infoart
inovacija. znanje. tehnologija.

Automatizirano funkcijsko testiranje ADF aplikacija

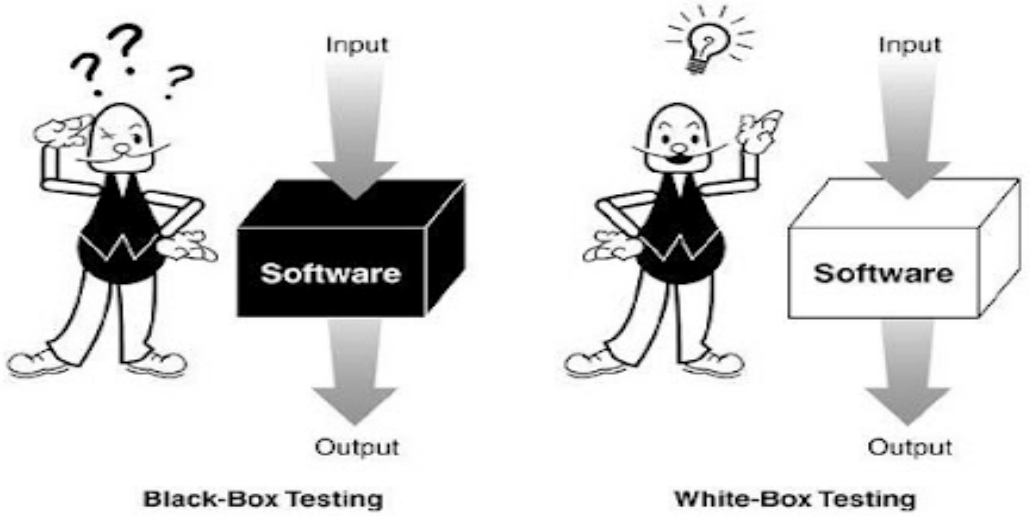
Mirna Katičić, Infoart d.o.o.

Rovinj, 16.10.2014.

Sadržaj

- Uvod
- Motivacija i ciljevi
- Izazovi u priremi testnog modula
- Dizajniranje preglednih i održivih testova - smjernice
- Osnovne komponente testnog modula
  - ConnectFixture
  - Fluent builder pattern
    - EntityBuilder
    - ConfigBuilder
  - Builder fixtures
- Q&A

Funkcijski / integracijski testovi



- Vrsta testiranja na principu "crne kutije"
  - Ne zanima nas trenutna implementacija pojedinih dijelova sustava
  - Testni slučajevi se baziraju na specifikaciji ponašanja sustava kojeg testiramo (SUT-a)
- "white box testing" - testiranje strukture

infoart

KONTINUIRANI RAZVOJ I ODRŽAVANJE POSLOVNIH APLIKACIJA

POSiA*Sustav upravljanja robnim tokovima

FiKs*Sustav upravljanja financijama i računovodstvom

HRiDs*Sustav upravljanja ljudskim resursima

Motivacija za unapređenje procesa testiranja

Obračun plaća

- **Izmjene u centralnom dijelu obračuna utječu na sve korisnike**
  - Proširenje funkcionalnosti za novog korisnika zahtijeva izniman oprez
- **Mnoštvo parametara**
  - Na obračun jednog primitka utječe oko 100 različitih parametara razmještenih na nekoliko razina (osoba, org. jedinica, radno mjesto, vrsta primitka, obrada...)
  - Netrivijalna priprema testne okoline
- **Učestale izmjene zakonskih regulativa**
  - Stalne izmjene/nadogradnje aplikacije su zagarantirane i bez novih korisničkih zahtjeva
  - Nadogradnje ove vrste se obično moraju isporučiti žurno
  - Na raspolaganju je kratak period za kvalitetno testiranje

Automatizacija testnog procesa - ciljevi

Kratkoročni ciljevi:

- Identifikacija i prevencija grešaka u što ranijoj fazi razvoja
- Sprečavanje preranog puštanja u produkciju
- Procjena usklađenosti sa specifikacijama (zakonskim, korisničkim)
- Definiranje uobičajenog načina

  korištenja programa i njegovih komponenti

Dugoročni ciljevi:

- Podizanje kvalitete aplikacije
- Zadovoljstvo korisnika
- Smanjenje troškova održavanja

Dodana vrijednost

Sistematizirano dokumentiranje funkcionalnosti aplikacije

- dobro dizajnirani testovi mogu biti vrlo koristan pomagač u opisu očekivanog ponašanja sustava

Mogućnost odgođenog uništavanja testnog okruženja – pregled testnih podataka kroz aplikaciju

- priprema testnog okruženja je bitan i često najzahtjevniji korak u samom procesu testiranja (bilo ručnog ili automatiziranog)
- Uz opciju "odgođenog" uništavanja testnog okruženja moguće je rezultate automatski generiranih testova pogledati kroz samu aplikaciju što je vrlo korisno u samoj fazi pisanja testova, ali i kasnije za razumijevanje, debugiranje i sl.

Osnovne opcije za pokretanje testova

- svih testova od jednom – npr. jednom nekad u noći
- određenog podskupa – kada nemamo vremena zavrtiti sve

JUnit – vrlo kratak pregled

Set up and clean up metode

`@BeforeClass, @AfterClass` - izvršit će se samo jednom za testnu klasu

`@Before, @After` - izvršit će jednom za svaki test (metodu)

Test runners

```
@RunWith(Suite.class)
@RunWith (Parameterized.class)
@RunWith (Categories.class)
```

Rules

```
@Rule  public TestName testNameRule = new TestName();
@Rule  public Timeout globalTimeout = new Timeout(60000);
@Rule TemporaryFolder
```

Ostalo

```
@Test(expected = ArithmeticException.class)
@Ignore
```

Izazovi u pripremi testnog modula

IZAZOV

- Svi (smisleni) testovi usko vezani uz bazu
  - Osigurati kontrolirano generiranje i "čišćenje" podataka
- Velik broj parametara
  - Osmisliti učinkovit način izgradnje početnog stanja
- Velik broj testnih kombinacija
  - Osmisliti učinkovit način sistematizacije podataka i testova
- Dizajniranje održivih i preglednih testova

Izazovi u pripremi testnog modula

IZAZOV **RJEŠENJE**

- Svi (smisleni) testovi usko vezani uz bazu
  - Osigurati kontrolirano generiranje i "čišćenje" podataka → *ConnectFixture*
- Kreiranje objekata sa velikim brojem parametara
  - Osmisliti učinkovit način izgradnje početnog stanja → *Fluent Builder pattern*
- Velik broj testnih kombinacija
  - Osmisliti učinkovit način sistematizacije podataka i testova → *Builder fixtures*

