# Knjižnice in ogrodja

# Ponovna uporaba programske kode

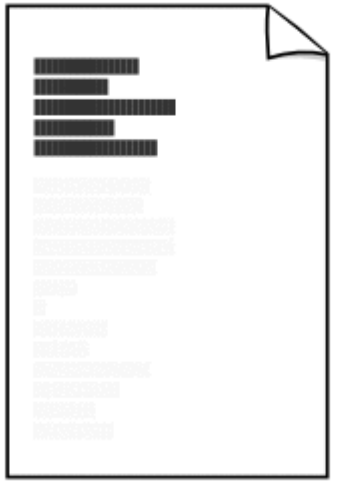
- V programskem inženirstvu je včasih bil poudarek na originalnem razvoju. Danes pa je ponovna uporaba programskih rešitev ne le zaželena, ampak največkrat tudi zahtevana.
  - Zakaj bi reševali izumljali toplo vodo – reševali izzive, ki so jih drugi že rešili, te rešitve pa so že preverjene?

- Včasih s(m)o se programerji hvalili naročniku, da bodo razvili vse iz nič. Danes pa se hvalijo, da pri razvoju uporabijo določene že obstoječe rešitve.

# Ponovna uporaba programske kode

- V inženirskih disciplinah težimo k temu, da se rešitve ponovno uporabijo čim večkrat.
  - Primer: ko ugotovimo kako narediti motor za avto, ga repliciramo in ponovno uporabimo v vsakem vozilu.

- Izkušnje kažejo, da ponovna uporaba programske kode vodi do hitrejšega razvoja, manj napak pri načrtovanju, hitrejšo odpravo napak razvoja. To vse pa privede do cenejšega razvoja.



Original code fragment

# Ponovna uporaba programske kode

- **WET** princip razvoja – “**W**rite **E**very **T**ime” oz. “write everything twice” oz. “we enjoy typing” oz. “waste everyone’s time”.
  - Nezaželeno in pokazatelj slabega razvijalca/razvijalke.

- **DRY** princip razvoja – “**D**on’t **R**epeat **Y**ourself”
  - Je princip razvoja programske opreme, kjer strmimo k zmanjšanju ponavljanja ponovljivih in ponovno uporabnih delov kode.

  - Zato pišemo metode, ker ne želimo programirati enake ukaze vedno znova.
  - Zato pišemo razrede, ker ne želimo programirati enake strukture vedno znova.
  - Zato pišemo knjižnice, ogrodja in komponente…
  - Zato uporabljamo vzorce – *več o tem pri drugem predmetu!*

# Tri vrste ponovne uporabe

1. **Ponovna uporaba *celotnih* sistemov**
   - Celotna aplikacija se ponovno uporabi in se uvedejo le manjše spremembe.
   - Centraliziran razvoj, kar pomeni, da vse izpeljanke dobijo nove funkcionalnosti in popravke istočasno in hitro (saj razvojna ekipa ni razpršena po več produkti).
   - Prilagoditve so omejene – največkrat le kozmetične prilagoditve in vključitev/izključitev funkcionalnosti.

   - Primer: spletne banke, spletne trgovine…

# Tri vrste ponovne uporabe

2. **Ponovna uporaba *večjih* delov sistema**
   - Ponovno se uporabijo le deli sistemov – **skupek funkcij** ali **razredov**.
   - Nove funkcionalnosti in popravki pridejo s posodobitvijo komponent. Razvoj teh največkrat vedno poteka v ločeni razvojni ekipi (ekipa, ki razvija sistem, ne razvija komponente).
   - Prilagoditve aplikacije so seveda mogoče, prilagoditev komponent je še vedno omejena.

   - Primer: uporaba komponente za bančno plačevanje preko spleta, uporaba komponente za izris zemljevida na spletnih straneh…

# Tri vrste ponovne uporabe

3. **Ponovna uporaba *manjših* delov sistema**
   - Ponovno se uporabijo le najmanjši gradniki – **posamezne funkcije** ali **razredi** oz. deli teh.
   - Razvoj novih funkcionalnosti in popravkov največkrat poteka v isti ekipi, ki razvija sistem.
   - Ponovna uporaba tukaj je v obliki knjižnic ali kopiranje kode – možnosti za prilagoditve so velike.

   - Primer: uporaba kode za računanje DDV, delo z datumi, obdelava besedila s šumniki, vizualni efekti na spletnih straneh…

# Prednosti ponovne uporabe

- **Večja zanesljivost** – ponovno uporabljena programska koda, ki je že bila testirana in preizkušena v produkciji, bi naj bila bolj zanesljiva kot na novo napisana koda. Pomanjkljivosti v načrtovanju in implementaciji bi že morale biti odpravljene.

