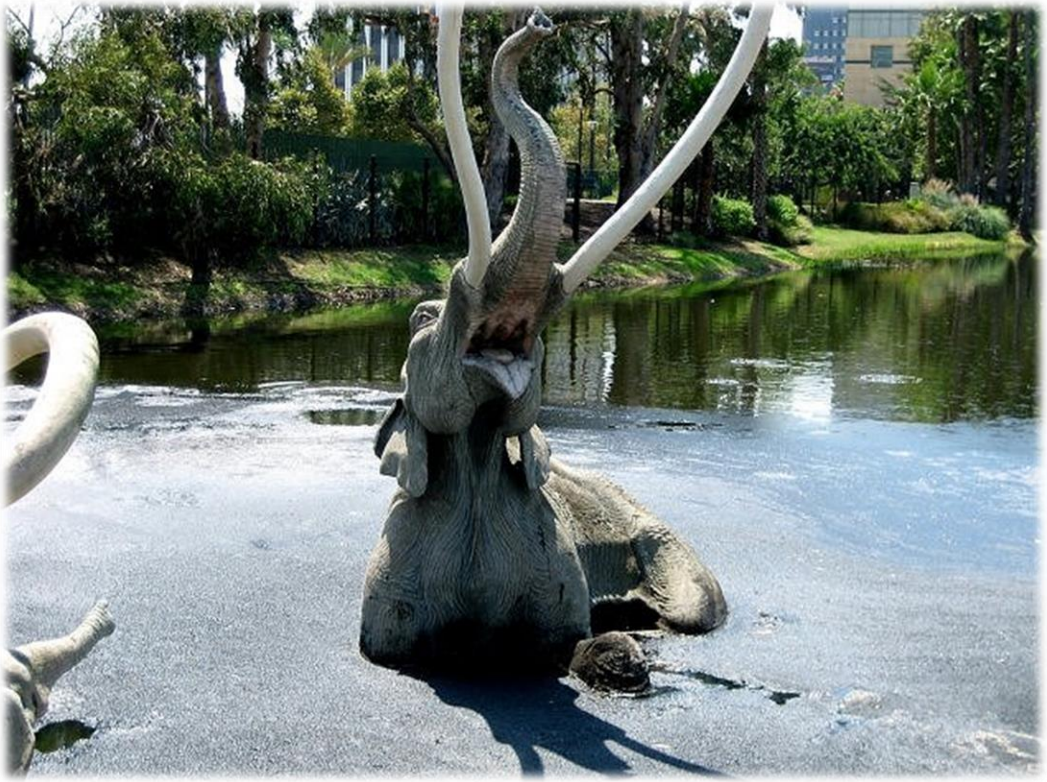
# Zaglavljeni u jami katrana

**Vjeran Marčinko**

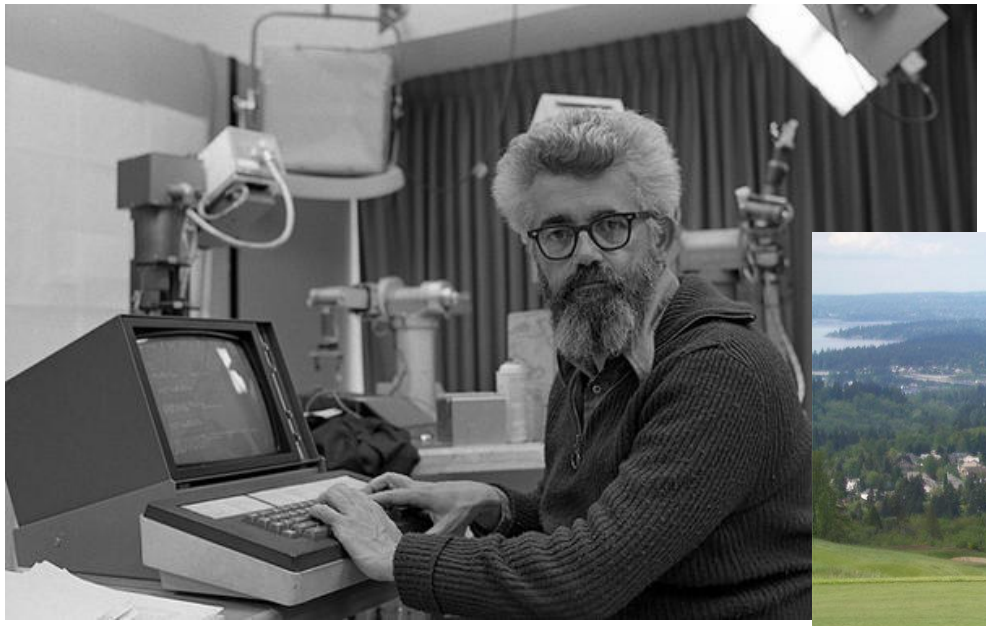
***Kapsch CarrierCom Croatia***



# Dragi dnevniče, bilo je to ljeta gospodnjeg 2012 ...

**JAVA → ?**

# LISP



```
(define (sum-of-squares x y)
    (+ (square x)
    (square y)))
```

**“onaj sa zagradama”**

# JVM

# LISP + JVM ....

CLOJURE !!!

# Malo akademije ...

## Out of the Tar Pit

Ben Moseley
ben@moseley.name

Peter Marks
public@indigomail.net

February 6, 2006

**Abstract**

Complexity is the single major difficulty in the successful development of large-scale software systems. Following Brooks we distinguish *accidental* from *essential* difficulty, but disagree with his premise that most complexity remaining in contemporary systems is essential. We identify common causes of complexity and discuss general approaches which can be taken to eliminate them where they are accidental in nature. To make things more concrete we then give an outline for a potential complexity-minimizing approach based on *functional programming* and *Codd's relational model of data*.