  Test suites

- Dizajniranje održivih i preglednih testova →

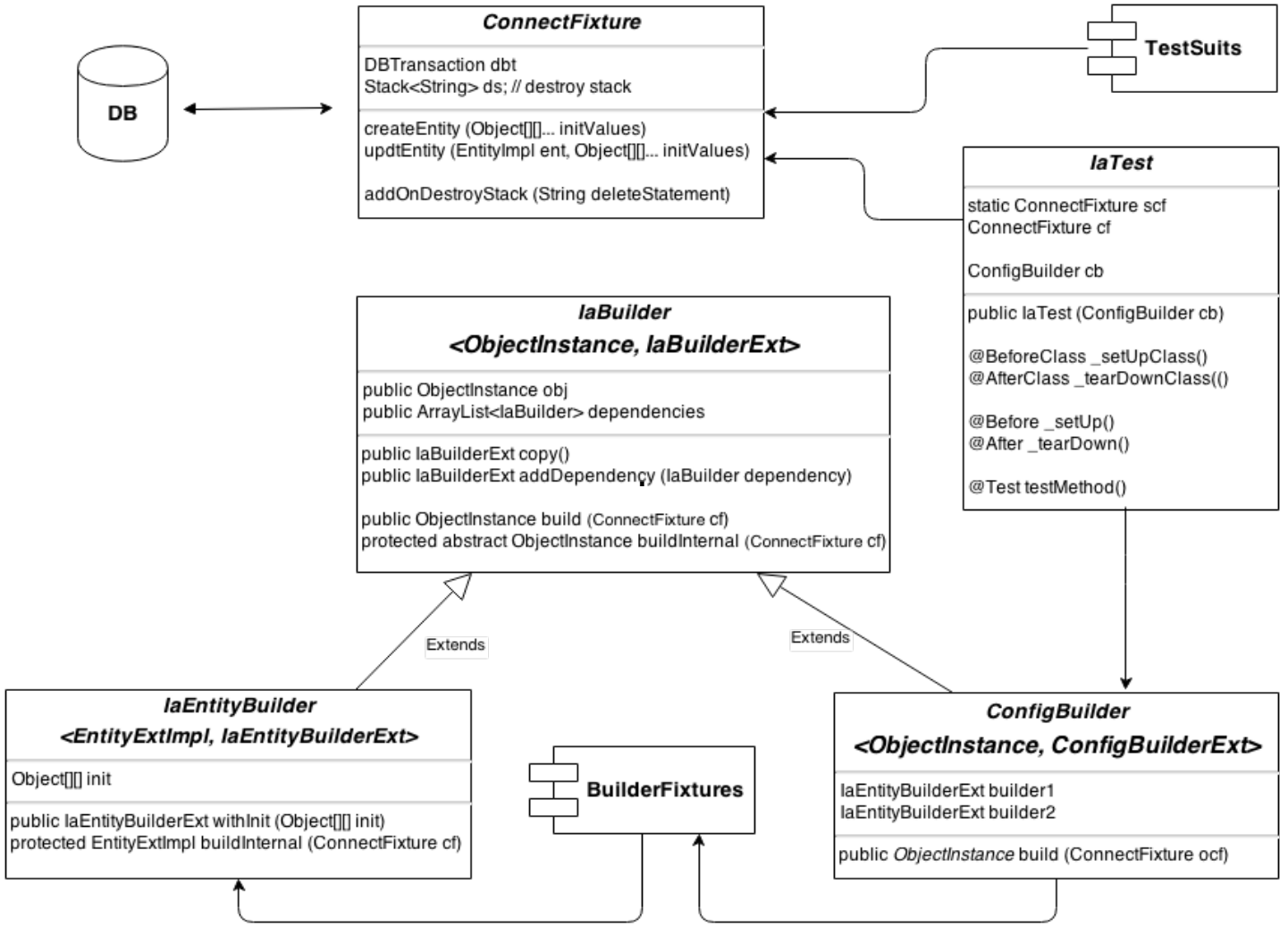
  AAA *- arrange / act / assert*

  DRY *– dont't repeat yourself*

  DAMP *- descriptive and meaningful phrases*

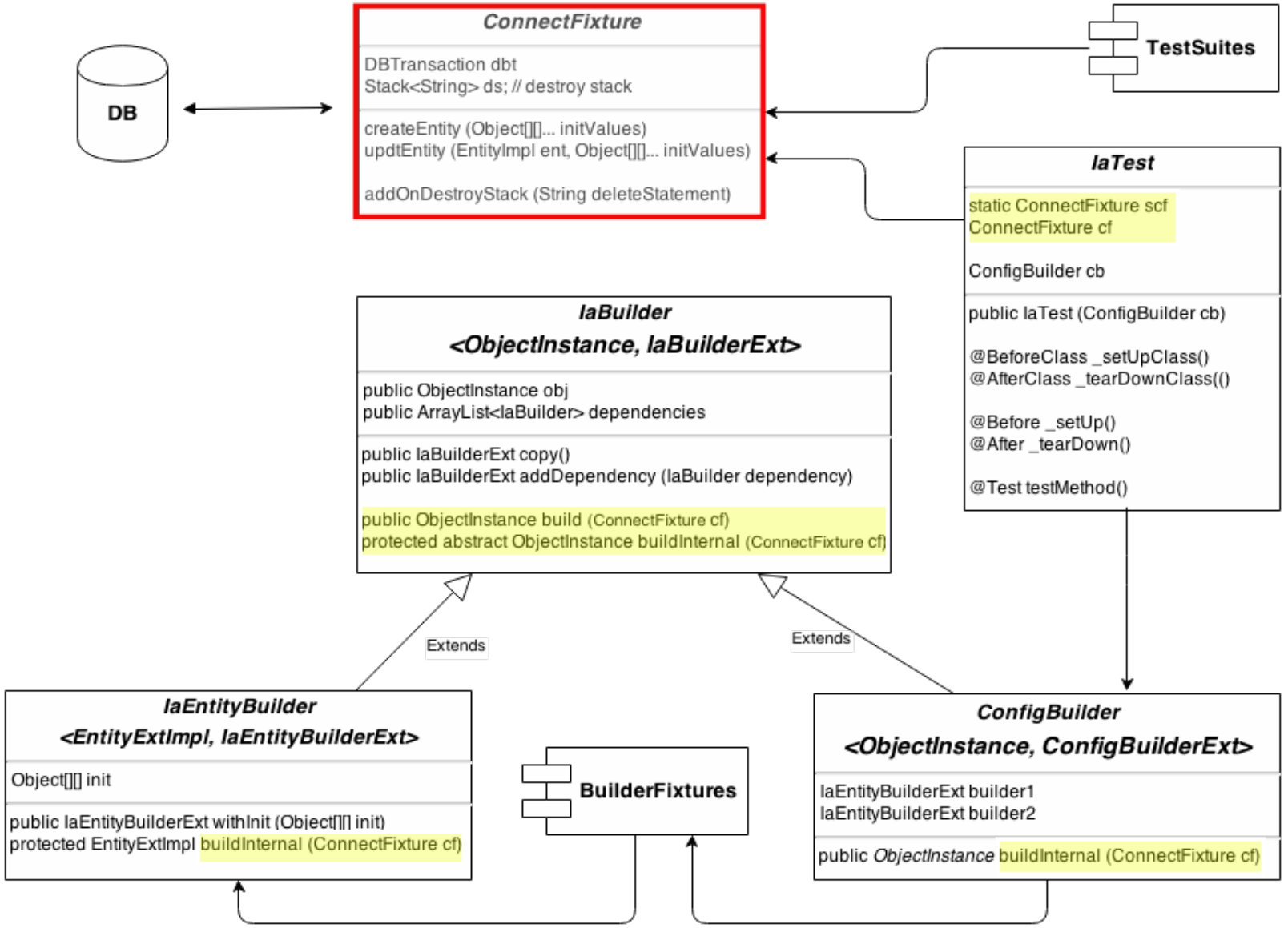
  composition over inheritance

Osnovne komponente testnog modula



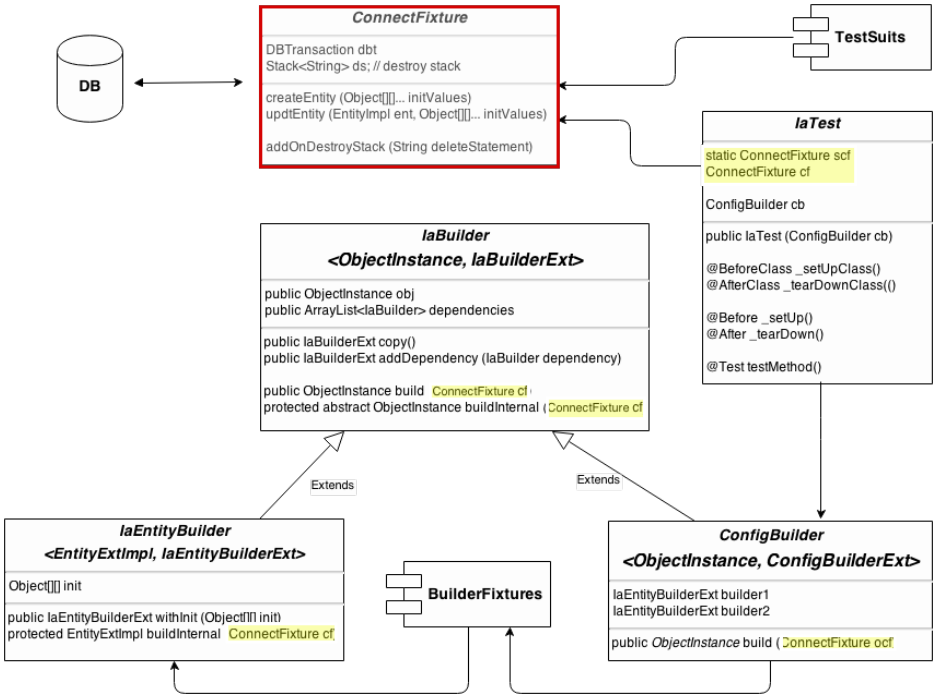
*ConnectFixture

- kontrolirano generiranje i "čišćenje" podataka -



*ConnectFixture

- Ima vezu na bazu i zna stvarati konkretne objekte iz domene (podatke u bazi)
- pamti redoslijed stvaranja objekata što omogućava i njihovo uništavanje
  - odmah po završetku testiranja
  - ili naknadno – obično sa sljedećim pokretanjem TestSuite-a
- Statički cf se koristi za pripremu zajedničke okoline za više srodnih testova u @BeforeClass
- Za svaki test se dodatno instancira po jedan radni cf



BasicConnectFixture

- osnovna komunikacija s bazom
- ažuriranje ADF entiteta na temelju predanog inicijalizacijskog niza

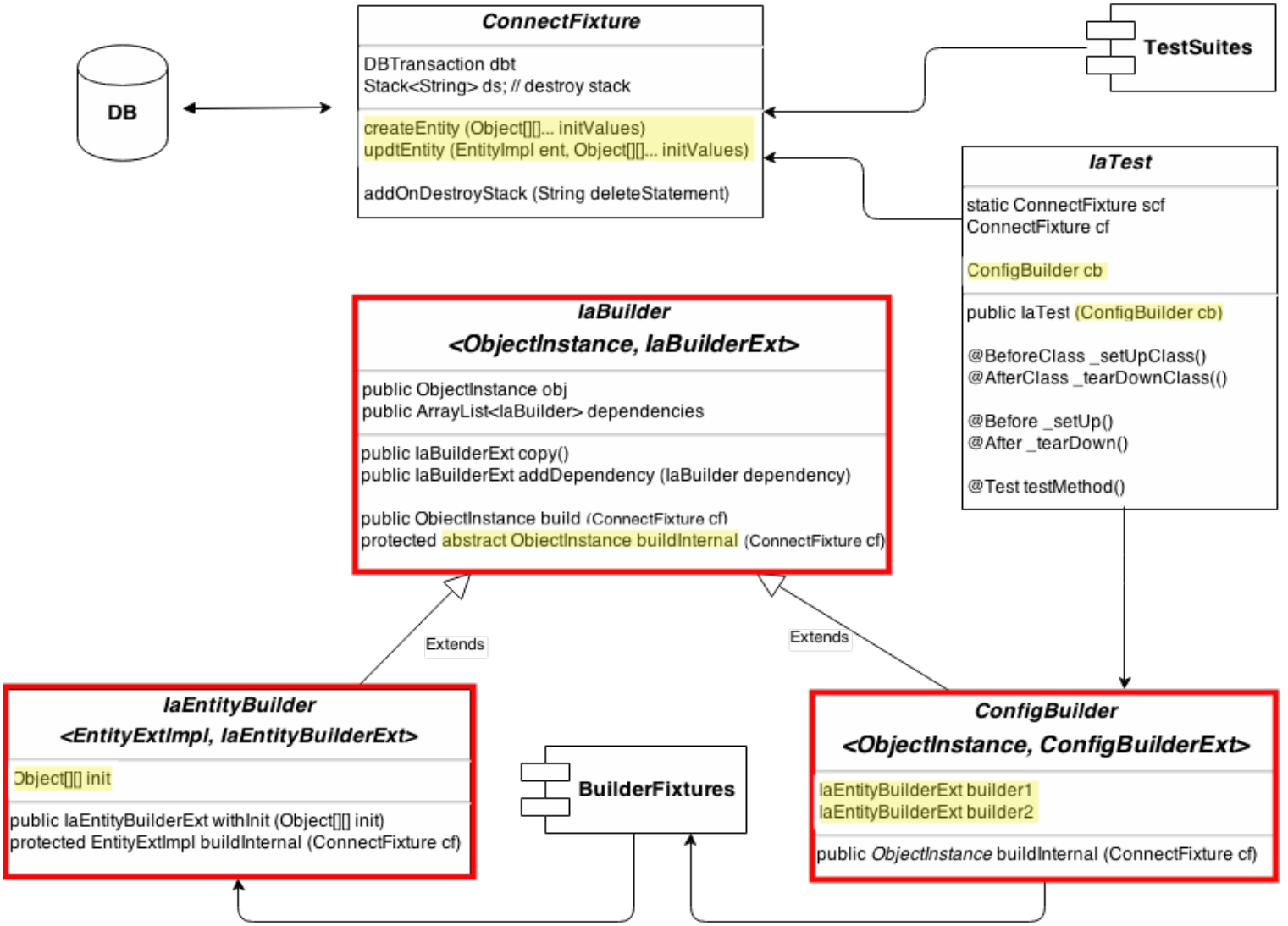
- **initValues** struktura:

```
new Object[][] {
        {EntityImpl.INDEX1, value1},
        {EntityImpl.INDEX2, value2}...
}
```

```
public class BasicTestConnectFixture {

    public final void updtEnt (EntityImpl ent, Object[][]... initValues) {
        for (Object[][] initPart: initValues) {
            for (Object[] init: initPart) {
                if (init.length != 2) {
                    continue;
                }

                int index = (Integer)init[0];
                Object val = init[1];
                log.debug(ent.getClass() + ": setAttribute(" + index + ", " + val + ")");
                ent.setAttribute(index, val);
            }
        }
    }
```

*Builders & Builder fixtures

- učinkovita izgradnja početnog stanja -



*Builders & Builder fixtures

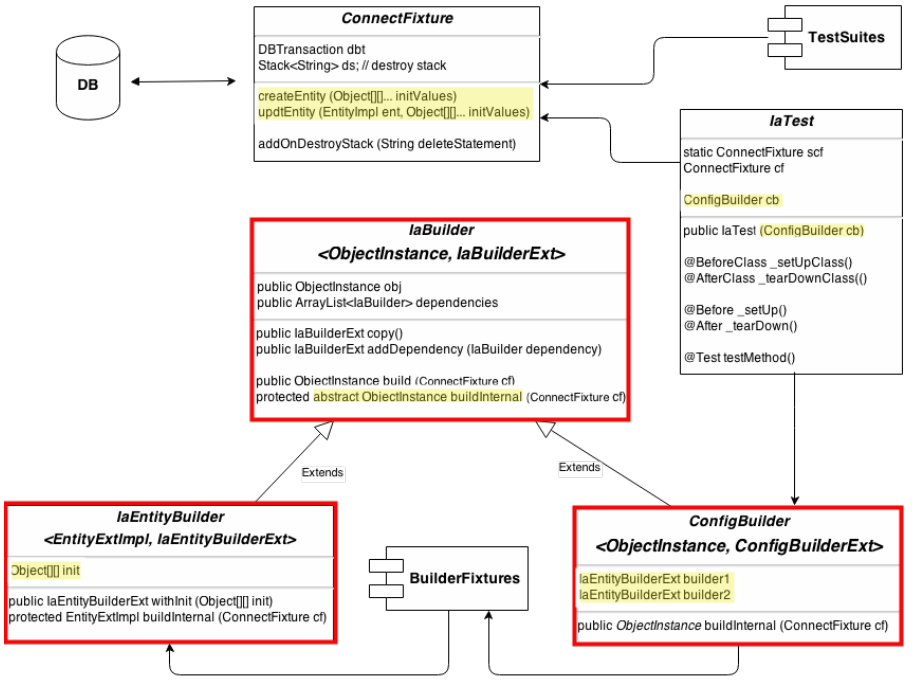
- učinkovita izgradnja početnog stanja -