- **Hitrejši razvoj** – ponovna uporabe programske kode omogoča preskok razvoja in testiranja določenega dela sistema. S sabo prinese zmanjšanje stroškov razvoja.

- **Efektivni izkoristek virov** (predvsem zaposlenih) – razvoj enih in istih komponent je zamuden. Naše vire, programerje, testerje, načrtovalce raje uporabimo pri razvoju sistemu specifičnih funkcionalnosti.

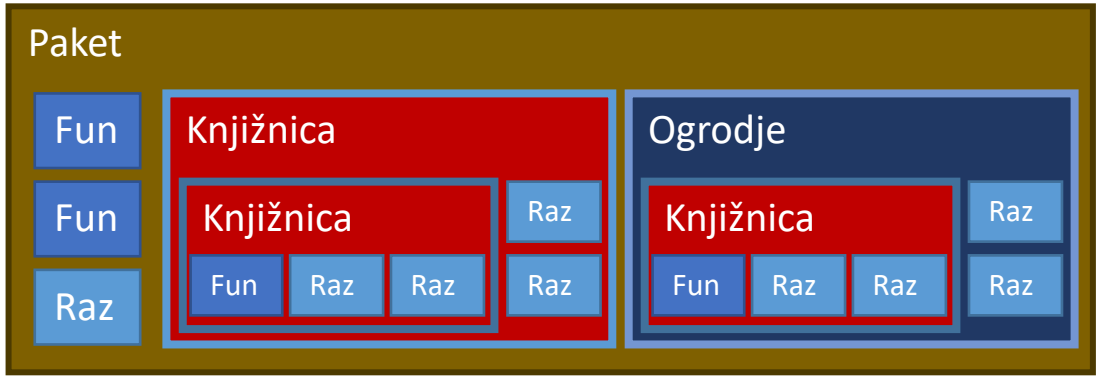
# Slabosti ponovne uporabe

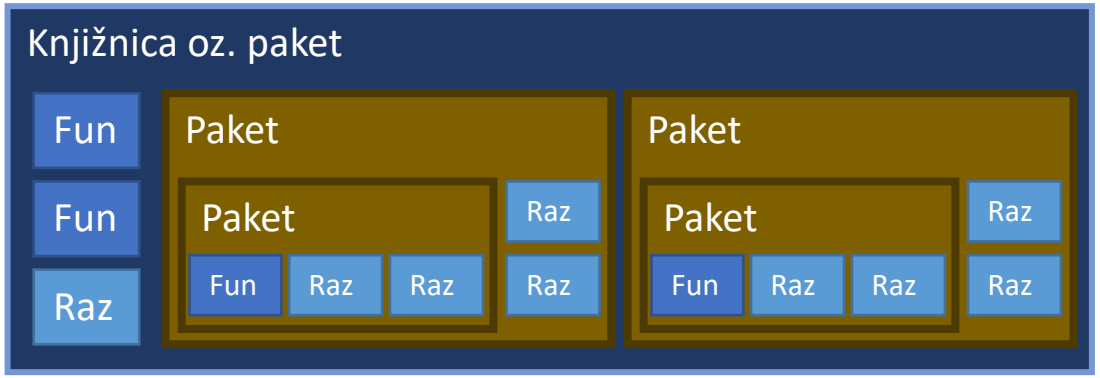
- **Višja sklopljenost** – sistem v razvoju moramo prilagoditi delovanju tiste komponente, ki jo ponovno uporabimo. Včasih to ne zahteva ogromno truda, včasih pa celoten sistem temelji na ponovno uporabljeni kodi.
  - Če smo tekom načrtovanja izbrali neprimerno kodo za ponovno uporabo, se razvoj podaljša in oteži.
  - Če ponovno uporabljena koda ne deluje prav, smo največkrat odvisni od kreatorja, da pomanjkljivosti odpravi.

- **Razpoložljivost** – iskanje primerne programske kode za ponovno uporabi, ki je primerna za naš sistem, zna biti dolgotrajno in težavno.
  - Pri izbiri moramo paziti na zahtevane funkcionalnosti našega sistema vs. ponujene funkcionalnosti že narejene kode, ki jo želimo ponovno uporabiti.
  - Pomembna je tudi ponujena dokumentacija, pomoč izvornih razvijalcev in težavnost učenja uporabe ponovno uporabljene kode.

