    console.log("Button 2")
  }
  app.add(button2)
// continues on next slide
```

Súbor *main.js*

- pridáme *container* (inštancia triedy *Widget* bez zmien) a do neho vložíme 2 textové polia (s vlast. impl. metódy *action()*)

```
  // Add a container
  var container = new Widget(10, 80, 500, 200)
  app.add(container)
    //container.visible = false

  // Add textfields
  var field1 = new Textfield("Textfield1", 10, 10, 300, 50)
  field1.action = function() {
    alert(field1.text)
  }
  container.add(field1)

  var field2 = new Textfield("Textfield2", 10, 80, 300, 50)
  field2.action = function() {
    console.log(field2.text)
  }
  container.add(field2)
// continues on next slide
```

Súbor *main.js*

- *generatedContainer* - obsahuje mriežku vygenerovaných tlačidiel
- pridáme *buttonx* tlačidlo, ktoré vymení celu scénu

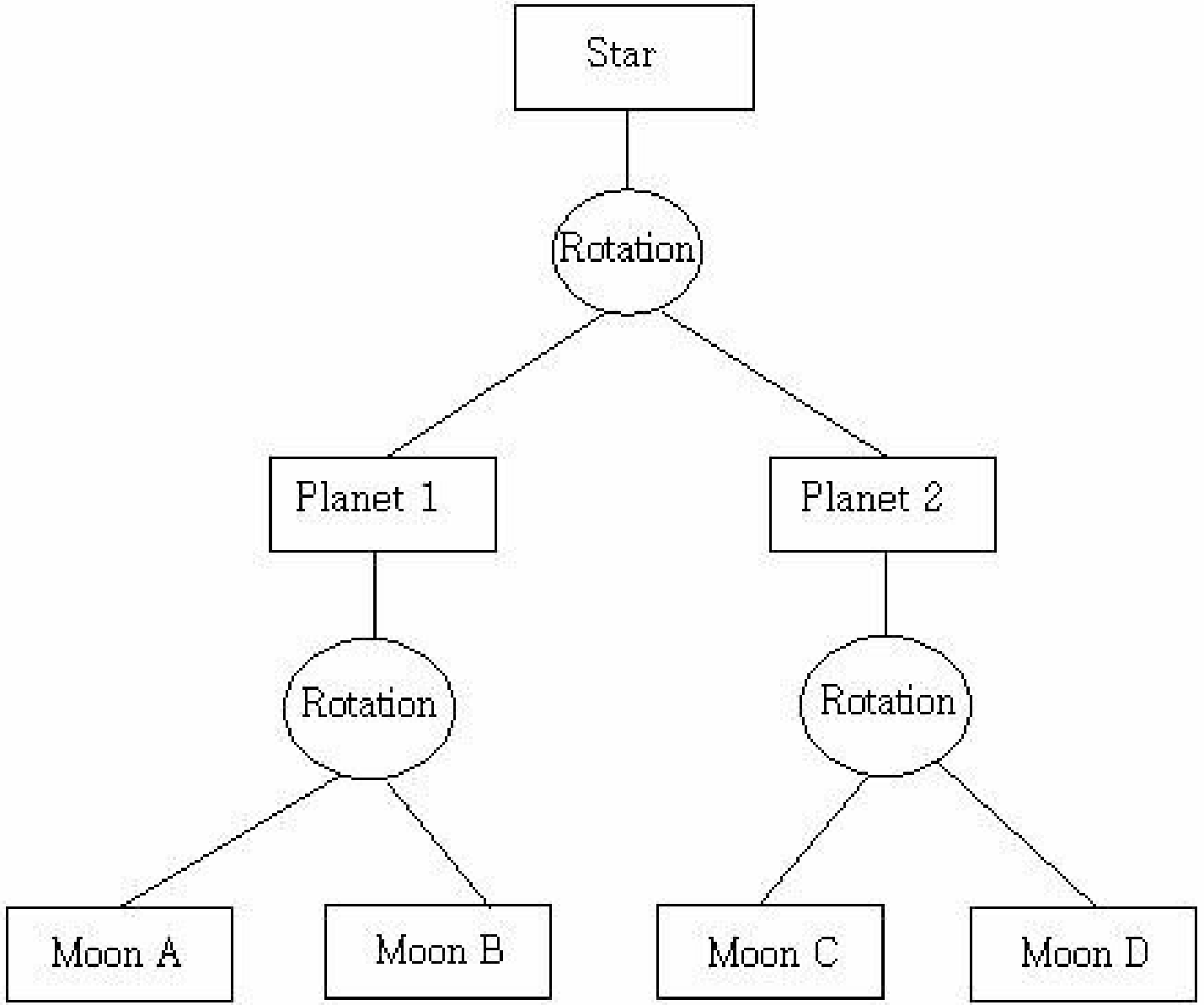
```
  var generatedContainer = new Widget(0, 0, 370, 200)
  for( i=0; i<370; i+=70) {
    for( j=0; j<200; j+=70) {
      var button = new Button(i+","+j, i, j, 50, 50)
      button.action = function() {
        alert("Button: "+i+","+j)
      }
      generatedContainer.add(button)
    }
  }

  var buttonx = new Button("ClearButton", 320, 10, 120, 50)
  buttonx.action = function() {
    field1.text = ""
    app.nodes = generatedContainer.nodes   // replace the whole scene
  }
  container.add(buttonx)

