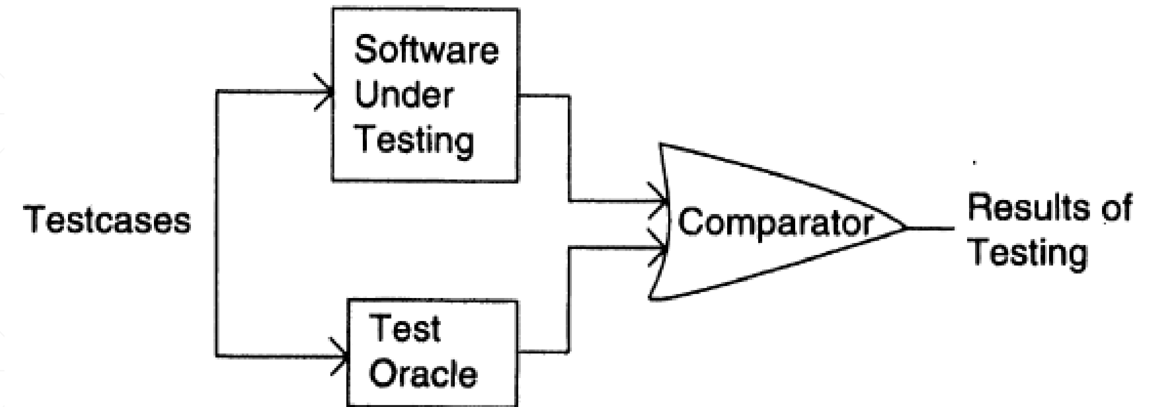
Testiranje softvera

Principi softverskog inženjerstva, *Elektrotehnički fakultet Univerziteta u Beogradu*

Terminologija u vezi sa greškama i testiranjem

- Izvor: Glossary of Software Engineering Terminology. ANSI/IEEE Std. 729–1983
- **Greška** (*Error*) - napravi je čovek, na primer, prilikom specifikacije zahteva ili kodiranja programa
- **Mana / defekat** (*Fault*) - posledica greške (na primer, programu nešto nedostaje, ili ima funkciju koja radi neispravno - "Bug")
- **Otkaz** (*Failure*) - nemogućnost sistema da obavi zahtevanu funkciju, najčešće se javlja izvršavanjem defektnog koda
- **Poremećaj** (*Incident*) - simptom koji korisniku obznanjuje otkaz
- **Test primer** (*Test case*) - isputuje određeno ponašanje programa; TP poseduje skup ulaznih vrednosti i skup očekivanih rezultata
- **Testiranje** - postupak provere ispravnosti rada softvera, izvršavanjem test primera; Dva su različita cilja testiranja: da se pronađu otkazi ili da se demonstrira ispravno izvršavanje

Očekivani rezultati testa



- **Predviđanje / predikcija rezultata testa** (*test oracle*)
  - Prediktor testa je mehanizam, nezavisan od samog programa, koji se može upotrebiti za proveru ispravnosti rada programa za test primere.
  - Konceptualno, test primeri se zadaju programu i prediktoru i njihovi izlazi potom međusobno upoređuju.
- Prediktor testa može biti čovek ili automat
  - Ljudi koriste specifikaciju programa da odluče šta je ispravno ponašanje programa. Međutim, specifikacije programa su često sa greškama, nepotpune ili dvosmislene, a i ljudi mogu napraviti previd
  - Automatizovni prediktori koriste formalne specifikacije i tačni su onoliko koliko je specifikacija tačna. Formalne specifikacije često ne postoje za program (nije ih lako napraviti).