# Upravljanje s paketi
## *Kaj je paket?*

- V **Pythonu in JavaScriptu** imamo pakete, ki predstavljajo skupek kode.
  - Paketi lahko vsebujejo funkcije in razrede, knjižnice, ter ogrodja.
  - Knjižnice in ogrodja nameščamo preko njihovih paketov.
- V **Javi in C#** pa je paket le ena logična enota znotraj knjižnice.
  - V naših aplikacijah uporabljamo tuje “knjižnice”, ki pa predstavljajo bodisi programsko knjižnico ali ogrodje, so pa sestavljene in enega ali več paketov. Tukaj pa paketi vsebujejo druge pakete ali datoteke razredov.
- Mi govorimo o paketih kot datotekah, ki so skupki kode (vsebujejo knjižnice in ogrodja).



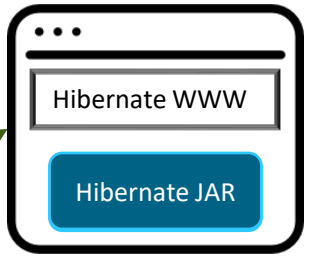
# Upravljanje s paketi
## *Včasih*

**Projekt**

Rabi pakete:
- Spring
- Hibernate
- JSON
- Mail
- Machine Learning

Razvojna ekipa

Ročen prenos iz spletnih strani

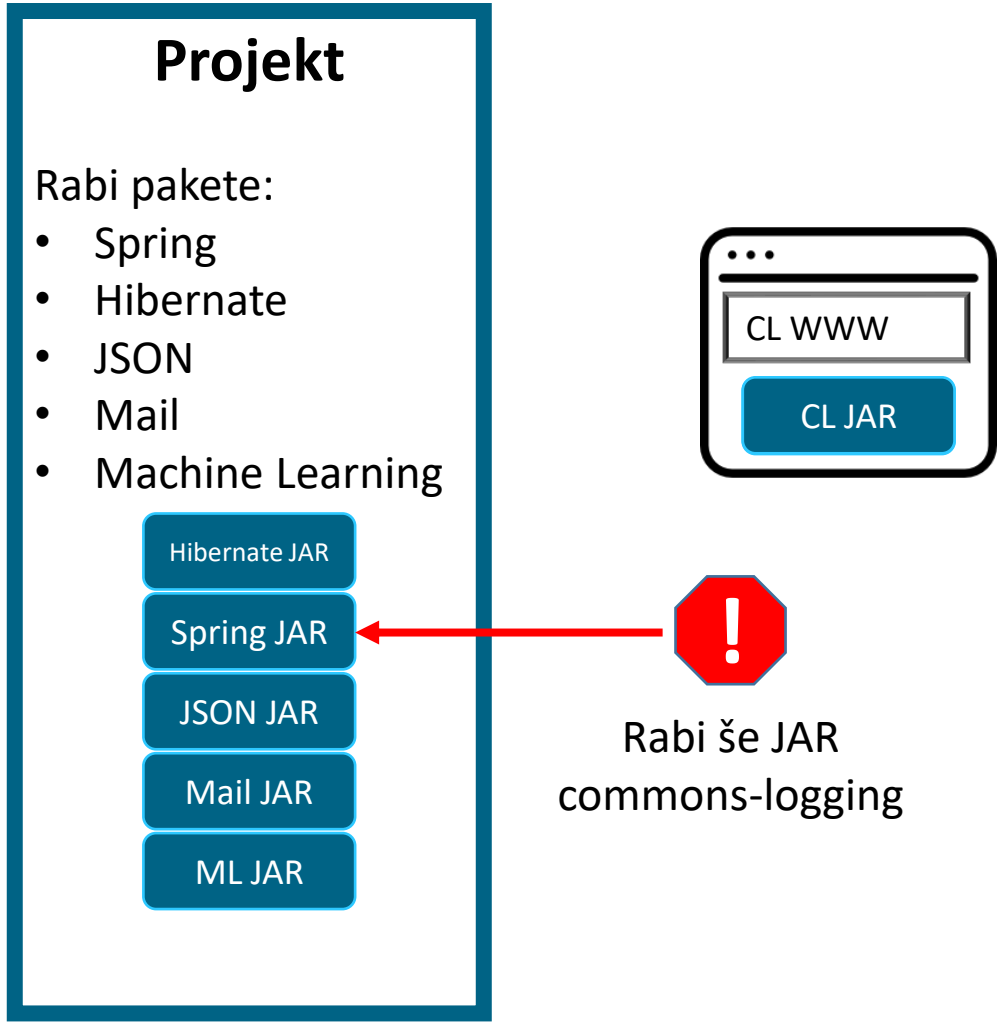






# Upravljanje s paketi
## *Odvisnosti med paketi*

- **Odvisnost** znotraj operacijskega sistema ali programerskega projekta se kaže s vključitvijo paketa, ki ga nismo razvili mi temveč nekdo drug.