# Problem

"**Kompleksnost** je glavni uzrok većine problema u današnjem softveru. [....] to jednostavno dolazi od činjenice da **razumjevanje sustava je preduvjet za izbjegavanje problema**, a to je upravo ono što kompleksnost uništava."

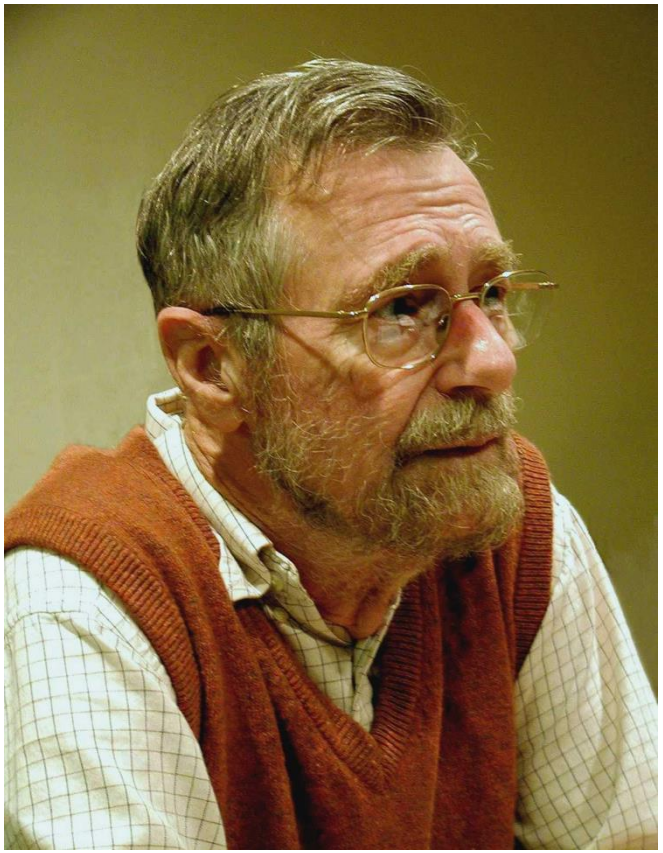
# Razumjevanje sustava

- **Formalno** – specifikacija sustava izvana:
  - ključni problem sa testranjem je da testiranje komponente/sustava sa jednim ulaznim podacima ništa ne govori o ponašanju te komponente kada joj se proslijede drugi ulazni podaci
  - **VIŠE BUGOVA PRONAĐENO**

- **Neformalno** – razumjevanje sustava iznutra (npr. debugiranje)
  - **MANJE BUGOVA NAPRAVLJENO (VAŽNIJE!)**

# Edsger W. Dijkstra (1930-2002)

- “...we have to keep it crisp, disentangled, and simple if we refuse to be crushed by the complexities of our own making...”
- “Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with.”