*Builders

- pomoćnici u učinkovitoj izgradnji objekata s velikim brojem mogućih konfiguracija (parametara)
  - Predlošci za kreiranje konkretnih objekata
- tipovi
  - ***EntityBuilder:*** na bazičnoj razini – pomoćnik u izgradnji retka tablice u bazi,
  - ***ConfigBuilder:*** izgradnja ulaznog argumenta za parametrizirane testove
    - često kao gradivne elemente nosi EntityBuilder-e

*BuilderFixtures

- Sistematizirani i hijerarhijski organizirani predlošci builder objekata



Fluent Builders

Osnovna ideja

- Olakšano kreiranje kompleksnih objekata ulančanim pozivanjem gradivnih metoda

```
/* primjer izgradnje entiteta pomoću fluent buildera */
PrimiciImpl entPrimitak1 =
    new PrimitakBuilder()
            .withSati(NUM_168)
            .withIznos(NUM_1000)
            .withInit(new Object[][]{
                {PrimiciImpl.DANPOCETKA, NUM_1},
                {PrimiciImpl.DANKRAJA, NUM_31}
            })
            .build(of);
```

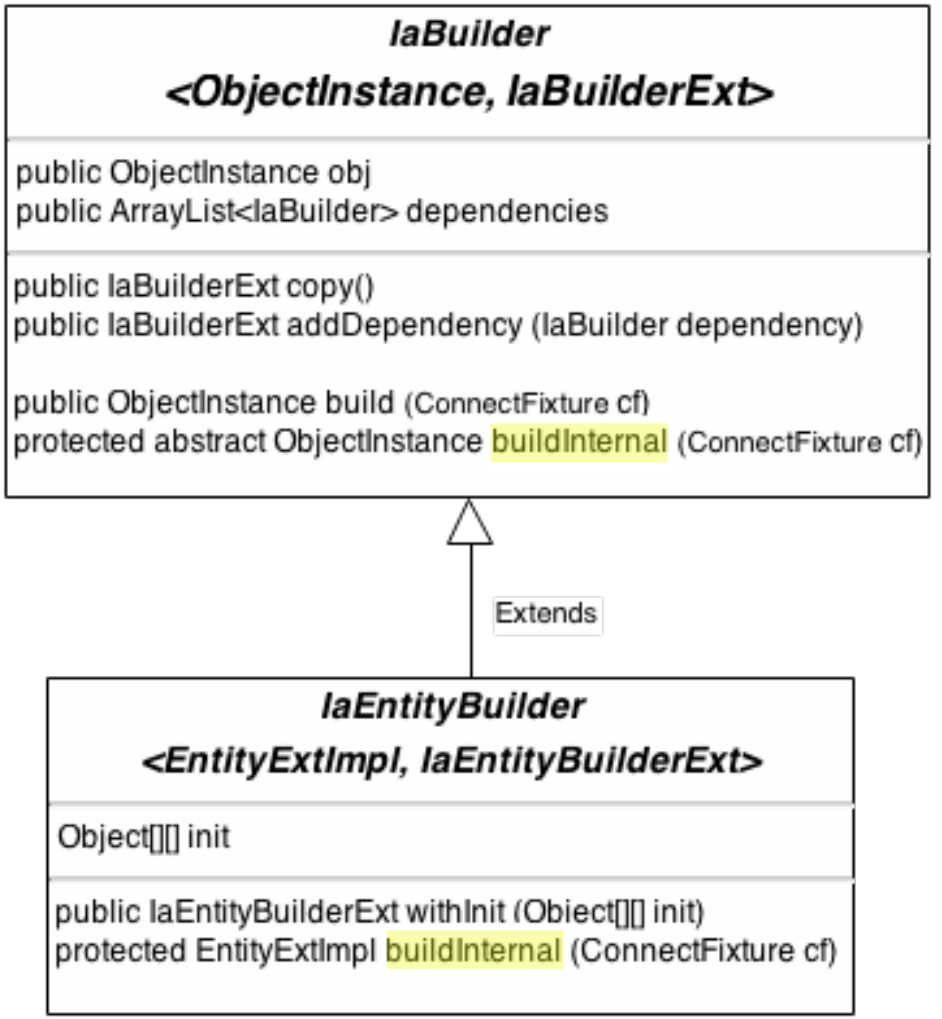
- Čišći i pregledniji kod nego uz korištenje setera za postizanje istog cilja

```
/* primjer izgradnje entiteta uz upotrebu setera*/
PrimiciImpl entPrimitak2 = createPrimiciImpl(of.getDbt());
entPrimitak2.setSati(NUM_168);
entPrimitak2.setIznos(NUM_1000);
entPrimitak2.setDanPocetka(NUM_1);
entPrimitak2.setDanKraja(NUM_31);
```

IaEntityBuilder

– fluent builder pattern + generics –

- **IaBuilder – osnovna klasa**
  - **copy metoda**:
    - vraća (IaBuilderExt)this
    - kopiranje objekta (deep copy)
    - posebno bitno kod iskorištavanja fixture objekata
  - **build metoda**
    - osnovna logika oko izgradnje objekta



- **IaEntityBuilder – predložak za kreiranje retka tablice u bazi**
  - **public EntitityExtImpl withInit(...) metoda**

```
Object[][] init = new Object[][] {
    {EntityImpl.INDEX1, value1},
    {EntityImpl.INDEX2, value2}...
}
```

  - **buildInternal** implementacija

Fluent Builder pattern – implementacija

- Metoda ***build Internal***
  - vraća konkretni objekt
  - u ovom slučaju (adf) entitet *PrimiciImpl*
- Metode ***withAtt**:*
  mogućnost ulančanog pozivanja gradivnih metoda
  - postavljanje željenih (potrebnih) parametara
  - vraćaju ***this***: mogućnost ulančanog pozivanja u fazi pripreme builder objekta

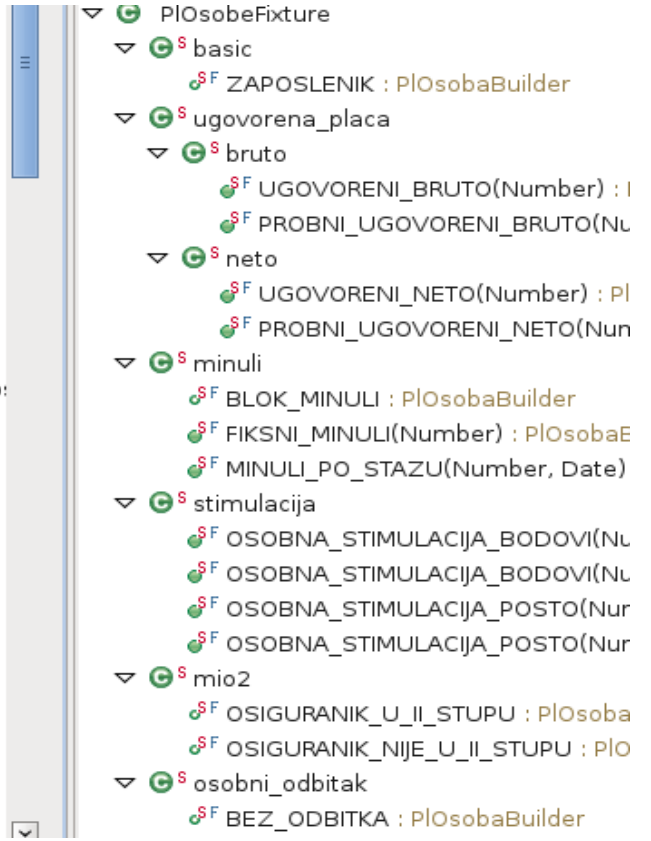
```
public class PrimitakBuilder extends IaObBuilder<PrimiciImpl, PrimitakBuilder>{
    PrimanjeBuilder primanjeBuilder = null;

    @Override
    protected PrimiciImpl buildInternal (HridsConnectFixture ocf){
        Number primanjeId = null;
        if(primanjeBuilder.obj == null){
            primanjeId = primanjeBuilder.withVlasnikId(vlasnikId)
                                        .withObradaId(obradaId)
                                        .build(ocf)
                                        .getPrimanjeId_Num();
        }

        primanjeId = primanjeBuilder.obj.getPrimanjeId_Num();

        PrimiciImpl ent = ocf.crPrimitak(obradaId, osobaId, primanjeId, init);
        return ent;
    }

    public PrimitakBuilder withPrimanjeBuilder(PrimanjeBuilder primanjeBuilder){
        this.primanjeBuilder = primanjeBuilder;
        return this;
    }

    public PrimitakBuilder withSati (Number sati) {
        return withInit(PrimiciImpl.SATI, sati);
    }

    public PrimitakBuilder withIznos (Number iznos) {
        return withInit(PrimiciImpl.IZNOS, iznos);
    }

    public PrimitakBuilder withRazdoblje (Number danOd, Number danDo) {
        return withInit(new Object[][]{
                {PrimiciImpl.DANPOCETKA, danOd},
                {PrimiciImpl.DANKRAJA, danDo},
        });
    }
}
```

Entity builder fixtures

– primjer: zaposlenici s različitim karakteristikama gledano u domeni obračuna plaća -