- Ne le, da je lahko naš sistem odvisen od tujega paketa, taisti tuji paket je lahko odvisen od drugih paketov.
  - Npr. Java Graph je odvisen od Apache Commons.



# Upravljanje s paketi

- **Upravitelji paketov** (angl. *package managers*) so programi, ki skrbijo za pakete v naših sistemih.
  - Na operacijskem sistemu, ali
  - v naših projektih.

- Povežejo se na **repozitorije paketov** in od tam na naš sistem prenesejo **zahtevane pakete** in vse **njihove podporne pakete**.
  - Pakete lahko nalagajo lastniki ali tretje osebe (nenavadno).

# Upravljanje s paketi
## *Programerskih projektov*

### Nivo sistema

- Ko namestijo paket na naš računalnik, lahko tega uporabimo v poljubnem projektu.
  - **Prednosti**: ni podvajanja paketov; paketi niso del projekta (manjša velikost projekta).
  - **Slabosti**: težavno (ali nemogoče) shraniti več verzij istega paketa; paketi niso del projekta (vsak razvijalec mora te namestiti pri sebi).
- Primer: Pythonov pip, Javin Maven

### Nivo projekta

- Ko namestijo paket na naš računalnik, ga namestijo v mapo projekta.
  - **Prednost**: projekti so prenosljivi (zaradi tega so projekti večji).
  - **Slabosti**: paketi se podvajajo na računalniku; paketi so del projekta (večja velikost projekta).
- Primeri: JavaScriptov npm

# Knjižnica

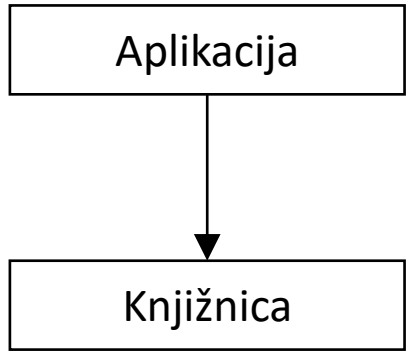
- Definicija 1 (**struktura**): ”Knjižnica je množica razredov, metod in funkcionalnosti, ki so ponovno uporabljivi.”
- Definicija 2 (**namen**): “Knjižnica je kot orodje, ki nam pomaga pri reševanju specifičnih ozko usmerjenih problemov pri razvoju aplikacije.”

- Primeri:
  - **Pandas** (za delo z tabelaričnimi podatki v Pythonu)
  - **jQuery** (za manipulacijo z DOM-om HTML spletnih strani v JavaScriptu)
  - **Seaborn** (za risanje grafov v Pythonu)

# Knjižnica

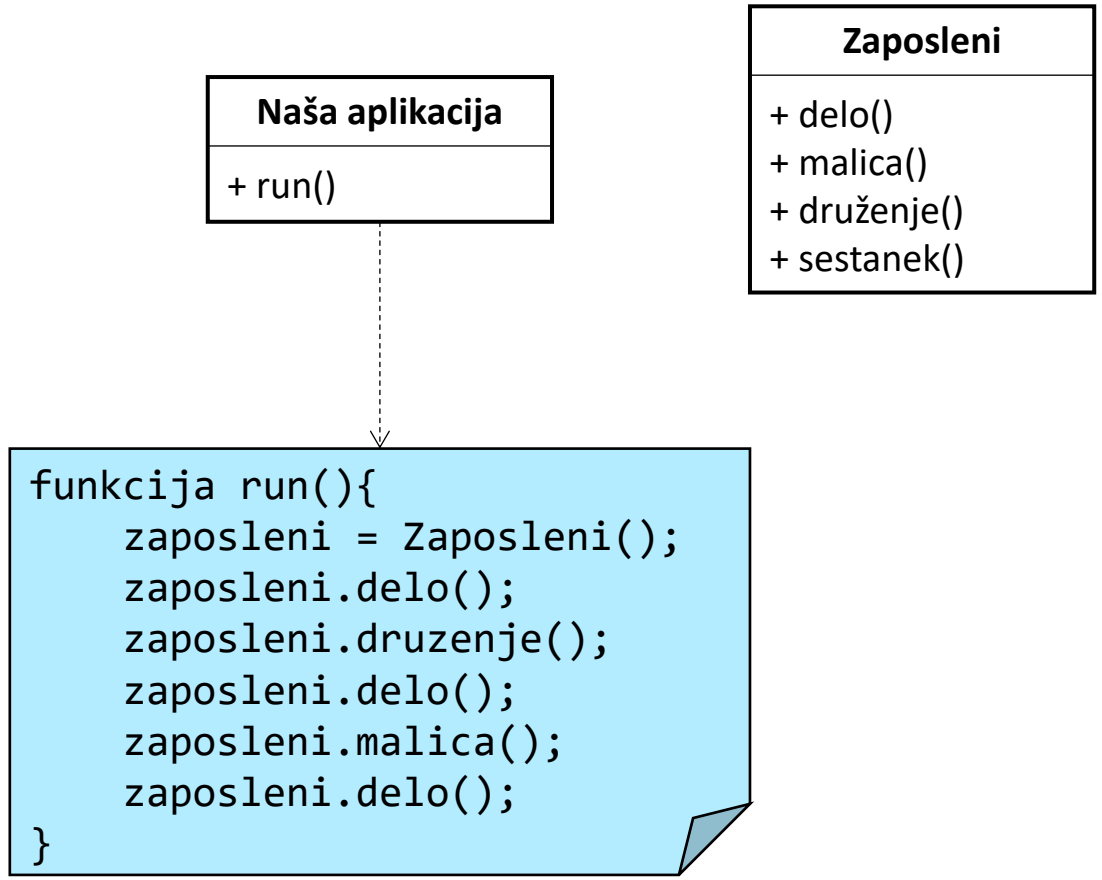
**Knjižnica** (angl. *library*): *"pokliči funkcijo in knjižnica bo naredila vse potrebno."*