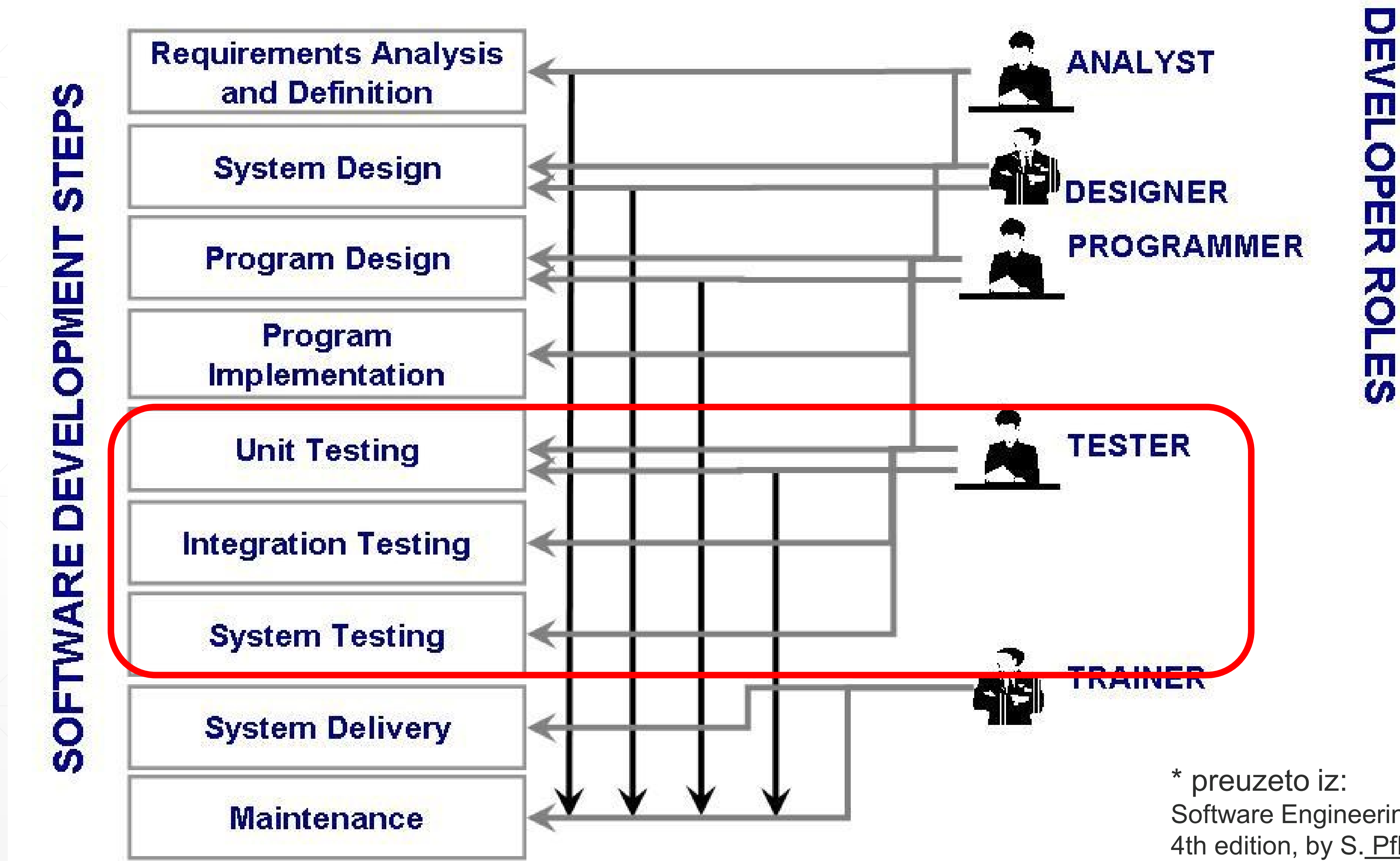
Projektovanje test primera

- Skoro svaki netrivijalni sistem ima ekstremno veliki domen ulaznih podataka => isrcpno testiranje za svaku zamislivu kombinaciju ulaza je nemoguće ili nepraktično.
- Slučajno izabrani test primer može biti bez značaja ako izlaže grešku koja je detektovana nekim drugim test primerom.
- Broj test primera ne određuje koliko je testiranje delotvorno.
- Da bismo uočili grešku u sledećem kodu:

```
if(x>y) max = x; else max = x;
{(x=3, y=2); (x=2, y=3)} zadovoljava
{(x=3, y=2); (x=4, y=3); (x=5, y = 1)} ne zadovoljava
```

- Svaki test primer trebalo bi da razotkrije različitu grešku.

Testiranje softvera

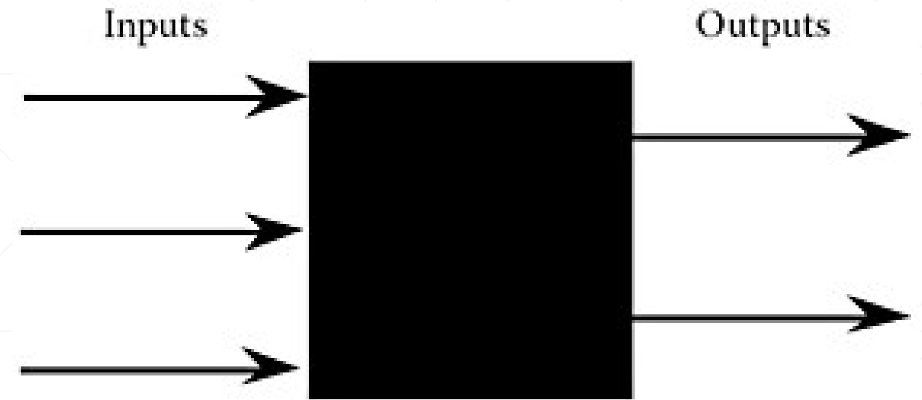


* preuzeto iz:
Software Engineering: Theory and Practice
4th edition, by S. Pfleeger, J. Atlee

Klasifikacije testiranja

- *Prema nivou granularnosti*:
  - Jedinično (eng. *Unit*) testiranje
  - Integraciono (eng. *Integration*) testiranje
  - Sistemsko (eng. *System*) testiranje
- *Prema pristupu testiranja*:
  - Funkcionalno
  - Strukturno

Funkcionalno testiranje



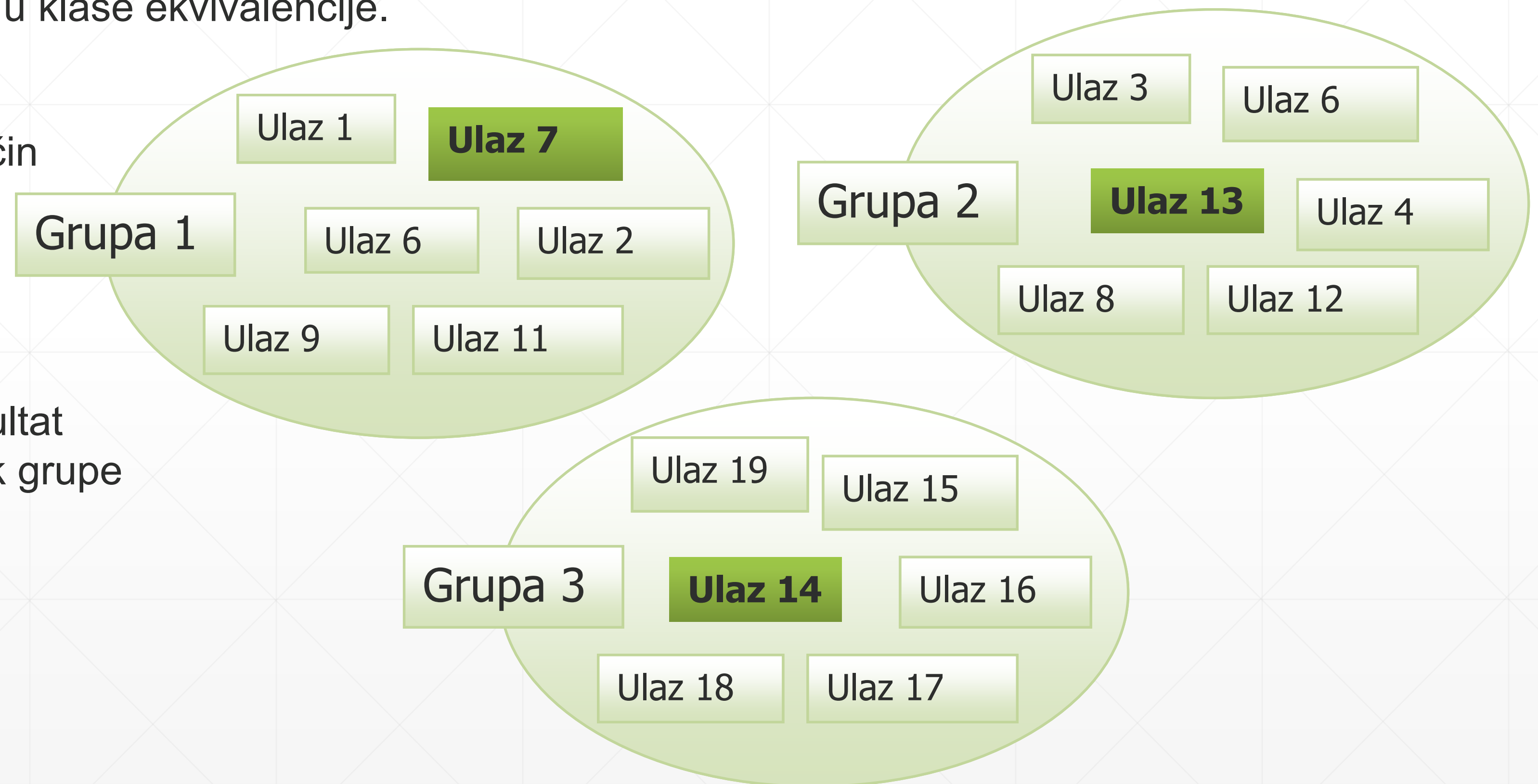
- Program se posmatra kao funkcija koja mapira vrednosti iz ulaznog domena u izlazni.
- Implemenacija nije poznata, program predstavlja **crnu kutiju**.
- Jedina informacija koja se koristi za određivanje test primera je specifikacija programa.
- Prednosti funkcionalnog pristupa:
  - Test primeri su nezavisni od konkretne implementacije, tako da su upotrebljivi i pri promeni implementacije.
  - Razvoj testova može teći u paraleli sa razvojem implementacije, čime se smanjuje potrebno vreme.
- Mane ovog pristupa:
  - Često postoji velika redundantnost testova u seriji.
  - Mogućnost da se deo implementacije ne pokrije testovima (npr. ne može se otkriti da je u program zaražen virusom, pošto ga nema u specifikaciji).