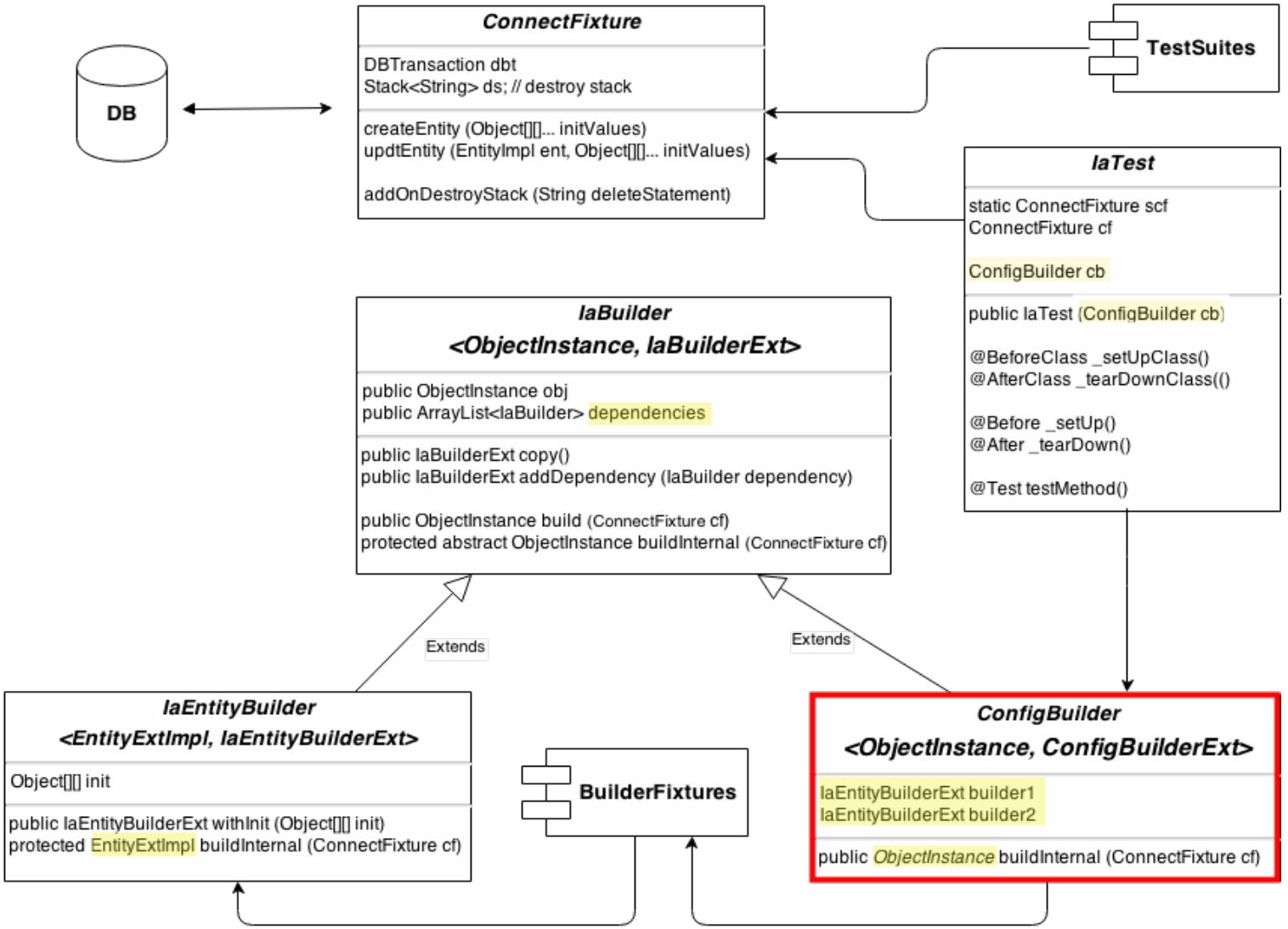
- Predefinirane, hijerarhijski organizirane konfiguracije builder objekata
- Mogu biti u obliku statičkih metoda koje primaju parametre, ali mogu biti i u obliku običnih statičkih atributa
- Osnovna uloga: sistematizacija predložaka za kreiranje entiteta (entity buildera)
- Kompliciranije varijante se temelje na nadogradnji jednostavnije varijante (metoda copy!) uz postavljanje dodatnih atributa

```
public class PlOsobeFixture {
    public static class basic{

    public static class ugovorena_placa{

        public static class bruto{
            public static final PlOsobaBuilder UGOVORENI_BRUTO (Number iznos){
                return basic.ZAPOSLENIK.copy()
                        .withInit(new Object[][] {
                                {PlOsobeImpl.PREZIME, "UGOVORENI_BRUTO"},
                                {PlOsobeImpl.PLACAUGOVORENA, iznos}
                        });
            }

            public static final PlOsobaBuilder PROBNI_UGOVORENI_BRUTO (Number izno
                return basic.ZAPOSLENIK.copy()
                        .withInit(new Object[][] {
                                {PlOsobeImpl.PREZIME, "PROBNI_UGOVORENI_BRUTO"},
                                {PlOsobeImpl.PLACAUGOVORENAPROBNI, iznos},
                                {PlOsobeImpl.PROBNIDATUMDO, probniDatumDo}
                        });
            }
        }

        public static class neto{
            public static final PlOsobaBuilder UGOVORENI_NETO (Number iznos){
                return ugovorena_placa.bruto.UGOVORENI_BRUTO(iznos).copy()
                        .withInit(new Object[][] {
                                {PlOsobeImpl.PREZIME, "UGOVORENI_NETO"},
                                {PlOsobeImpl.PLACAUGOVORUNETUFLAG, Boolean.TRUE}
                        });
            }
```



*Config builder

- izgradnja složenih konfiguracija SUT-a -



Config builder

- Priprema ulaznih parametara za parametrizirane testove -

```
@RunWith(Parameterized.class)
public class StimulacijeTest extends ObracunTest {
    protected static ObracunConnectFixture stf = null;

    //...

    public StimulacijeTest (String name, Config config) {
        //...
    }
```

```
/* osobna stimulacija zadana u bodu sa obracunom minulog rada samo na osnovni dio place*/
konfiguracije.put(Konfiguracije.OSB_MINULI_NA_BOD,
        new Config()
            .withPlosoba(
                PlOsobeFixture.stimulacija.OSOBNA_STIMULACIJA_BODOVI(bodovi, stimulbodovi)
                    .withInit(PlOsobeImpl.FIKSMINULIPOSTO, minuliPosto)
            )
            .withPrimitak(
                PrimiciFixture.basic.OSNOVNI_SA_SATIMA(fondSatiZaObracunBoda)
                    .withPrimanjeBuilder(
                        PrimanjeFixture.stimulacija.osobna_stimulacija.bruto_bod1obr.s_minulim.OSOBNA_STIMULACIJA_BLOK_MINULI.copy()
                    )
            )
            .withOcekivaniBruto(
                (
                vrijednostBoda
                    .multiply(bodovi)
                    .multiply(minuliPosto.divide(100).add(1))
                )
                .add(vrijednostBoda.multiply(stimulbodovi))
            )
);
```

Dizajniranje preglednih i održivih testova

- Osnovni smisao testa mora biti očit na prvi pogled
  - Arrange/Act/Assert (Given/When/Then) princip
  - DAMP - descriptive and meaningful phrases
  - DSL – domain specific language
- Composition over inheritance!
- Kratke metode
  - Idealno : 10 – 15 linija koda
  - DRY - dont repeat yourself
- Izbjegavati predetaljnu specifikaciju SUT-a
  - Ne želimo da nam se testovi raspadnu nakon refaktoriranja, npr. promjene potpisa konstruktora i metoda

```
public void testStimulacija(Number vlasnikId, Number obradaId){
    // Arrange (Given)
    PrimiciImpl entPrimitak = config
                              .withVlasnikId(vlasnikId)
                              .withObradaId(obradaId)
                              .withPrezime(testName)
                              .build(of);
    Number osobaId = entPrimitak.getOsobaId();


    // Act (When)
    of.obracunaj(obradaId, osobaId);
    entPrimitak.refresh(Row.REFRESH_WITH_DB_FORGET_CHANGES);


    // Assert (Then)
    assertEquals("primici.bruto", config.ocekivaniBruto, entPrimitak.getBruto());
}
```

Izazovi + rješenja

IZAZOV **RJEŠENJE**

- Svi (smisleni) testovi usko vezani uz bazu
  - Osigurati kontrolirano generiranje i "čišćenje" podataka → *ConnectFixture*
- Kreiranje objekata sa velikim brojem parametara
  - Osmisliti učinkovit način izgradnje početnog stanja → *Fluent Builder pattern*
- Velik broj testnih kombinacija
  - Osmisliti učinkovit način sistematizacije podataka i testova → *Builder fixtures*

  Test suites

- Dizajniranje održivih i preglednih testova →

  AAA *- arrange / act / assert*

  DRY *– dont't repeat yourself*

  DAMP *- descriptive and meaningful phrases*

  composition over inheritance

Zaključak

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence”

Dijkstra (1972)

Mirna Katičić
mkaticic@infoart.hr

infoart
inovacija. znanje. tehnologija.

Automatizirano funkcijsko testiranje ADF aplikacija

Mirna Katičić, Infoart d.o.o.

Rovinj, 16.10.2014.

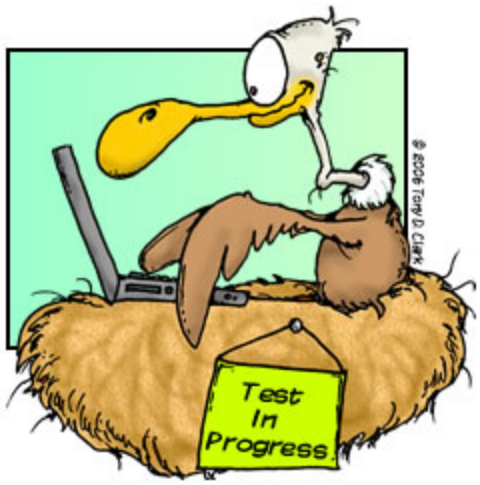
Dobar dan,
Moje ime je Mirna Katičić i zaposlena sam u poduzeću Infoart gdje već šestu godinu zaredom sudjelujem u osmišljavanju i razvoju poslovnih rješenja temeljenih na javi i ADF-u

Cilj današnjeg predavanja je podjeliti s vama moja iskustva vezana uz razvoj testnog modula za provedbu automatiziranog funkcijskog testiranja .

infoart
inovacija. znanje. tehnologija.

Sadržaj

- Uvod
- Motivacija i ciljevi
- Izazovi u priremi testnog modula
- Dizajniranje preglednih i održivih testova - smjernice
- Osnovne komponente testnog modula
  - ConnectFixture
  - Fluent builder pattern
    - EntityBuilder
    - ConfigBuilder
  - Builder fixtures
- Q&A



Dakle – da vidimo što nas čeka...

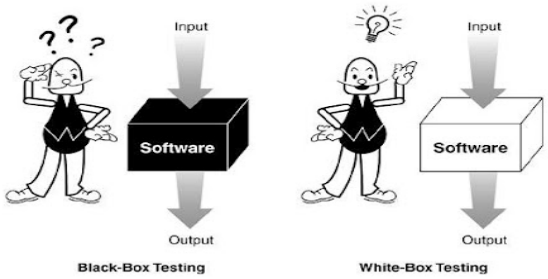
Nakon kratkog uvoda u temu brzo ćemo se baciti na konkretne stvari.

Osnovni cilj prezentacije je dati smjernice za dizajniranje preglednih i održivih (iako u svojoj strukturi kompleksnih) funkcijskih testova.

Pokazati ću vam i objasniti dizajn osnovnih komponenti testnog modula te na primjerima pokazati na koji način ih koristimo za pisanje boljih i preglednijih testova.

infoart
inovacija. znanje. tehnologija.

Funkcijski / integracijski testovi



- Vrsta testiranja na principu "crne kutije"
  - Ne zanima nas trenutna implementacija pojedinih dijelova sustava
  - Testni slučajevi se baziraju na specifikaciji ponašanja sustava kojeg testiramo (SUT-a)
- "white box testing" - testiranje strukture

3/26

Za početak – da razjasnimo **termin** "Funkcijsko testiranje".

Za razliku od unit testova – funkcijski testovi ne mare za strukturu pojedinih dijelova sustava već se baziraju na testiranju specifikacije ponašanja sustava.

Na njih možemo gledati kao na svojevrsnu **crnu kutiju** u koju šaljemo sustav u poznatom incijalnom stanju, a na izlazu provjeravamo da li su se određeni dijelovi sustava promijenili prema očekivanjima.

Funkcijski testovi često opisuju **očekivanu korisničku interakciju** sa aplikacijom.

Termini "funkcijski" i "integracijski" testovi se često koriste kao sinonimi. Tj. ispravna integracija povezanih komponenti preduvjet je za ispravno funkcioniranje sustava.

infoart

KONTINUIRANI RAZVOJ I ODRŽAVANJE POSLOVNIH APLIKACIJA

POSiA*Sustav upravljanja robnim tokovima

FiKs*Sustav upravljanja financijama i računovodstvom

HRiDs*Sustav upravljanja ljudskim resursima

4/26

Prije nego krenemo u tehničke detalje izvedbe samog testnog modula, ukratko bih vam predstavila svoju firmu kako bi bar ugrubo mogli dobiti predodžbu o testiranju kakvih aplikacija je ovdje riječ.