- Ponovna uporaba **kode**.
- Hitra vključitev v poljubno aplikacijo.
- Kontrolo nad potekom aplikacije ima razvijalec.
- Odločitev za vključitev sprejmemo v fazi razvijanja.
- Zamenjava knjižnice je praviloma enostavno opravilo.
- Hitro učenje – preberemo dokumentacijo in vemo uporabiti.



# Delovanje knjižnic

- Imamo enostavno knjižnico v obliki ene Java datoteke z razredom `Zaposleni`.
- Naša aplikacija uporablja ponujeno funkcionalnost.



# Ogrodje

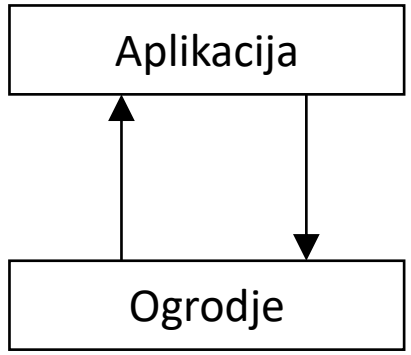
- Definicija 1 (**struktura**): ”Ogrodje je množica razredov, metod in funkcionalnosti, ki so ponovno uporabljivi in *prilagodljivi* z dodatno programsko kodo razvijalca.”
- Definicija 2 (**namen**): “Ogrodje je kot okostje aplikacije, ki se ga lahko prilagodi po željah razvijalcev.”

- Primeri:
  - **Express.js** (back-end storitve v ogrodju Node.js)
  - **React** (pročelne/front-end spletnih aplikacij v JS ali TypeScriptu)
  - **Bootstrap** (izgled spletne aplikacije v CSS-u)

# Ogrodje

**Ogrodje** (angl. *framework*): *“ne kliči nas, mi bomo poklicali tebe.”*

- Ponovna uporaba **kode** in **arhitekture**.
- Aplikacije (ali deli teh) temeljijo na ogrodjih.
- Kontrolo nad potekom aplikacije ima ogrodje.
- Odločitev za vključitev teh sprejmemo v fazi načrtovanja.
- Učenje je dolgotrajno  in posledično je znanje teh tudi bolj cenjeno.
- Zamenjava ogrodja je skorajda nemogoča.