# Uzroci kompleksnosti

- Stanje
- Kontrola toka

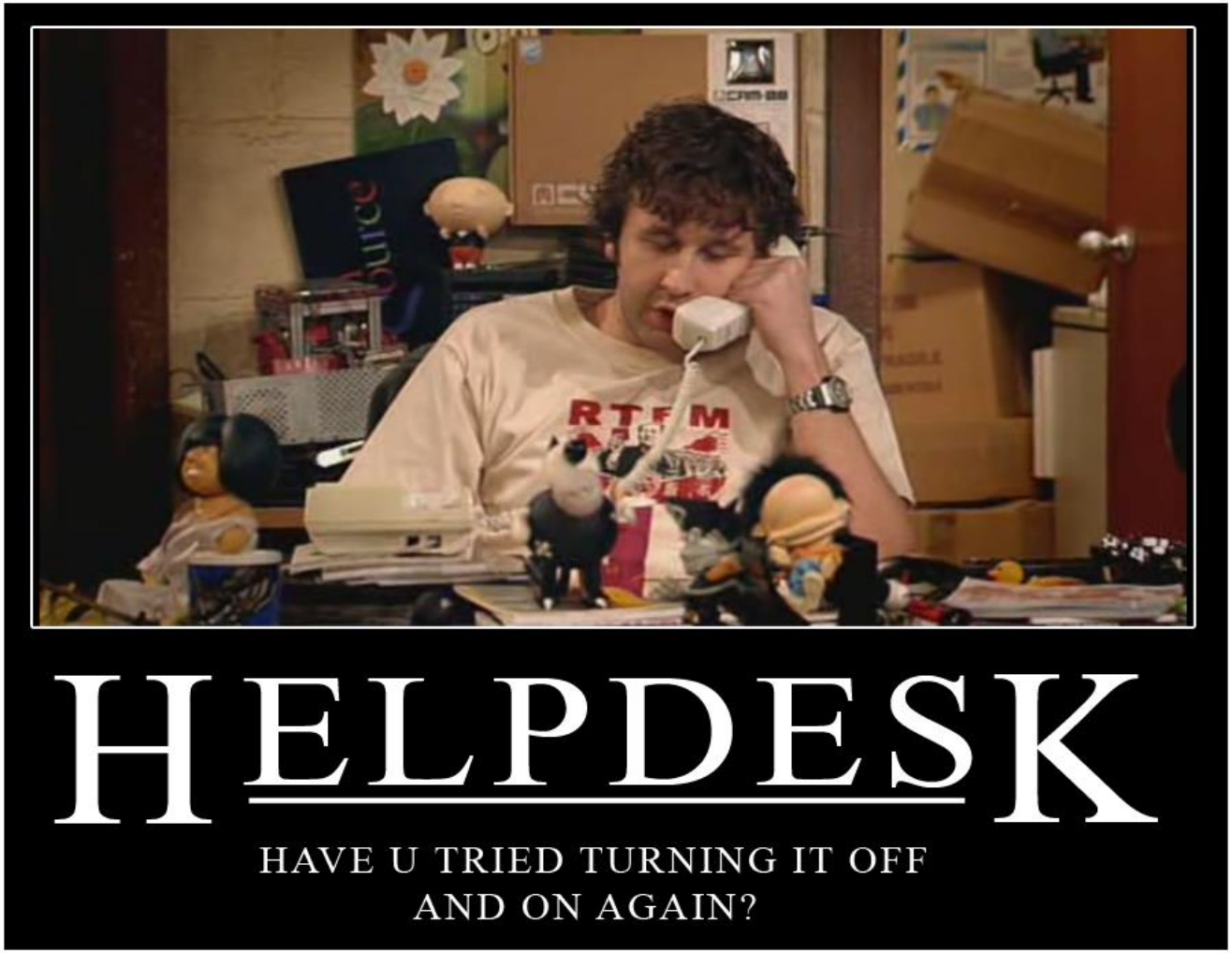
# Stanje i razumjevanje sustava

## Stanje je nužno, ali ga treba maksimalno smanjiti i izolirati

Stanje uništava razumjevanje sustava:

- Formalno (“izvana”) - testiranje sustava u jednom stanju ništa ne govori o sustavu kada je on u drugom stanju
- Neformalno (“iznutra”) - svako novo stanje u programu eksponencijalno  povećava broj mogućih stanja programa → mentalni “overload”

# Workaround za probleme sa stanjem



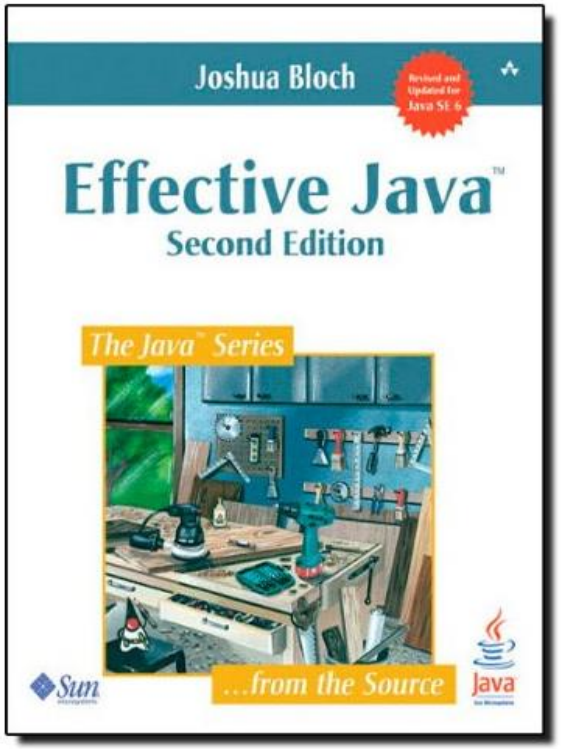
# Stanje i OOP (1)

- Stanje je sakriveno u objektu (private fieldovi)
- Nepredvidivost rezultata poziva metode → ovisi o stanju objekta

```
public class Incrementer {
    private int step; // sakriveno stanje!

    // nepredvidiv rezultat za pozivatelja
    public int increment(int number) {
        return number + step;
    }

    public void doSomething () {
        // promjena stanja ovdje
        this.step = new Random().nextInt(1000);
    }
}
```

# Stanje i OOP (2)

```
Incrementer inc = new Incrementer(1);

// DANGER !!
someDeeplyNestedCall(inc);

// da li je rezultat stvarno 7 ?
int result = inc.increment(6);
```




Promjenjivi objekti se ne mogu sigurno:

- niti primiti, niti proslijediti kao argumenti
- koristiti u hash kolekcijama (Map, Set)
- cachirati
- koristiti u multithreaded okolini