Dakle, zaposlena sam u Infoartu gdje trenutno radi oko 40 stalnih zaposlenika.

Najveći dio poslova se odnosi na **kontinuirani** razvoj poslovnih rješenja za velika i srednje velika poduzeća.

infoart
inovacija. znanje. tehnologija.

Motivacija za unapređenje procesa testiranja

Obračun plaća

- **Izmjene u centralnom dijelu obračuna utječu na sve korisnike**
  - Proširenje funkcionalnosti za novog korisnika zahtijeva izniman oprez
- **Mnoštvo parametara**
  - Na obračun jednog primitka utječe oko 100 različitih parametara razmještenih na nekoliko razina (osoba, org. jedinica, radno mjesto, vrsta primitka, obrada...)
  - Netrivijalna priprema testne okoline
- **Učestale izmjene zakonskih regulativa**
  - Stalne izmjene/nadogradnje aplikacije su zagarantirane i bez novih korisničkih zahtjeva
  - Nadogradnje ove vrste se obično moraju isporučiti žurno
  - Na raspolaganju je kratak period za kvalitetno testiranje

5/26

Taj **kontinuirani** razvoj je zapravo i bio osnovna motivacija za automatizaciju testnog procesa.

Konkretno ja osobno radim na razvoju aplikacije za obračun plaća.

Plaće su u svojoj prirodi vrlo nezgodne za održavanje i nadogradnju iz nekoliko razloga:

- učestale izmjene bitnih dijelova aplikacije zagaratirane su
  - što zbog korisničkih zahtjeva
  - što zbog promjene zakonskih regulativa.
- te nadogradnje su osobito opasne jer se obično zahtjeva **žurna instalacija** što znači da nemamo na raspolaganju dovoljno vremena za provođenje kvalitetnog testiranja!

infoart
inovacija. znanje. tehnologija.

Automatizacija testnog procesa - ciljevi

Kratkoročni ciljevi:

- Identifikacija i prevencija grešaka u što ranijoj fazi razvoja
- Sprečavanje preranog puštanja u produkciju
- Procjena usklađenosti sa specifikacijama (zakonskim, korisničkim)
- Definiranje uobičajenog načina korištenja programa i njegovih komponenti

Dugoročni ciljevi:

- Podizanje kvalitete aplikacije
- Zadovoljstvo korisnika
- Smanjenje troškova održavanja

6/26

Koji su – dakle – ciljevi automatizacije testnog procesa?

Svakako je osnovni cilj prevencija grešaka u što ranijoj fazi razvoja i sprečavanje preranog puštanja u produkciju.

Kasnije dolazi sve ostalo. Procjena uskl. Sa spec., definiranje uobičajenog načina korištenja...

Dugoročno, naravno, želimo podići kvalitetu aplikacije što će rezultirati dizanjem zadovoljstva korisnika te posljedično smanjiti troškove održavanja.

Ukratko - testiranje i analiza kvalitete će ovog malog čovječuljka dovesti do uspjeha :)

infoart
inovacija. znanje. tehnologija.

Dodana vrijednost

Sistematizirano dokumentiranje funkcionalnosti aplikacije

- dobro dizajnirani testovi mogu biti vrlo koristan pomagač u opisu očekivanog ponašanja sustava

Mogućnost odgođenog uništavanja testnog okruženja – pregled testnih podataka kroz aplikaciju

- priprema testnog okruženja je bitan i često najzahtjevniji korak u samom procesu testiranja (bilo ručnog ili automatiziranog)
- Uz opciju "odgođenog" uništavanja testnog okruženja moguće je rezultate automatski generiranih testova pogledati kroz samu aplikaciju što je vrlo korisno u samoj fazi pisanja testova, ali i kasnije za razumijevanje, debugiranje i sl.

Osnovne opcije za pokretanje testova

- svih testova od jednom – npr. jednom nekad u noći
- određenog podskupa – kada nemamo vremena zavrtiti sve

Automatizacija testnog procesa donosi i neke (na prvu ruku) neočekivane benefite.

1. dobro dizajnirani testovi mogu biti vrlo koristan *pomagač* u opisu očekivanog ponašanja sustava
2. također, pokazala se vrlo korisna opcija *"odgođenog pospremanja"*. To nam omogućava pregled podataka generiranih za potrebe testiranja kroz samu aplikaciju – što se pokazalo osobito korisno u fazi pisanja testova.

infoart
inovacija. znanje. tehnologija.

JUnit – vrlo kratak pregled

Set up and clean up metode

`@BeforeClass, @AfterClass` - izvršit će se samo jednom za testnu klasu

`@Before, @After` - izvršit će jednom za svaki test (metodu)

Test runners

```
@RunWith(Suite.class)
@RunWith(Parameterized.class)
@RunWith(Categories.class)
```

Rules

```
@Rule  public TestName testNameRule = new TestName();
@Rule  public Timeout globalTimeout = new Timeout(60000);
@Rule TemporaryFolder
```

Ostalo

```
@Test(expected = ArithmeticException.class)
@Ignore
```


Prije nego krenemo na glavni dio prezentacije, vrlo ćemo se kratko osvrnuti na Junit.

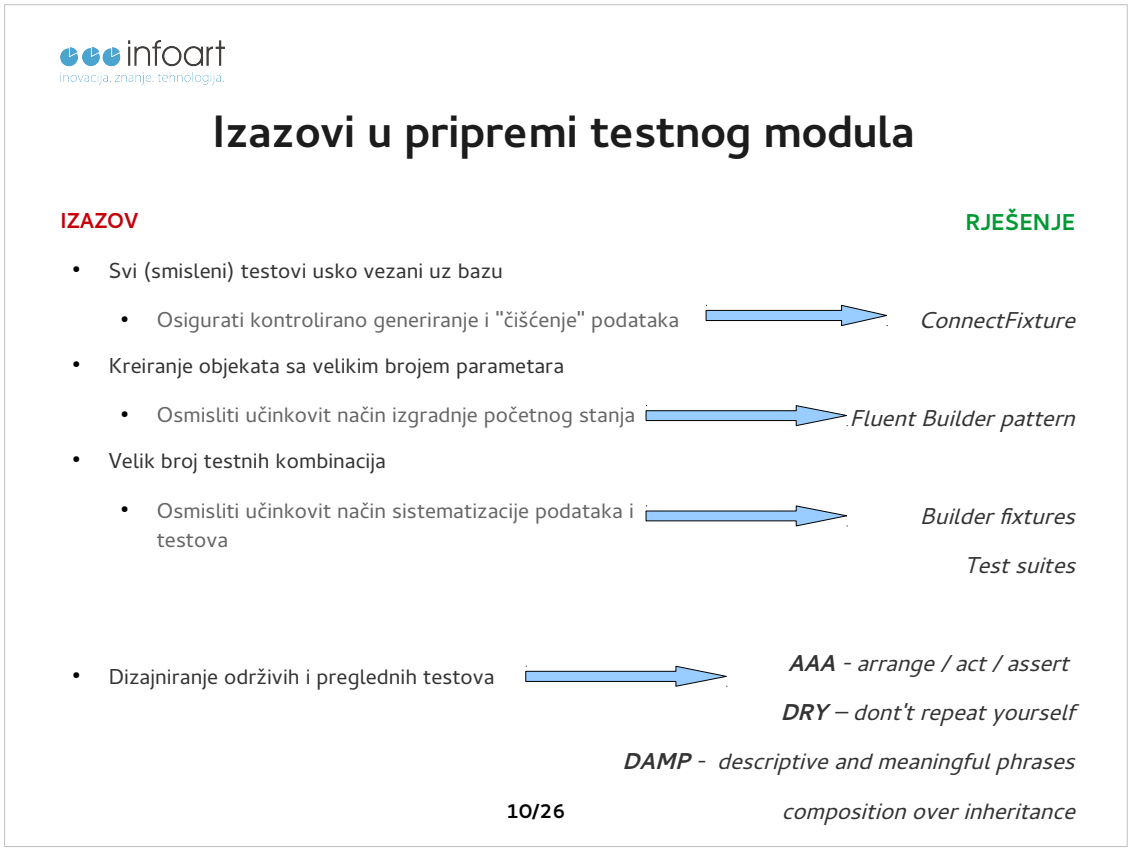
U izgradnji testnog modula je – dakle – korišten Junit 4 FW. Ipak, JU nije tema ove prezentacije te vas potičem da ukoliko još niste upoznati sa mogućnostima Junita ili nekog sličnog fw-a svakako istražite to područje.

Na ovom slajdu je dan pregled nekih osnovnih anotacija koje nam omogućuju da određenim djelovima naše testne klase pridružimo posebno značenje u *životnom ciklusu jednog testa*.

Dakle da vidimo – koji su to bili najveći izazovi u pripremi testnog modula?

Pisanje testova za kompleksne poslovne aplikacije nije baš usporedivo sa jednostavnim primjerima koje nam izbacuje google ako damo upit na temu.

Svi znamo da bi u nekom utopijskom svijetu testiranja svaki test morao biti u potpunosti nezavisan od okoline i drugih testova.



Naš prvi izazov je to što je **većina smislenih funkcijskih testova usko vezana uz bazu**. Prema tome – potrebno je osmisliti učinkovit način za generiranje i brisanje podataka.

Drugi izazov je **velik broj parametara** o kojima ovise određene funkcionalnosti. To znači da **trebamo pomagača** za izgradnju složenih podatkovnih struktura na kojima nam se temelji rad aplikacije.

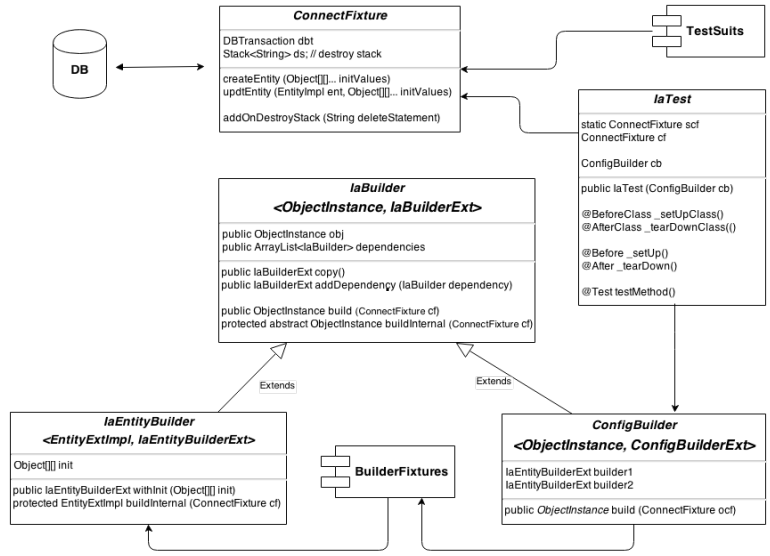
Dalje – testiranje složenih aplikacija podrazumjeva velik broj testnih kombinacija – trebamo osmisliti učinkovit način **sistematizacije podataka i testova** kako se ne bi izgubili u cijeloj toj šumi podataka.

I u konačnici – trebamo **metodologiju** za pisanje preglednih i održivih testova.

U nastavku ćemo detaljnije objasniti koncepte koje smo predložili kao rješenja za navedene izazove

infoart
inovacija. znanje. tehnologija.