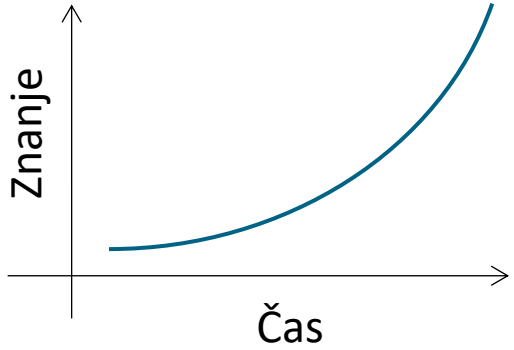
# Prednosti in slabosti dela z ogrodji

## Prednosti

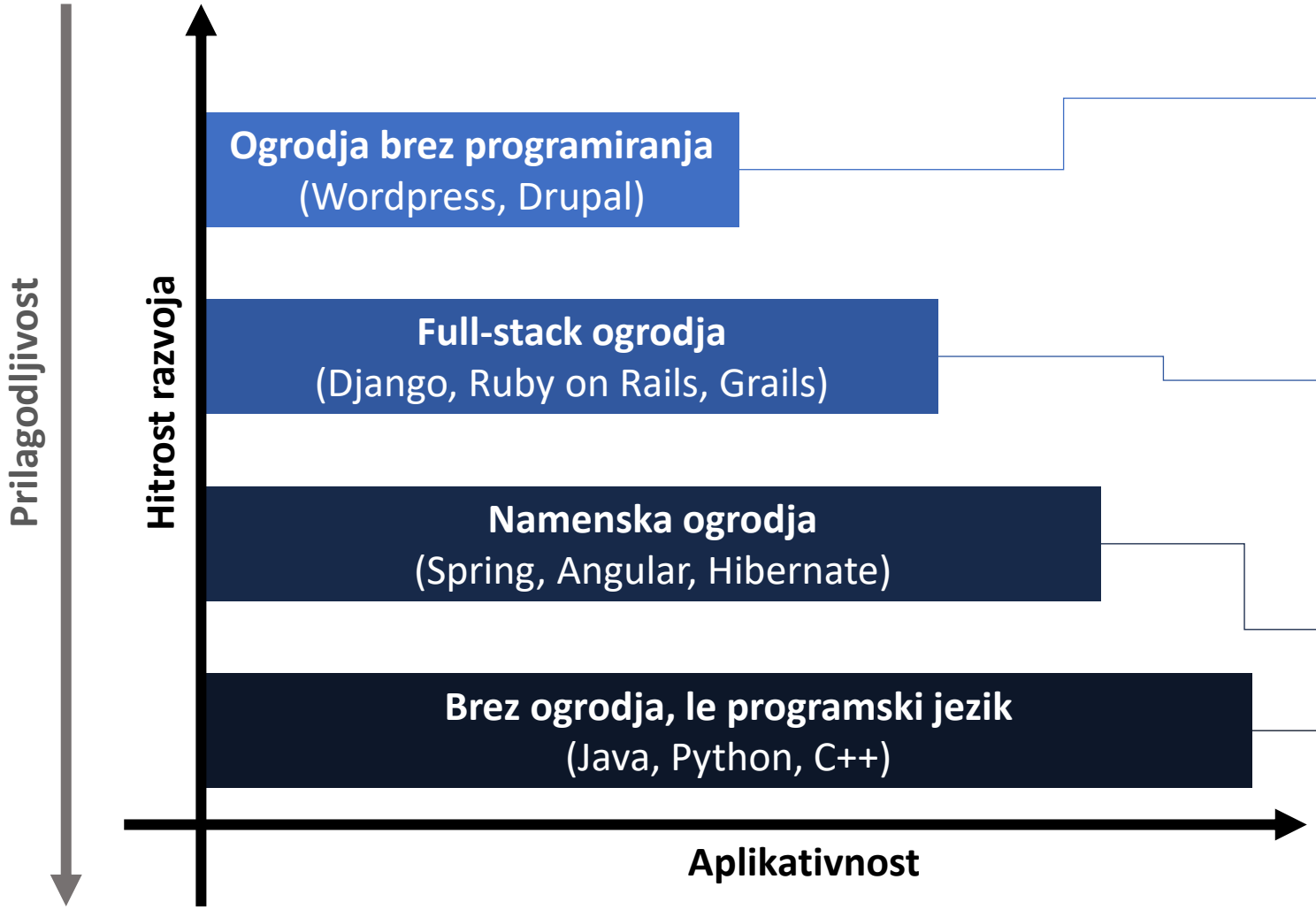
- Ko enkrat obvladamo ogrodje, je delo na novem projektu s taistim ogrodjem lažje in hitrejše.
- Razvoj in rešitev pogostih funkcionalnosti ni potreben – lahko se osredotočimo na razvoj projektu specifičnih funkcionalnosti.
  - Primer: pri namiznih aplikacijah se ne ukvarjamo z razvojem izrisovanja gumbkov na ekran.

## Slabosti

- Ogrodja največkrat vsebujejo podporo za funkcionalnosti, ki jih ne potrebujejo v naši aplikaciji. Če je tega preveč, so ogrodja prenapihnjena (angl. *bloated*).
- Pri prvi seznanitvi z ogrodjem je potrebno veliko časa in truda, da se naučimo in obvladujemo vse specifike ogrodij.