Strukturno testiranje

- Strukturno testiranje fokusira se na implementaciju programa.
  Naziva se još pristup **bele kutije** (providne kutije).
- Cilj nije da se izvrše sve moguće funkcije programa, već da se izvrše/aktiviraju različite programske i strukture podataka u programu.
- Različiti kriterijumi strukturnog testiranja znatno preciznije od kriterijuma funkcionalnog testiranja definišu koje programske strukture treba pokriti (npr. iskaze, odluke, putanje).
- Ovim testiranjem nije moguće otkriti da li neki element specifikacije nije implementiran u programu, pošto se specifikacija ne uzima u obzir za selekciju testova.

Podela na klase ekvivalencije

- Ulazne vrednosti programa se dele u klase ekvivalencije.
- Podela se vrši tako da važi:
  - program se ponaša na sličan način za sve ulazne vrednosti koje pripadaju istoj klasi ekvivalencije
- Grupe međusobno ekvivalentnih ulaznih podataka proizvode isti rezultat iz kojih se bira po jedan predstavnik grupe koji se koristi u tokom testiranja.



Zašto i kako definišemo klase ekvivalencije?

- Zašto? Da bismo testirali program sa po jednom reprezentativnom vrednošću ulaza iz svake klase ekvivalencije
  - Jednako delotvorno kao testiranje bilo kojim drugim vrednostima iz istih klasa ekvivalencije (naći će iste greške).
- U idealnom slučaju podskupovi su međusobno disjunktni i pokrivaju ceo skup ulaza (relacija "sličnosti" ulaza je relacija ekvivalencije)
- Kako odrediti klase ekvivalencije?
  - Posmatraju se svi uslovi vezani za ulaze i izlaze programa koji proizilaze iz specifikacije (tipovi promenljivih, eksplicitna ograničenja).
  - Za svaki uslov se posmatraju dva grupe klasa prema zadovoljenosti uslova:
    - legalne klase obuhvataju dozvoljene situacije.
    - nelegalne klase obuhvataju sve ostale situacije.

Saveti za podelu na klase ekvivalencije

- Ako je ulazni uslov programa definisan u opsegu vrednosti:
  - Na primer, broj između 1 i 5000.
  - Definišu se jedna legalna (1≤broj≤5000) i dve nelegalne klase ekvivalencije: (broj < 1) i (broj > 5000)

- Ako ulazni uslov programa definiše neki fiksni broj:
  - Na primer: jedinstveni matični broj građana (JMBG) ima 13 cifara.
  - Definišu se jedna legalna (mbr_c=13) i dve nelegalne klase ekvivalencije: (mbr_c < 13) i (mbr_c > 13)

Saveti za podelu na klase ekvivalencije (2)

- Ako ulazni podatak uzima vrednosti iz nabrojivog skupa, pri čemu se program različito ponaša za svaku vrednost:
  - Na primer: {a,b,c}
  - Po jedna klasa za svaku dozvoljenu vrednost ulaza (tri klase)
  - Jedna klasa za ulazne vrednosti van dozvoljenog skupa (npr. d).
- Za složene tipove podataka (zapise): Odrediti klase ekvivalencije za svaku komponentu posebno, po potrebi razmotriti kombinacije klasa ekvivalencije.
- Generalno, ako postoji sumnja da se program ne ponaša isto za svaki element već definisane klase ekvivalencije, treba klasu razbiti na više manjih.

Smernice za generisanje klasa ekvivalencije za varijable: opsege i tekst (stringove)

Vrsta	Klasa ekvivalencije	Ograničenje	Reprezentativne klase - primeri
Opseg	Jedna klasa sa vrednostima u okviru opsega i dve klase vrednosti van opsega	brzina ϵ [60 ... 90]	{ {50}, {75}, {92} }
		površina : float površina ≥ 0	{ {-1.0}, {15.52} }
		godine : int 0 ≤ godine ≤ 100	{ {-1}, {56}, {122} }
		slovo : char	{ {J}, {3} }
Tekst	Jedna klasa koja sadrži legalan tekst i jedna koja sadrži nelegalan tekstualni sadržaj. Može se pratiti i dužina stringa.	ime : string (samo tekst) max_lenght = 8	{ {Drazen}, {Drazen5}, {#}, {PredugackoIme} }
		naziv : string (alfanumerici)	{ {Drazen}, {BulevarKraljaAleks73}}

Smernice za generisanje klasa ekvivalencije za varijable: nabrojive tipove i nizove

Vrsta	Klasa ekvivalencije	Ograničenje	Reprezentativne klase - primeri
Enum	Svaka vrednost u zasebnu klasu	auto_color є { crvena, plava, zelena }	{ {crvena}, {plava}, {zelena} }
		uslov : Boolean	{ {true}, {false} }
Niz	Jedna klasa koja sadrži niz sa legalnim elementima, jedna klasa koja sadrži prazan niz, i jedna klasa koja sadrži broj elemenata niza veći od očekivanog.	Java array: int [] niz = new int [3]	{ { [ ] }, { [-10, 20] }, { [-9, 0, 12, 15] } }

Analiza graničnih vrednosti

- Programeri često previđaju:
  - Posebnu obradu koja je potrebna na granicama klasa ekvivalencije
- Na primer, programer može nepravilno napisati < umesto <=
- Analiza graničnih vrednosti:
  - Izabrati test primere na granicama različitih klasa ekvivalencije (jedno ili multidimenzionalnih), ili na osnovu specificiranih veza među ulazima
  - Po jedan test primer za svaku graničnu vrednost

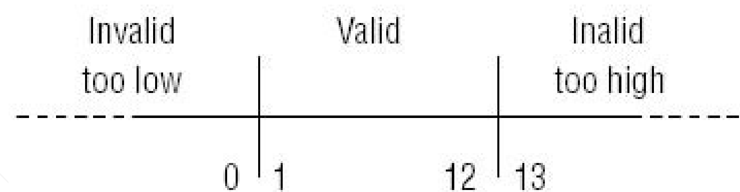
Majersove preporuke za granične vrednosti