Osnovne komponente testnog modula



11/26

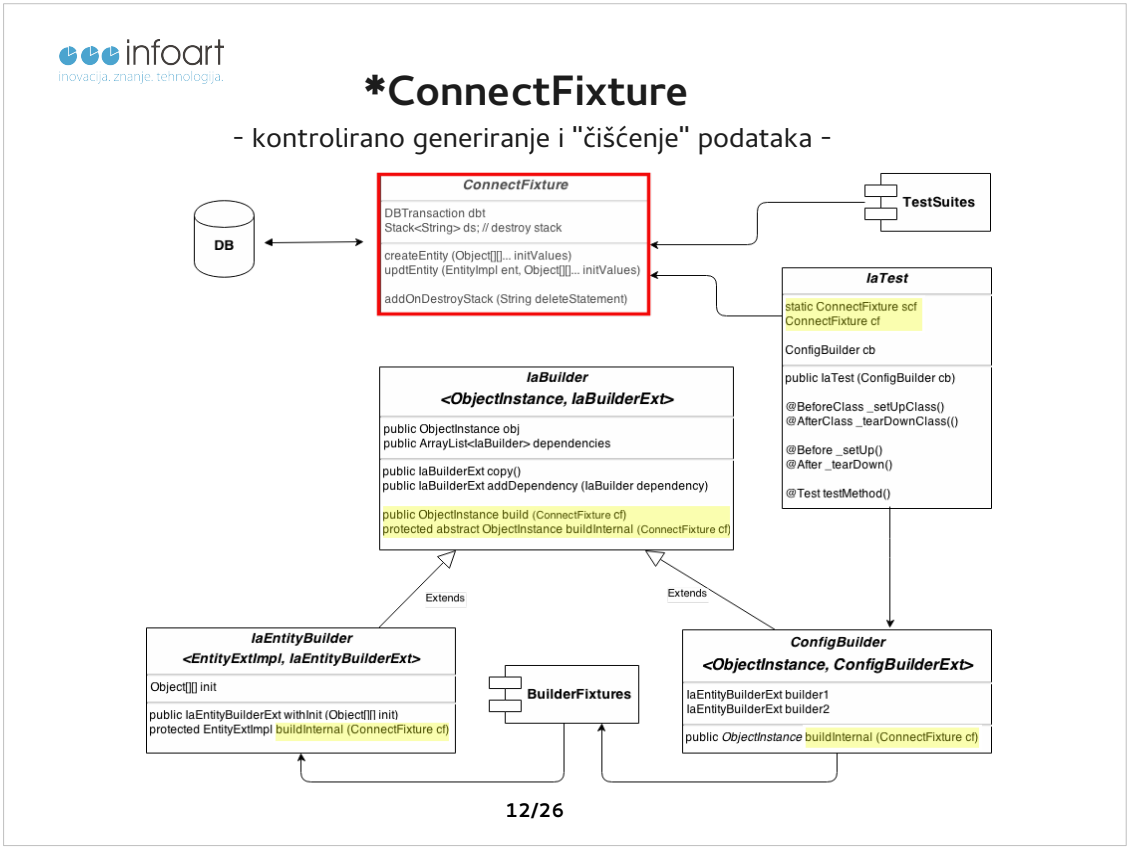
Dakle, koje su to – osim samih testova i test suiteova - osnovne komponente testnog modula?

Kada u testiranju kažemo da kao preduvjet za provednu testa moramo izgraditi tzv. **Fixture –** zapravo želimo reć da želimo izgraditi sustav u poznatom stanju. U kontekstu naših složenih poslovnih aplikacija koje su – kao što smo rekli – neodvojive od baze – pod dovođenjem sustava u poznato stanje mislimo zapravo na punjenje baze poznatim podacima.

Komponenta koja je zadužena za održavanje konekcije, kreiranje i čišćenje podataka nužnih za provedbu testiranja zovemo ***ConnectFixture***

Donja skupina komponenti spada u kategoriju tzv. **Buildera**. Što su builderi? Builderi su naši pomagači u izgradnji kompleksnih konfiguracija koje su preduvjet za naše testove.

U nastavku ćemo detaljnije objasniti svaku od prikazanih komponenti.



Dakle – da vidimo koje je osnovna uloga cf-a i na koji način se on koristi u ostalim komponentama sustava.

Rekli smo, dakle, da CF Ima vezu na bazu i zna stvarati konkretne objekte iz domene (podatke u bazi). Između ostalog - pamti redoslijed stvaranja objekata što omogućava i njihovo uništavanje i to

- odmah po završetku testiranja
- ili naknadno – obično sa sljedećim pokretanjem TestSuite-a

Ako malo bolje pogledamo ove žute mrlje na ekranu – vidit ćemo da su to zapravo mjesta na kojima se cf koristi u drugim komponentama testnog modula

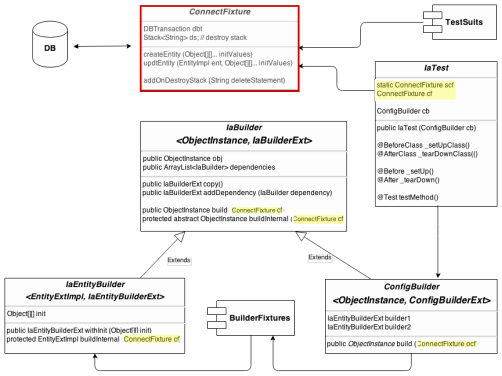
Pokazala se dobra praksa da za svaku **testnu klasu** imam po jedan statički cf – preko kojeg gradim podatke fiksne za sve testove te klase i jedan radni cf – koji se brine za održavanje podataka vezanih uz pojedini test

Pogledajmo na koji način se cf koristi u builderima? Radni cf se iz testa predaje se kao parametar **build** metodi. Ako na buildere gledamo samo kao na predloške za konkretne objekte koje želimo izgraditi u bazi – logično je da sami builderi ne sadrže referencu na cf – već se cf njima predaje tek u trenutku kad smo u potpunosti spremni izgraditi konkretan objekt – tj. Imamo popunjene sve parametre buildera koji su preduvjet za izgradnju objekta.

infoart
inovacija. znanje. tehnologija.

*ConnectFixture

- Ima vezu na bazu i zna stvarati konkretne objekte iz domene (podatke u bazi)
- pamti redoslijed stvaranja objekata što omogućava i njihovo uništavanje
  - odmah po završetku testiranja
  - ili naknadno – obično sa sljedećim pokretanjem TestSuite-a
- Statički cf se koristi za pripremu zajedničke okoline za više srodnih testova u @BeforeClass
- Za svaki test se dodatno instancira po jedan radni cf



infoart
inovacija. znanje. tehnologija.

BasicConnectFixture

- osnovna komunikacija s bazom
- ažuriranje ADF entiteta na temelju predanog inicijalizacijskog niza

- **initValues** struktura:

```
new Object[][] {
        {EntityImpl.INDEX1, value1},
        {EntityImpl.INDEX2, value2}...
}
```

```
public class BasicTestConnectFixture {

    public final void updtEnt (EntityImpl ent, Object[][]... initValues) {
        for (Object[][] initPart: initValues) {
            for (Object[] init: initPart) {
                if (init.length != 2) {
                    continue;
                }

                int index = (Integer)init[0];
                Object val = init[1];
                log.debug(ent.getClass() + ": setAttribute(" + index + ", " + val + ")");
                ent.setAttribute(index, val);
            }
        }
    }
```

14/26

BasicConnectFixture održava osnovnu komunikaciju sa bazom.

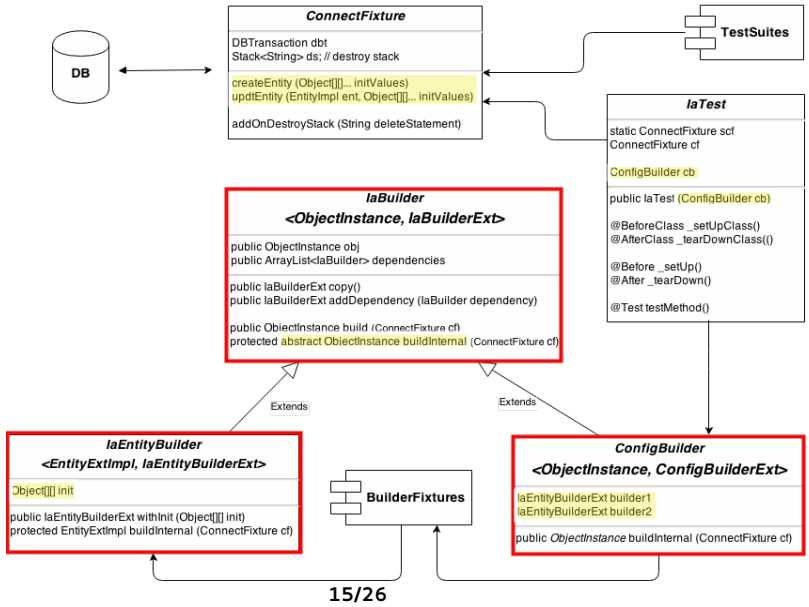
Za potrebe kreiranja/ažuriranja retka u bazi koristimo se činjenicom da svaki adf entitet omogućava ažuriranje svojih atributa preko metode **setAttribure(index, value).**

Tu činjenicu ćemo iskoristiti za pripremu općenite metode koja omogućava ažuriranje retka na temelju predane inicijalizacijske liste parova (indeks, value)

infoart
inovacija. znanje. tehnologija.

*Builders & Builder fixtures

- učinkovita izgradnja početnog stanja -



15/26

Što su builderi?

- Osnovna funkcija buildera jest učinkovita izgradnja objekata s velikim brojem mogućih parametara
- tipovi
  - ***EntityBuilder:*** na bazičnoj razini – redak tablice u bazi,
  - ***ConfigBuilder:*** apstraktnije konfiguracije se koriste npr. kao ulazni argument za parametrizirane testove, a često kao gradivne elemente nose EntityBuilder-e

*BuilderFixtures

- Predefinirane konfiguracije builder objekata

infoart
inovacija. znanje. tehnologija.

*Builders & Builder fixtures

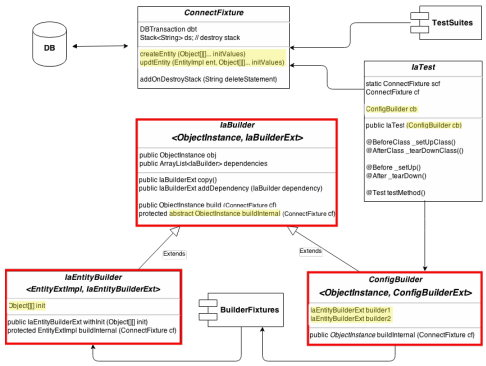
- učinkovita izgradnja početnog stanja -

***Builders**

- pomoćnici u učinkovitoj izgradnji objekata s velikim brojem mogućih konfiguracija (parametara)
  - Predlošci za kreiranje konkretnih objekata
- tipovi
  - ***EntityBuilder:*** na bazičnoj razini – pomoćnik u izgradnji retka tablice u bazi,
  - ***ConfigBuilder:*** izgradnja ulaznog argumenta za parametrizirane testove
    - često kao gradivne elemente nosi EntityBuilder-e

***BuilderFixtures**

- Sistematizirani i hijerarhijski organizirani predlošci builder objekata



infoart
inovacija. znanje. tehnologija.

Fluent Builders

Osnovna ideja

- Olakšano kreiranje kompleksnih objekata ulančanim pozivanjem gradivnih metoda

```
/* primjer izgradnje entiteta pomoću fluent buildera */
PrimiciImpl entPrimitak1 =
    new PrimitakBuilder()
            .withSati(NUM_168)
            .withIznos(NUM_1000)
            .withInit(new Object[][]{
                {PrimiciImpl.DANPOCETKA, NUM_1},
                {PrimiciImpl.DANKRAJA, NUM_31}
            })
            .build(of);
```

- Čišći i pregledniji kod nego uz korištenje setera za postizanje istog cilja

```
/* primjer izgradnje entiteta uz upotrebu setera*/
PrimiciImpl entPrimitak2 = createPrimiciImpl(of.getDbt());
entPrimitak2.setSati(NUM_168);
entPrimitak2.setIznos(NUM_1000);
entPrimitak2.setDanPocetka(NUM_1);
entPrimitak2.setDanKraja(NUM_31);
```

17/26

Dizajn osnovnog buildera se temelji na fluent builder patternu kojeg ćemo detaljnije objasniti u nastavku.