# Nivoji ogrodij



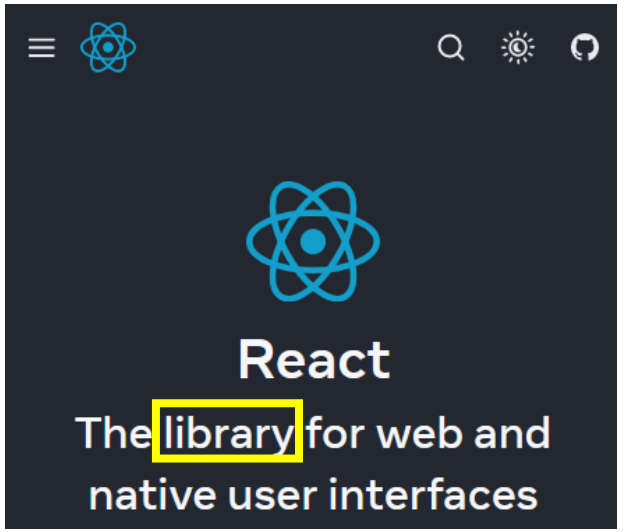

- Ni upravljanja programske kode
- Otežen skupinski razvoj
- Slaba (ali nemogoča) prilagoditev
- Najhitrejša vzpostavitev osnovnega sistema

- Enostavnejše učenje
- Osnovne funkcionalnosti že v paketu (prijava, registracija…)
- Težja integracija z drugimi ogrodji
- Slabša prilagodljivost

- Enostavna integracija z drugimi ogrodji
- Različni razvijalci
- Oteženo posodabljanje v produkciji

- Ogromno programske kode
- Težavno učenje
- Težavno vzdrževanje

# Kaj pa je React?




- Kreatorji React imenujejo “knjižnica”, ker:
  - *…React itself does not include many of the React-specific libraries you're going to need for most projects. Angular and Vue, by comparison, include many more tools all bundled within the core package itself…*
  - *…In other words, because React is a library and not a framework, becoming a skilled React developer entails having a good knowledge of third-party React libraries…*

- Po “pravi” definiciji, pa je React še vedno **namensko ogrodje**, ki pa ne vsebuje vseh gradnikov za celostno izgradnjo aplikacij in ne vsiljuje načina razvoja.

# Dva tipa ogrodij

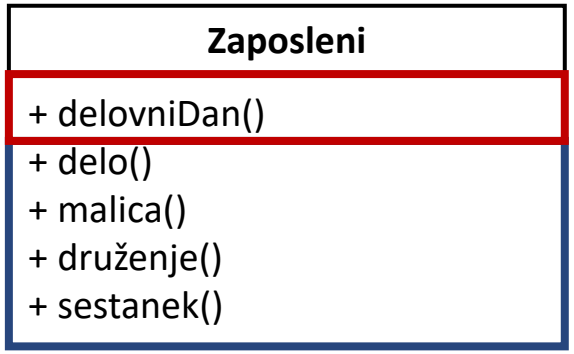
- **Ogrodja bele škatle** – v večji meri se uporablja dedovanje in dinamično povezovanje. Razširljivost takega ogrodja je zagotovljena z *dedovanjem razredov* ogrodja, potem pa z *implementacijo komponent po metodi kavlja*.

- **Ogrodja črne škatle** – uporablja vnaprej definirano strukturo komponent. Razširljivost takega orodja je zagotovljena z *implementacijo komponent po vnaprej definirani strukturi* in *integracijo* teh v ogrodje.

- Ogrodja so lahko tudi obeh tipov hkrati.

# Ogrodje bele škatle

- Ogrodje **bele škatle** ima dva mehanizma:
  - Šablona (angl. *template*) je glavna metoda ogrodja, ki definira obnašanje.
  - Kavelj (angl. *hook*) je prazna metoda, z implementacijo katere mi prilagodimo delovanje.




- Ogrodje bele škatle uporabimo (ali jo sami implementiramo), ko želimo čim večjo prilagodljivost delovanja. Naša šablona je le predlog in z njo spodbujamo, dajo razvijalci uporabijo na kar se da različne načine.