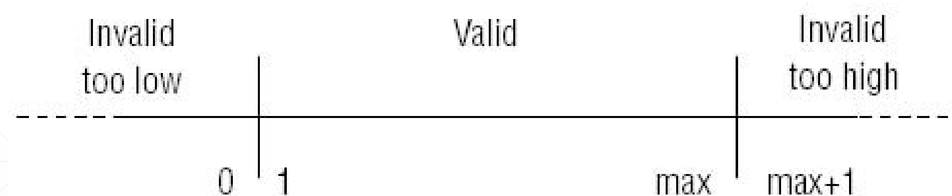
1. Ako ulazni uslov precizira opseg vrednosti, napisati testove za same krajeve opsega i testove nevažećih ulaza za situacije odmah iza legalnih krajeva.
   Na primer: ako je ulaz u opsegu -1.0 do 1.0 testirati treba za -1.0, 1.0, -1.0001, 1.0001 (ako se pretpostavi da je najmanja delta između dva broja koja pravi neku razliku u programu .0001).
2. Ako ulazni uslov precizira broj vrednosti, napisati testove za minimalni i maksimalni broj vrednosti i jedan ispod i iznad ovih vrednosti. Na primer, ako ulazni fajl može da sadrži 1-255 zapisa, napisati testove za 0, 1, 255, i 256 zapisa.
3. Primeniti preporuku 1. na izlazne uslove.
4. Primeniti preporuku 2. na izlazne uslove.
   Na primer: ako na stranu izveštaja staje 65 redova, napisati testove za 64, 65 i 66 redova.
5. Ako je ulaz (ili izlaz) programa uređen skup (sekvencijalni fajl, na primer, ili linearna lista ili tabela), fokusirati pažnju na prvi i poslednji element skupa.
6. Upotrebiti sopstvenu pamet za nalaženje drugih graničnih uslova

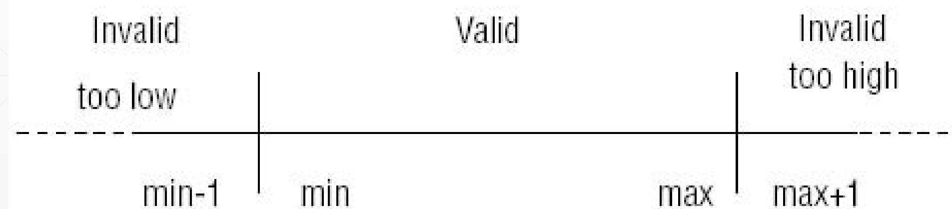
Datum

- Treba voditi računa i o formatu datuma u internoj reprezentaciji
(Y2K problem, UNIX brojač od 1. januara 1970.)

MM/DD/YY

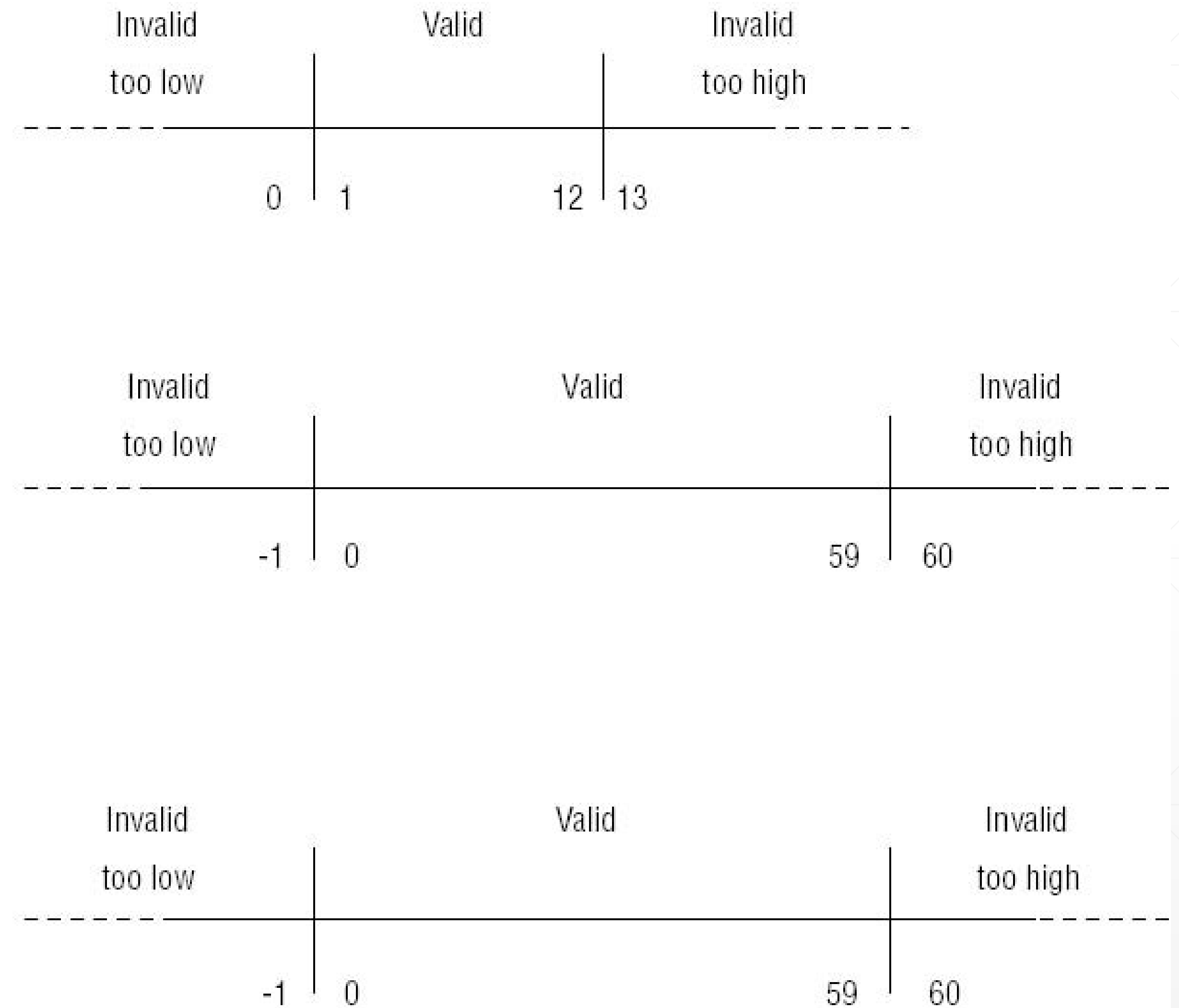






Vreme

- Treba voditi računa o formatima, vremenskim zonama, pomeraju vremena.