Ovaj design pattern je osobito koristan u pripremi testova jer nam pomaže u izgradnji kompleksnih objekata.

Ako pogledamo primjer desno, možemo uočiti da je izgradnja entiteta uz pomoć buildera puno preglednija u odnosu na klasičan način pomoću setera (u kodu nema beskorisnog ponavljanja imena entiteta koje ne nosi nikakvu informaciju kod postavljanja atributa).

Dodatno – u slučaju da nemamo – ili nam se ne da stvarati gradivne metode za sve atribute – nad svakim EntityBuilderom je moguće pozvati generičku **withInit** metodu.

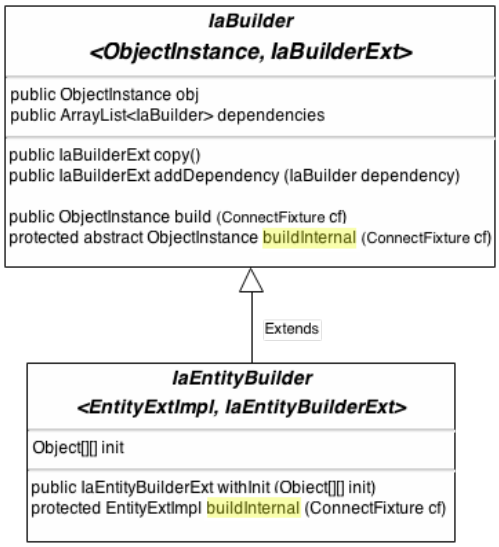
Kako je to moguće?

infoart
inovacija. znanje. tehnologija.

IaEntityBuilder

– fluent builder pattern + generics –

- **IaBuilder – osnovna klasa**
  - **copy metoda**:
    - vraća (IaBuilderExt)this
    - kopiranje objekta (deep copy)
    - posebno bitno kod iskorištavanja fixture objekata
  - **build metoda**
    - osnovna logika oko izgradnje objekta



- **IaEntityBuilder – predložak za kreiranje retka tablice u bazi**
  - public EntitityExtImpl withInit(...) metoda

```
Object[][] init = new Object[][] {
    {EntityImpl.INDEX1, value1},
    {EntityImpl.INDEX2, value2}...
}
```

  - **buildInternal** implementacija

18/26

Da bi mogli opće korisne gradivne metode "dignuti" u nadklase, a i dalje ih koristiiti u izvedenim builderima – trebaju na genericsi. Njima osiguravamo na gradivne metode iz nadklasa vraćaju objekt izvedenog buildera što omogućava ulančavanje gradivnih metoda – što iz nadklasa – što iz izvedenih buildera.

Koja svojstva mora imati osnovni builder?

Osnovni builder nosi referencu na ciljni objekt kojeg želi izgraditi. Kada se prvi put pozove build metoda, ta referenca će postat različita od null i to će biti znak eventualnim sljedećim pokušajima izgradnje da objekt već postoji.

Osim toga, osnovni builder omogućuje dubinsko kopiranje objekta i dodavanje dependency-ja. Ukoliko postoje dependecyji – njihovo buildanje će se pokrenuti prije izgradnje ciljnog objekta buildera.

Kao što smo već vidjeli u prethodnom primjeru, nad svakim entity builderom je moguće pokrenuti withInit metodu kojoj predajemo inicijalizacijsku listu

infoart
inovacija. znanje. tehnologija.

Fluent Builder pattern – implementacija

- Metoda ***build Internal***
  - vraća konkretni objekt
  - u ovom slučaju (adf) entitet *PrimiciImpl*
- Metode ***withAtt****:* mogućnost ulančanog pozivanja gradivnih metoda
  - postavljanje željenih (potrebnih) parametara
  - vraćaju ***this***: mogućnost ulančanog pozivanja u fazi pripreme builder objekta

```
public class PrimitakBuilder extends IaObBuilder<PrimiciImpl, PrimitakBuilder>{
    PrimanjeBuilder primanjeBuilder = null;

    @Override
    protected PrimiciImpl buildInternal (HridsConnectFixture ocf){
        Number primanjeId = null;
        if(primanjeBuilder.obj == null){
            primanjeId = primanjeBuilder.withVlasnikId(vlasnikId)
                                        .withObradaId(obradaId)
                                        .build(ocf)
                                        .getPrimanjeId_Num();
        }

        primanjeId = primanjeBuilder.obj.getPrimanjeId_Num();

        PrimiciImpl ent = ocf.crPrimitak(obradaId, osobaId, primanjeId, init);
        return ent;
    }

    public PrimitakBuilder withPrimanjeBuilder(PrimanjeBuilder primanjeBuilder){
        this.primanjeBuilder = primanjeBuilder;
        return this;
    }

    public PrimitakBuilder withSati (Number sati) {
        return withInit(PrimiciImpl.SATI, sati);
    }

    public PrimitakBuilder withIznos (Number iznos) {
        return withInit(PrimiciImpl.IZNOS, iznos);
    }

    public PrimitakBuilder withRazdoblje (Number danOd, Number danDo) {
        return withInit(new Object[][]{
                {PrimiciImpl.DANPOCETKA, danOd},
                {PrimiciImpl.DANKRAJA, danDo},
        });
    }
}
```

19/26

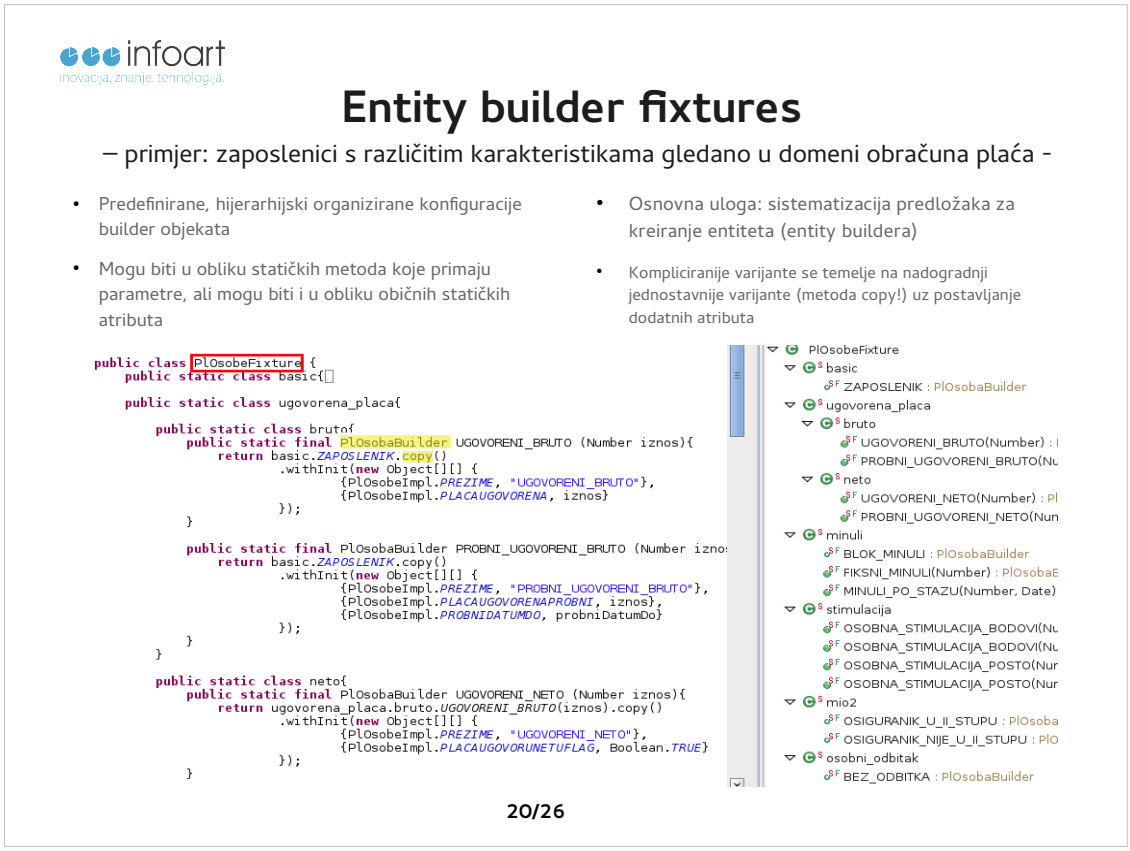
Evo, da pogledamo primjer jedne implementacije EntityBuildera.

U plaćama se cijeli obračun vrti oko primitaka – pa ćemo pokazati kako bi to trebala izgledati implementacija jednog pojednostavljenog **PrimitakBuildera.**

Osnovna metoda koju EntityBuilder mora implementirati iz nadkalse je **buildInternal** metoda. Primjetimo da je povratni tip te metode ciljni objekt – **PrimiciImpl –** odnosno ADF entitet ažuriran sa inicijalizacijskom listom koju smo pripremili preko gradivnih metoda **with***.

Kao što smo već spomenuli, ljepota fluent buildera je da ne moramo postaviti sve atribute nekog eniteta već samo one koje su nam nužne za provedbu određenog testa.

Fluent builder se pokazao puno fleksibilnijim rješenjem od nekih drugih "creational patterna" – npr. "Object mother" patterna.



Dakle, podsjetimo se - **Fixture** je pojam koji predstavlja sustav ili dio sustava u poznatom stanju.

Ja sam svoje fixture odlučila sistematizirati na sljedeći način. Za svaku tablicu u bazi (tj. adf entitet u modelu) imam jednu klasu u kojoj držim predefinirane i hijerarhijski organizirane builder objekte. Složenije varijante se temelje na jednostavnijim varijantama na način da se jednostavnija varijanta buildera kopira pomoću generičke metode copy koju nosi svaki IaBuilder i dodaju mu se specifični atributi za pojedinačnu svrhu.

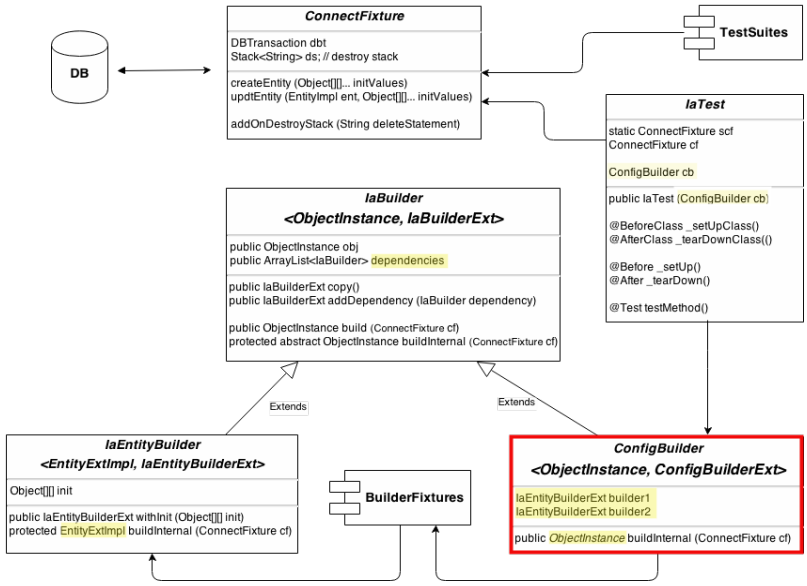
Na primjeru zaposlenika to izgleda ovako: na temelju osnovnog zaposlenika, možemo izgraditi "specijalne inačice" za potrebe testiranja ugovorene plaće. Pa tako imamo redom. Zaposlenika sa ugovorenom plaćom u brutu, jednog sa ugovorenom plaćom u netu i jednog sa probnom ugoovrenom plaćom.