OGRODJE

**Zaposleni**

+ delovniDan()
+ delo()
+ malica()
+ druženje()
+ sestanek()

```
funkcija delovniDan(){
    delo();
    druženje();
    delo();
    malica();
    delo();
}
```

IMPLEMENTIRAMO MI

**Naša aplikacija**

+ run()

```
funkcija run(){
    zaposleni = Programer();
    zaposleni.delovniDan();
}
```

**Oblikovalec**

+ delo()
+ druženje()

**Programer**

+ delo()
+ malica()

```
funkcija delo(){
    programiraj();
}

funkcija malica(){
    pizza();
}
```

**Tester**

+ delo()
+ sestanek()

**Vodja**

+ delo()
+ malica()

```
funkcija delo(){
    // TODO
}

funkcija malica(){
    kaviar();
}
```

# Primer ogrodij bele škatle

## Novo okno v Android aplikaciji

```
import ...AppCompatActivity
import android.os.Bundle


class MojeOkno : AppCompatActivity() {

    override fun onCreate(state: Bundle?) {
        super.onCreate(state)
        setContentView(R.layout.main)
    }

}
```

Dedujemo razred, ki ga bomo razširili

Kavelj – prepišemo z našim delovanje.

## Testiranje v Pythonu z unittest

```
import unittest

def add_numbers(a, b):
  return a + b

class TestAddNumbers(unittest.TestCase):
  def test_add(self):
    self.assertEqual(add_numbers(1, 2), 3)

if __name__ == '__main__':
  unittest.main()
```

Dedujemo razred, ki ga bomo razširili

Kavelj – prepišemo z našim delovanje.

# Ogrodje črne škatle

- Tudi ogrodje **črne škatle** ima dva mehanizma:
  - Šablona (angl. *template*) je glavna metoda ogrodja, ki definira obnašanje.
  - Register (metoda ali kak drugi mehanizem) kjer registriramo, našo funkcijo ali razred, ki bo uporabljen v šabloni.


- Ogrodje črne škatle uporabimo (ali jo sami implementiramo), ko želimo pustiti osnovno delovanje vedno enako.
  - Razlogi za to so lahko: velika kompleksnost delovanja, boljša varnost, boljša optimizacija/hitrost delovanja…

OGRODJE

**Zaposleni**

+ delovniDan()
+ setDelo()
+ setMalica()
+ setDruženje()
+ setSestanek()

```
funkcija delovniDan(){
    delo();
    druženje();
    delo();
    malica();
    delo();
}
```

IMPLEMENTIRAMO MI

**Naša aplikacija**

+ delo()
+ malica()
+ run()

```
funkcija delo(){
    programiraj();
}
```

```
funkcija malica(){
    pizza();
}
```

**BrezvezenSestanek**

+ soba
+ čas
+ sodelujoči
+ govori()

```
funkcija run(){
    zaposleni = Zaposleni();
    zaposleni.setDelo(delo);
    zaposleni.setMalica(malica);
    zaposleni.setSestanek(BrezvezenSestanek);

    zaposleni.delovniDan();
}
```

# Primeri ogrodij črne škatle

## Back-end v Pythonu z Flask

```
from flask import Flask

app = Flask(__name__)

@app.route('/crna')
def domov():
  return 'Črna škatla!'

@app.route('/zdravo/<ime>')
def pozdrav(name):
  return f'Pozdravljen, {ime}!'

if __name__ == '__main__':
  app.run(debug=True)
```

Z anotacijami @ metod registriramo metode, ki bodo poklicane, ko bodo potrebne.

## Back-end v JS z Express.js

```
const express = require('express');
const app = express();

app.get('/crna', (req, res) => {
  res.send('Črna škatla!');
});

app.listen(3000);
```

S klicem metode **get** našo metodo registriramo, da bo poklicana, ko bo potrebna.

Implementacija po vnaprej definirani strukturi (kaj metoda prejme in kaj vrne)

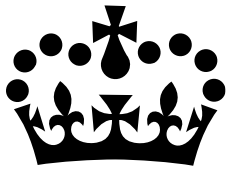
# Primer naloge

Naredite ogrodje za hrano, kjer želimo po standardnem postopku definirati kako operiramo s hrano:

1. Skuhamo hrano
2. Jemo hrano

- V poljubnem psevdokodu ali razrednem diagramu naredite razred `Hrana`, ki ima metodo `pojej()`.
- V poljubnem psevdokodu ali razrednem diagramu prikažite kako bi uporabili to ogrodje za `AzijskoHrano` (ki se kuhajo v voku in jedo s palčkami), `TekočoHrano` (ki se kuha v loncu in se je z žlico) in `DunajskiZrezek` (ki se kuha v ponvi in je z vilico in nožem).
- Kateri tip ogrodja bi uporabili in zakaj?

# Pazljivo pri pretirani uporabi (spletnih) ogrodij



GOV.UK
Service manual

If you use JavaScript (framework), it should only be used to enhance the HTML and CSS so users can still use the service if the JavaScript fails.

Do not build your service as a single-page application (SPA). This is where the loading of pages within your service is handled by JavaScript, rather than the browser.