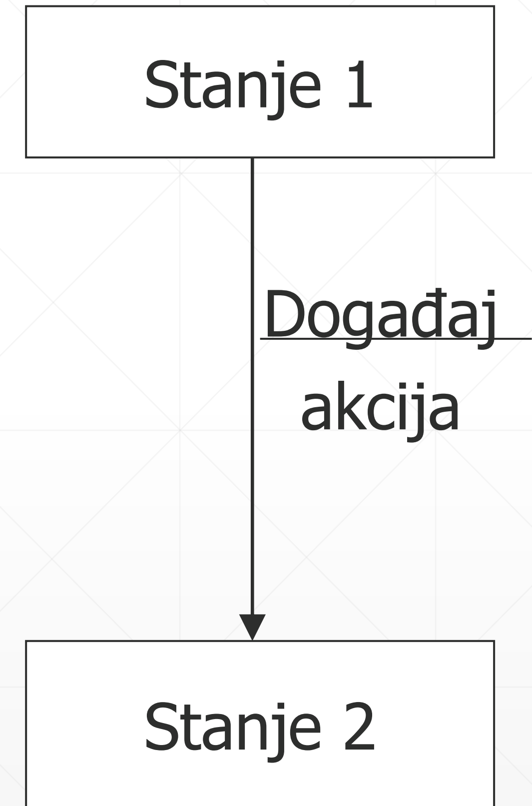


Novčane sume

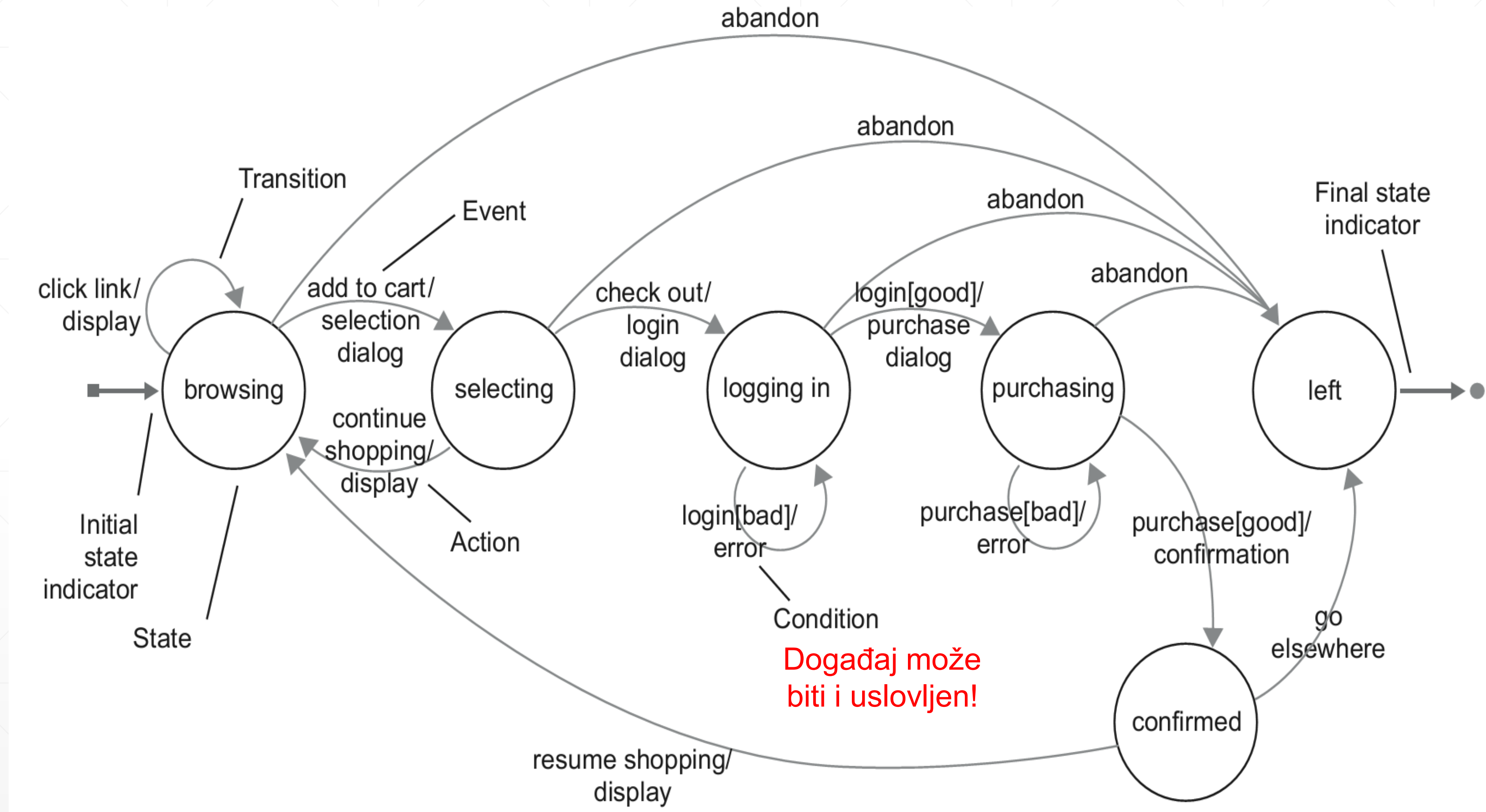
- Često se reprezentuju kao decimalni brojevi sa fiksnim zarezom (imaju dobro definisan epsilon).
- Treba voditi računa o valuti i formatu pisanja.

Model stanja

- Model stanja komponente određuje njena stanja, prelaze među stanjima i sa njima povezane događaje i akcije.
- Događaji su uvek izazvani nekim ulazom, dok akcije proizvode izlaz.
- Model stanja predstavlja se dijagramom promene stanja
  - Prelaz među stanjima definisan je tekućim stanjem i ulaznim događajem i označava se parom događaj/akcija
  - Stanje traje neodređen vremenski period, sve dok se nešto ne desi, spoljašnje u odnosu na sistem.
  - Događaj se desi trenutno ili u ograničenom vremenskom periodu. To je nešto što se desilo, spoljašnja pojava, koja pokreće tranzicije. Događaji se mogu aktivirati na različite načine, kao što su, na primer, od strane korisnika sa tastature ili miša, spoljnog uređaja, ili čak operativnog sistema.
  - Akcija je odgovor sistem tokom tranzicije između stanja. Akcija, kao i događaj, je ili trenutna ili zahteva ograničeno vreme.
    Često akcija može da se posmatra kao sporedni efekat događaja.



Primer: elektronska kupovina artikala



Događaj može biti i uslovljen!

Pokrivanje stanja i prelaza

1. Usvojiti pravilo gde test mora početi i gde mora (ili može) da se završi. Na primer, test mora početi u inicijalnom stanju i može završiti samo u konačnom stanju.