  // Start the application main loop
  app.start()
}
```

Graf scény

- Skoro všetky interaktívne 2D a 3D hry využívajú posielanie sprav medzi objektami
- Objekty sú organizovane v štruktúre, ktorá sa všeobecne nazýva *Graf scény* (Scene Graph)
- Objekty su organizovane do stromov, reps. do grafov, podobne ako v ukažkovej aplikacii
- DOM v prehliadači je *Graf scény*

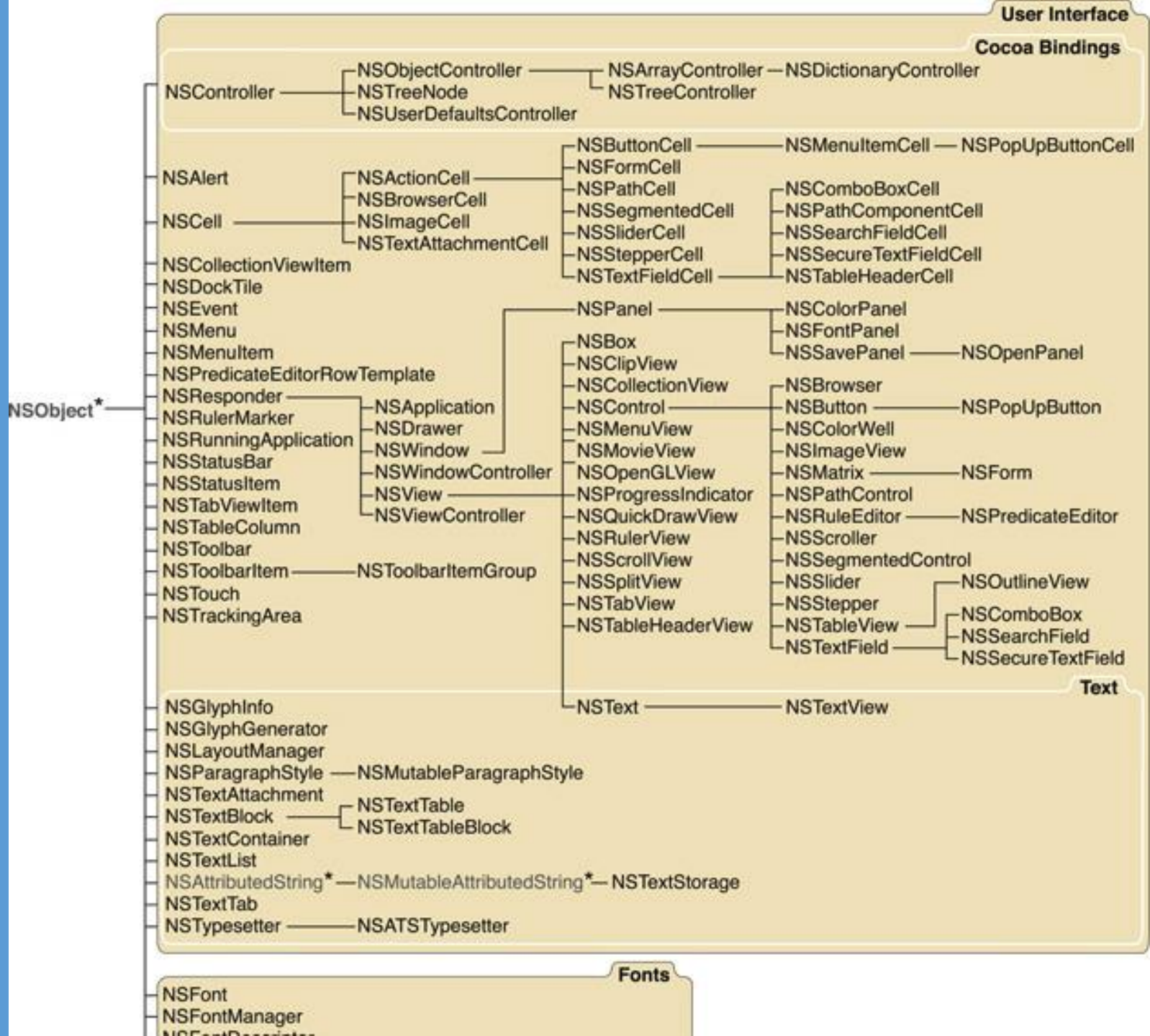
Vieme definovať priestorové závislosti a vytvárať hierarchické závislosti – viac na PPSGO

Graf scény



Reálne knižnice

- https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html



Zhrnutie

- Kľúčové poznatky z prednášky
  - *Observer* umožňuje notifikovať o zmenách stavu iné časti programu
  - Node generalizuje *Observer* a *Subject* a umožňuje organizovať komponenty aplikacie do stromovej štruktúry
    - mnoho výhod, nielen pre šírenie sprav / udalosti
  - Knižnice pre tvorbu GUI
    - Postavene na návrhových vzoroch
    - Komplexne pristupy širenia sprav a reprezentacie sceny