Na istom principu zaposlenike možemo kategorizirati s obzirom na obračun stimulacije, obračun poreza, obračun doprinosa i td.

infoart
inovacija. znanje. tehnologija.

*Config builder

- izgradnja složenih konfiguracija SUT-a -



ConfigBuilder: najčešće se koristi kao ulazni argument za parametrizirane testove, a kao gradivne elemente najčešće nose EntityBuilder-e

infoart
inovacija. znanje. tehnologija.

Config builder

- Priprema ulaznih parametara za parametrizirane testove -

```
@RunWith(Parameterized.class)
public class StimulacijeTest extends ObracunTest {
    protected static ObracunConnectFixture stf = null;

    //...

    public StimulacijeTest (String name, Config config) {
        //...
    }


/* osobna stimulacija zadana u bodu sa obracunom minulog rada samo na osnovni dio place*/
konfiguracije.put(Konfiguracije.OSB_MINULI_NA_BOD,
        new Config()
            .withPlosoba(
                PlOsobeFixture.stimulacija.OSOBNA_STIMULACIJA_BODOVI(bodovi, stimulbodovi)
                    .withInit(PlOsobeImpl.FIKSMINULIPOSTO, minuliPosto)
            )
            .withPrimitak(
                PrimiciFixture.basic.OSNOVNI_SA_SATIMA(fondSatiZaObracunBoda)
                    .withPrimanjeBuilder(
                        PrimanjeFixture.stimulacija.osobna_stimulacija.bruto_bod1obr.s_minulim.OSOBNA_STIMULACIJA_BLOK_MINULI.copy()
                    )
            )
            .withOcekivaniBruto(
                (
                vrijednostBoda
                    .multiply(bodovi)
                    .multiply(minuliPosto.divide(100).add(1))
                )
                .add(vrijednostBoda.multiply(stimulbodovi))
            )
);
```


Najbolje ćemo smisao postojanja svih ovih komponenti o kojima smo do sada pričali, pojasniti ćemo sa jednim jednostavnim primjerom izgradnje – ne tako jednostavne konfiguracije za testiranje jednog od oblika stimulacije na plaću.

Dakle – cilj ovog koda je izgraditi konfiguraciju za testiranje obračuna osobne stimulacije zadane u bodu.

U konfiguraciju moramo ubaciti predložak za osobu sa potrebnim parametrima, kao i primitak sa podešenim parametrima za obračun željene vrste stimulacije.

Predloške za osobu i primitak dohvaćamo iz tzv. Fixture klasa za pojedine tablice.

Osim toga, u konfiguraciju moramo pospremiti i očekivani bruto nakon puštanja obračuna koji se iskazuje kao funkcija parametara zapisanih na osobi i primitku.

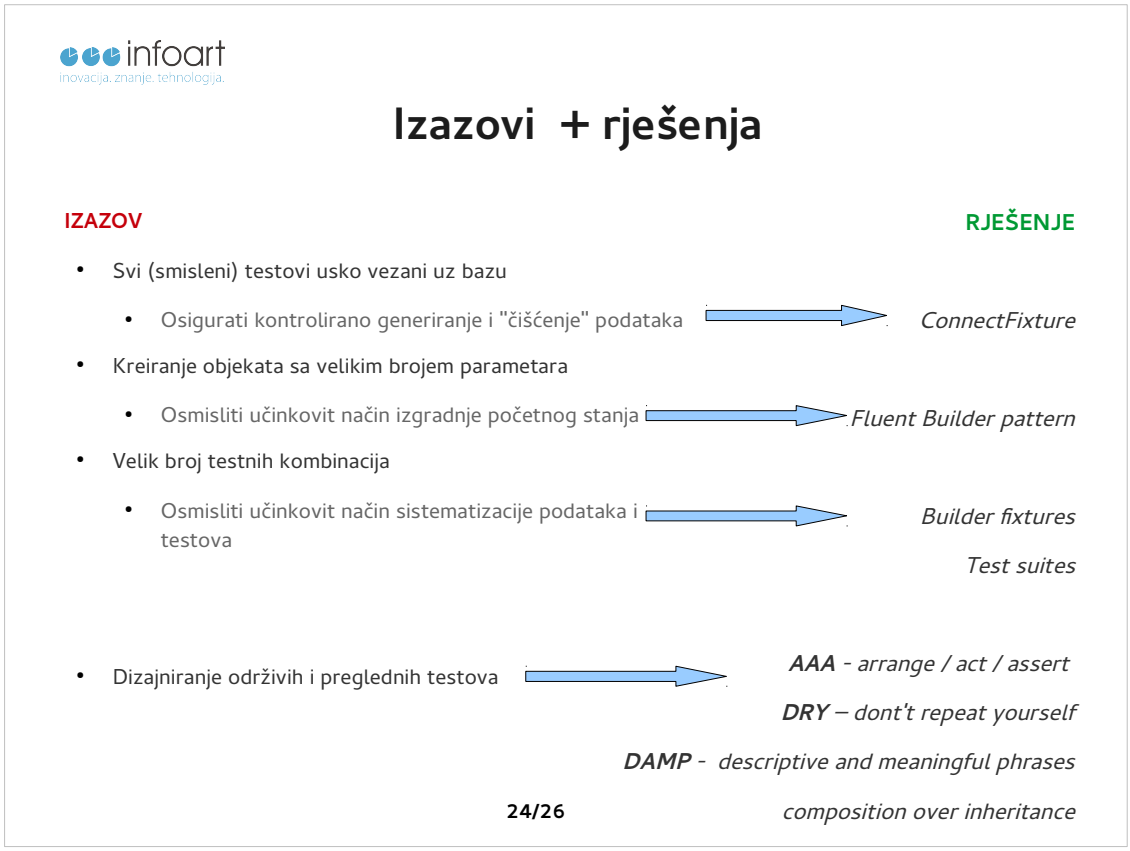
Kada smo pripremili konfiguracijski objekt – spremni smo s njim ući u osnovnu testnu metodu.

Preporuka za pisanje preglednih i održivih testova je sljedeća:

Osnovna testna metoda bi trebala biti kratka i iz nje bi na prvi pogled trebalo biti očito koja je njezina osnovna funkcija.

Osobito korisno za kasnije održavanje je istaknuti 3 osnona dijela testa po principu **arange-act-assert.**
Dakle – arrage: priprema test fixture-a, act – puštanje same testne funkcije (ili niza testnih funkcija) i assert: provjera rezultata.

Primjetimo da je ova naša funkcija fleksibilna i u smislu konfiguracije koju može "zavrtiti". Na prethodnom slajdu smo pokazali primjer slaganja konfiguracije za samo jednu vrstu stimulacije. U praksi – svaki korisnik obračunava stimulaciju na neki sebi svojstven način – parametri mogu biti na raznim mjestima (npr. Org jedinica, radno mjesto, obrada i sl.). Naš fleksibilni configuration builder dopušta da izgradimo sve te moguće kombinacije i zavrtimo ih u ovoj istoj **jednostavnoj** testnoj metodi.



Eto – samo još jednom da se podsjetimo – pogledajmo ovu listu izazova sa početka predavanja.

Da rezimiramo: probleme pripreme i uništavanja početnog stanja u bazi rješili smo ConnectFixture-om.

Kreiranje objekata sa velikim broj parametara nam je olakšao fluent builder pattern.

Veliki broj testnih kombinacija i testova smo organizirali su BuilderFixture klase i test suiteove.

Kod pisanja testnih metoda vodili smo se nekim općenitim dobrim praksama u dizajnu koda kako bi postigli fleksibilnost, preglednost i održivost naših testova.

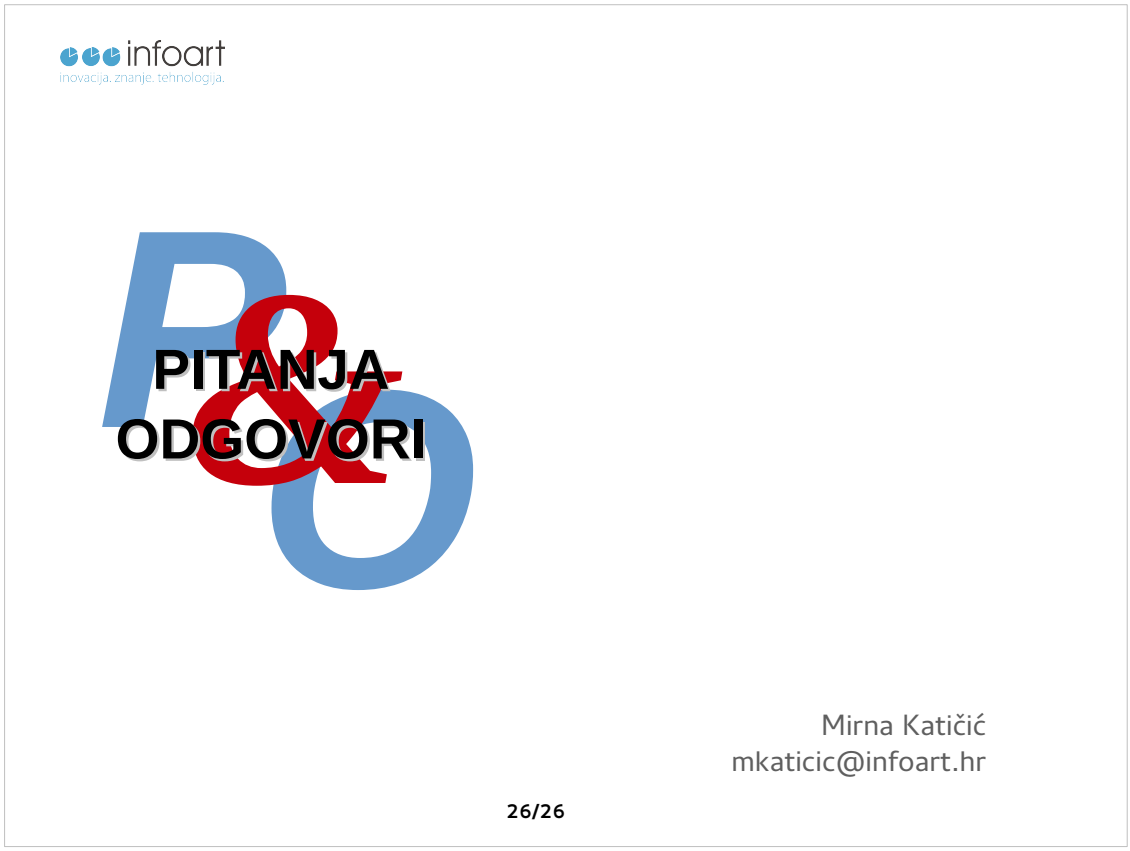
infoart
inovacija. znanje. tehnologija.

Zaključak

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"

Dijkstra (1972)

Kao zaključak možemo citirati Dijsktru koji je vrlo mudro primjetio da je testiranje vrlo učinkovit način za detekciju grešaka, ali je zapravo beznadno uzaludna metoda za dokazivanje da grešaka nema.



Eto, toliko od mene. A sada nam ostaje nešto vremena za pitanja i eventualnu diskusiju. Zapravo i mene zanima kakva su vaša iskustva sa testiranjem (pogotovo u manjim poduzećima).