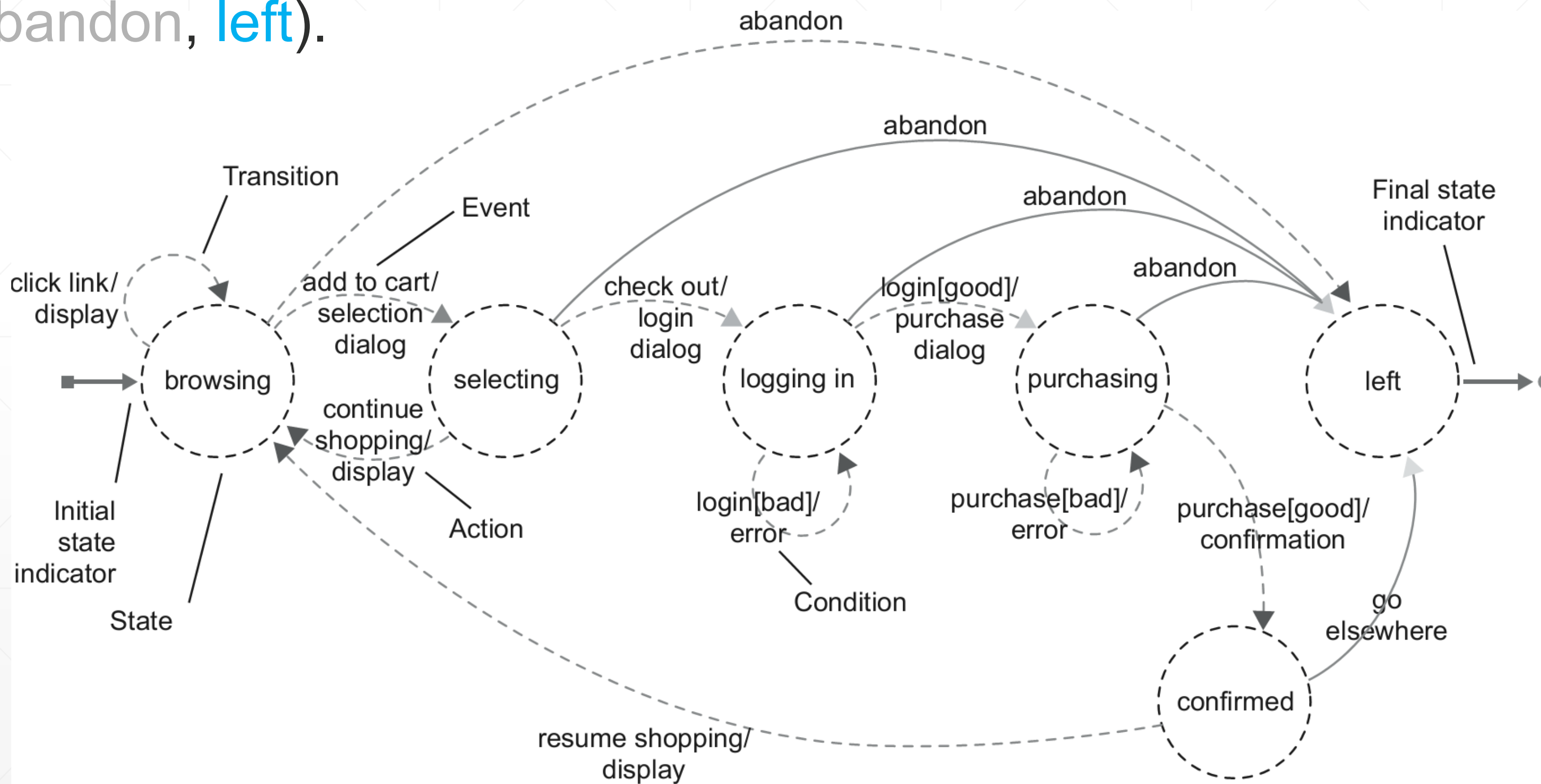
2. Od dozvoljenog početnog stanja definisati sekvencu kombinacija događaj/stanje koja dovodi do dozvoljenog završanog stanja. Za svaku tranzicije koja će se desiti, zabeležiti očekivanu akciju koju bi trebalo da preuzme sistem.
   To je očekivani rezultat testa.

3. Označiti svako stanje i prelaz iz definisanog testa kao pokrivene
   (npr. na test dijagramu).

4. Ponoviti korake 2 i 3 sve dok se ne pokriju sva stanja i svi prelazi.
   a) Ova procedura će generisati logičke slučajeve testiranja. Da bi se napravili konkretni slučajevi testiranja, moraju se usvojiti stvarne vrednosti ulaznih i izlaznih podataka.

Primer pokrivanja stanja / prelaza

TP1. (browsing, click link, display, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[bad], error, login[good], purchase dialog, purchase[bad], error, purchase[good], confirmation, resume shopping, display, abandon, left).



Primer pokrivanja stanja / prelaza (2)

Preostali TP (iskustvena procena za ukupan broj je koliko ima ulaznih grana u finalno stanje):

2. (browsing, add to cart, selection dialog, abandon, <no action>, left)

3. (browsing, add to cart, selection dialog, checkout, login dialog, abandon, <no action>, left)

4. (browsing, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)

5. (browsing, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, purchase[good], confirmation, go elsewhere, <no action>, left)

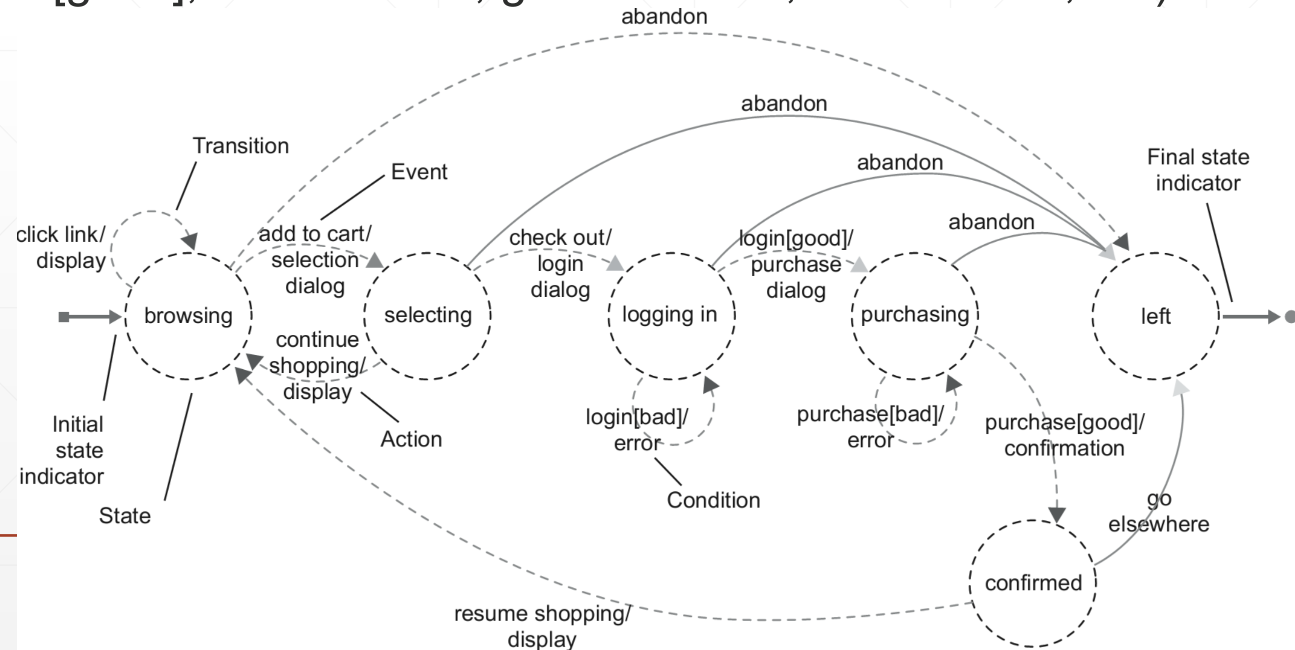


Tabela prelaza

- Tabelarni prikazuje svih kombinacija stanja sa ulaznim događajima / uslovima
- Prikazuju se i dozvoljene i nedozvoljene kombinacije
- Na taj način analizira se šta se događa u nedozvoljenim i nedefinisanim situacijama
- Prvo se izlistaju sva stanja i svi ulazi / uslovi prikazani na dijagramu stanja
- Zatim se kreira tabela koja ima red za svako stanje u kombinaciji sa svakim ulazom / uslovom:

Tekuće stanje	Ulaz / uslov	Akcija	Novo stanje

Primer: elektronska kupovina

Znači mušterija može da uradi checkout samo neposredno posle dodavanja robe (ovo je nedostatak u specifikaciji)

Browsing
Selecting
Logging
Purchasing
Confirmed
Left

×

Click link
Add to cart
Continue shopping
Check out
Login[bad]
Login[good]
Purchase[bad]
Purchase[good]
Abandon
Resume shopping
Go elsewhere

=

Current State	Event/cond	Action	New State
Browsing	Click link	Display	Browsing
Browsing	Add to cart	Selection dia	Selecting
Browsing	Continue shopping	Undefined	Undefined
Browsing	Check out	Undefined	Undefined
Browsing	Login[bad]	Undefined	Undefined
Browsing	Login[good]	Undefined	Undefined
Browsing	Purchase[bad]	Undefined	Undefined
Browsing	Purchase[good]	Undefined	Undefined
Browsing	Abandon	<no action>	Left
Browsing	Resume shopping	Undefined	Undefined
Browsing	Go elsewhere	Undefined	Undefined
Selecting	Click link	Undefined	Undefined
(Fifty-three rows, generated in the pattern shown above, not shown)			
Left	Go elsewhere	Undefined	Undefined

Primer: elektronska kupovina

- Da bismo pokrili slučaj: Browsing, checkout, undefined, undefined
- Kreće se od legalnog inicijalnog:
  - TP4. (browsing, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)
- Koji se modifikuje na sledeći način:
  - (browsing, *attempt: check out, action undefined*, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)
- Ideja je da svaki nedefinisani red tabele pokrijemo posebnim prelazom, slično kao kod klasa ekvivalencije, da ne bi jedna greška prikrila drugu (pitanje je da li bi sistem stigao do drugog prelaza).

Jedinično testiranje

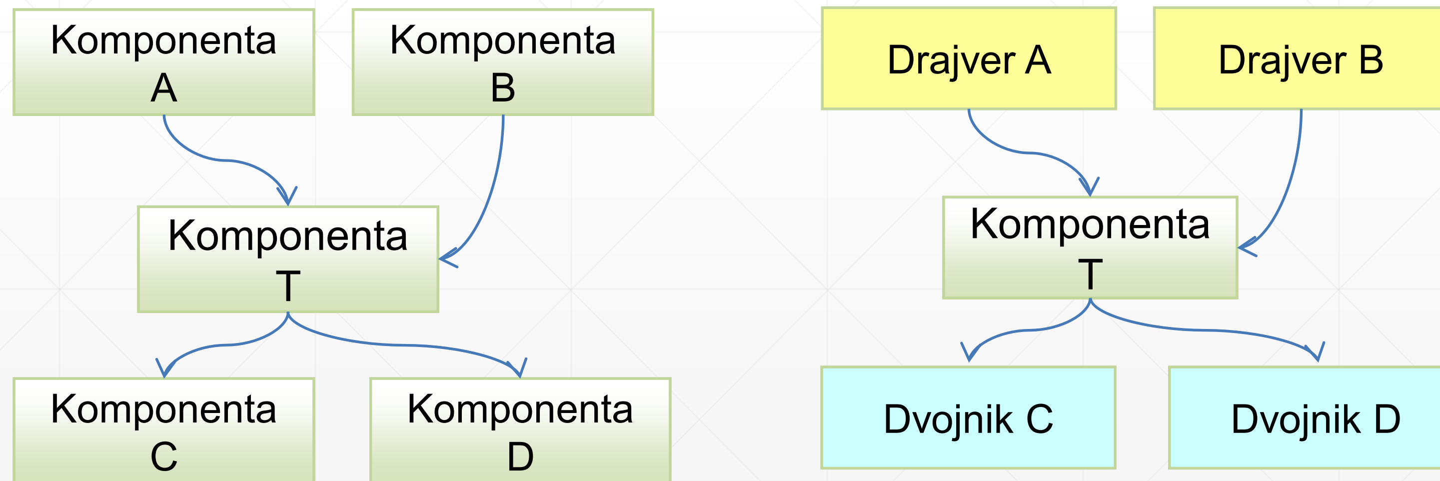
Unit testing

Uvod

- Jedinično testiranje predstavlja testiranje izolovanih celina (komponenti) u sistemu.
- Uslov je da se komponenta koja se testira može posmatrati kao nezavisna celina koja se može izvući iz konteksta sistema i testirati izolovano od ostalih komponenti.
- Komponentama se mogu smatrati klase, moduli, paketi, čak i fragmenti koda.
- Komponente mogu biti ili deo korisničkog interfejsa (strane ili forme) ili komponente koje vrše neke akcije i procesiraju rezultate (na primer komponente za implementaciju algoritama, dohvatanje podataka, komunikaciju sa ostalim sistemima).

Okruženje za testiranje pojedinačne komponente

- Komponenta koja se testira se izvlači iz konteksta sistema u kome komunicira sa ostalim komponentama i ubacuje se u test kontekst koji je simulacija pravog sistema gde se nalaze komponente koje simuliraju rad stvarnih komponenti.
- Primer je prikazan na slici – komponenta koja se testira T u realnom okruženju biva pozvana od strane komponenti A i B kojima pri pozivu vraća neke podatke i poziva komponente C i D koje joj daju informacije kada ih komponenta pozove.



Okruženje za testiranje pojedinačne komponente (2)