- final keyword (immutability)
- defanzivno kopiranje

# Stanje i FP

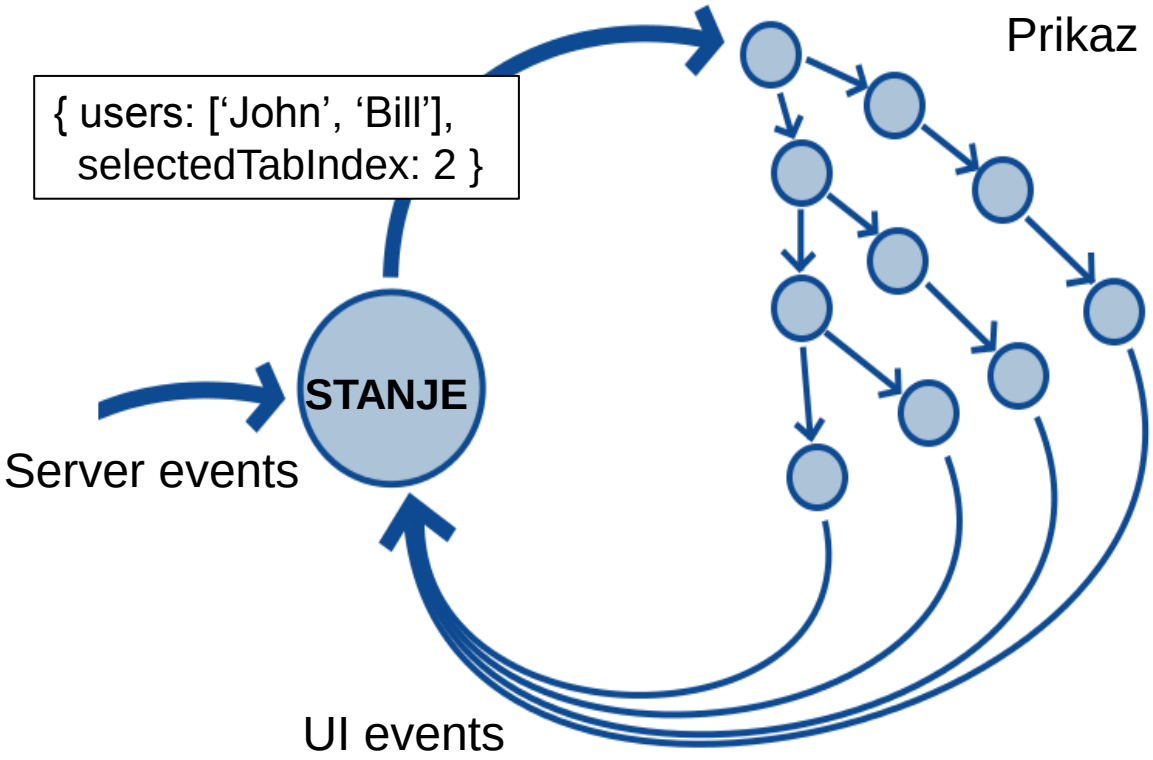
- Referencijalna transparentnost - svi podaci koji određuju rezultat funkcije su vidljivi kao ulazni argumenti

```
public static int increment(int number, int step) {
    return number + step;
}

public static User activateUser(User user) {
    User newUser = new User();
    newUser.name = user.name;
    newUser.active = true;
    return newUser;
}
```

- Stanje još uvijek postoji, ali drži se bliže „površini” i prenosi se „dole” kao argument

# Stanje i Javascript UI

- Problem – DOM je „globalna promjenjiva varijabla” (stanje)
- Stanje bi trebalo biti izolirano (idealan slučaj), a DOM funkcija stanja




Prikaz = DOM = **f** (STANJE)

```
<body>
  <ul>
    <li>John</li>
    <li>Bill</li>
  </ul>
  <div id=” tab2” >...</div>
</body>
```

# React.js

- Programer ne dira pravi DOM - „Virtualni DOM” se rekreira na svaku promjenu stanja (poput server side web developmenta)
- Programer brine samo o sadržaju **cijelog (virtualnog) DOM-a** u danom trenutku
- React.js u pozadini radi **add/update/remove pojedinih elemenata na pravom DOM-u**


Performanse ?

***Trenutak t( i ):***

```
<body>
  <ul>
    <li>John</li>
  </ul>
</body>
```

***Trenutak t( i + 1 ):***

```
<body>
  <ul>
    <li>John</li>
    <li>Bill</li> → „add” akcija
  </ul>
</body>
```

# Kontrola toka

- Red po kojem se stvari događaju – izvor kompleksnosti
- Primjer:

```
int a = 4;
int b = a + 2;
int c = b * 3;
```

- Tok u konkurentnom sustavu?

Kako uopće shvatiti/testirati nešto gdje ne znamo kojim slijedom se stvari događaju?

# Deklarativno programiranje

- „Reci mi što želiš, ne kako” – eliminacija kontrole
- Funkcionalni jezici su više deklarativni (npr. izbjegavanje petlji – funkcije filter, map...)
- Logično programiranje (npr. Prolog) - ultimativni deklarativni jezik
- core.logic library (Clojure)

```
(run* [a b c]
      (* b 3 c)
      (== a 4)
      (+ a 2 b))
=> ([4 6 18])
```

```
(run* [a b c]
      (* b 3 c)
      (membero a [4 5 6])
      (+ a 2 b))
=> ([4 6 18] [5 7 21] [6 8 24])
```


That's all Folks!