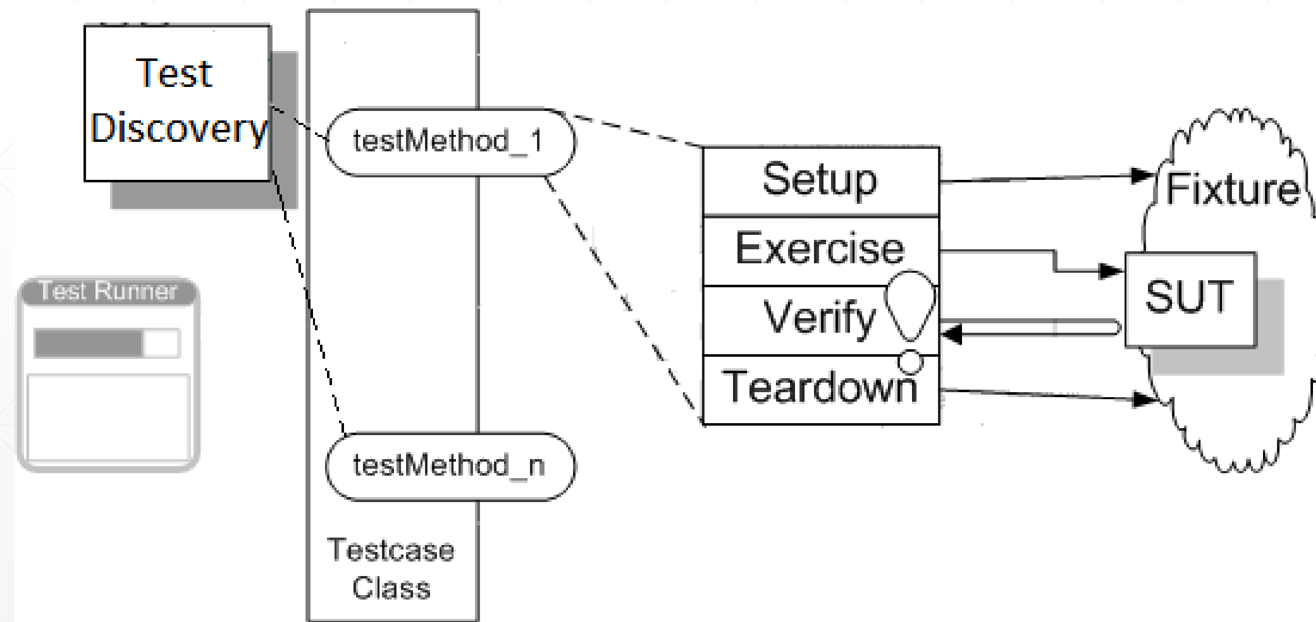
- Umesto sa realnim komponentama iz svog okruženja, komponenta u test kontekstu komunicira sa simuliranim komponentama koje je ili pozivaju ili joj odgovaraju na pozive na isti način kao i realne komponente. U zavisnosti od toga da li simulirane komponente pozivaju ili bivaju pozvane one se dele na dve vrste:

- **Pokretači (Drajveri)** – komponente koje simuliraju rad realnih komponenti koje pozivaju druge komponente i očekuju neki odgovor. Ove komponente pokreću test pošto iniciraju pozive ka komponenti koja se testira. Za realizaciju drajvera često se koriste alati familije xUnit.

- **Dvojnici (Doubles)** – komponente koje simuliraju rad realnih komponenti koje primaju pozive i vraćaju iste rezultate kao i realne komponente.

Radni okviri za jedinično testiranje - xUnit

- Ovi radni okviri (engl. *framework*) predstavljaju alat koji olakšava automatizaciju jediničnog testiranja, na taj način što omogućavaju pisanje test skriptova, specifikaciju očekivanih rezultata, grupisanje testova u serije testova i izvršavanje jednog ili serije testova da bi se dobio izveštaj o rezultatima testiranja.



Arhitektura xUnit radnog okvira

- TestRunner je aplikacija iz komandne linije ili sa GUI-jem koja izvršava metode označene kao testovi i tretira ih kao uspešno izvršene (pass) ako nijedno tvrđenje (engl. assertion) nije narušeno i ako nije došlo do grešaka u izvršavanju (dakle čak i prazan test je uspešan).
- Okviri za različite jezike na različite načine rade pronalaženje test metoda za izvršavanje, najčešće preko metapodataka (anotacija ili atributa). Kod nekih jezika naziv metoda mora da počinje sa test. kod nekih mora čak programski da se napravi test Factory.
- Okvir ne pruža garancije o redosledu izvršavanja pojedinačnih test metoda (što ujedno znači da oni ne smeju biti međusobno spregnuti).

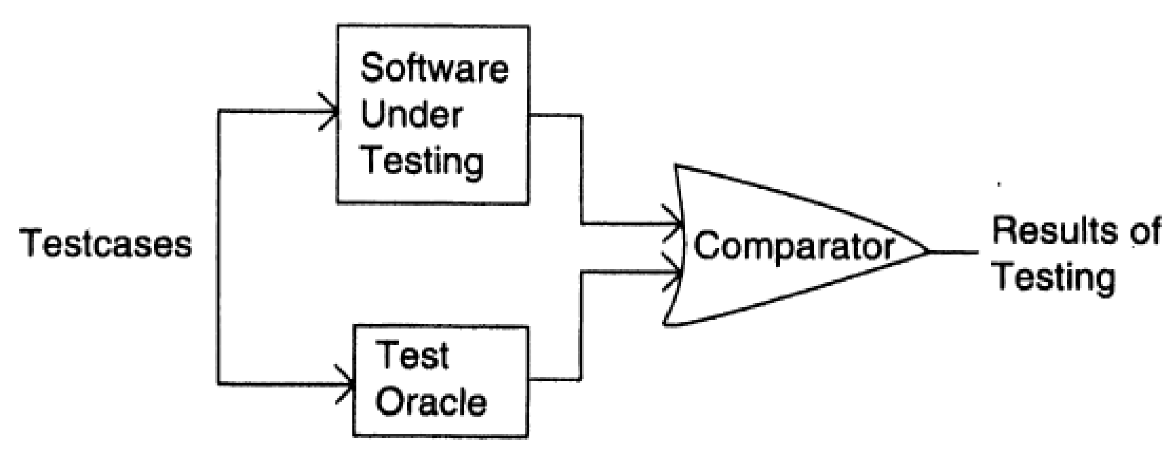
Izvršavanje test metoda

- Izvršavanje xUnit test metoda odvija se u četiri koraka;

1. Da bi bili ponovljivi, jedinični testovi se moraju izvoditi iz poznatog početnog stanja. Fiksno stanje programskih elemenata od kojih test zavisi naziva se ***test fixture***, a svrha ***setup*** koda je da ga postavi.
2. Sledeći korak je izvršavanje sistema koji se testira (engl. system under test, SUT) i interakcija sa njime
3. Zatim sledi verifikacija ponašanja SUT, to jest utvrđivanje da li se sistem ponaša prema očekivanjima
4. Posle svakog testa zove se ***teardown*** kod koji treba da pospremi (ukloni) **test fixture**.

Verifikacija rezultata testova “teorije”

- Predviđanje, predikcija rezultata testa (test oracle)
  - Prediktor testa je mehanizam, nezavisan od samog programa, koji se može upotrebiti za proveru ispravnosti rada programa za test primere.
  - Konceptualno, test primeri se zadaju programu i prediktoru i njihovi izlazi potom međusobno upoređuju
  - Problem određivanja očekivanih izlaza je takozvani **test oracle problem**



Verifikacija rezultata testova “teorije” (2)

- Jedan od načina za realizaciju prediktora je alternativna implementacija istog algoritma (tada se mogu neposredno verifikovati izlazne vrednosti). Da bi se izbeglo ponavljanje istih grešaka, najbolje je da alternativnu implementaciju radi neko drugi mimo originalnog implementatora, ili koristeći alternativni algoritam (npr. *insertion sort* proveravati *quick sort*) ili neki sasvim drugi programski jezik.
- Ako nemamo resurse za alternativnu implementaciju, tada nećemo neposredno verifikovati izlaze, nego nešto manje precizno, samo neke osobine izlaznih rezultata.

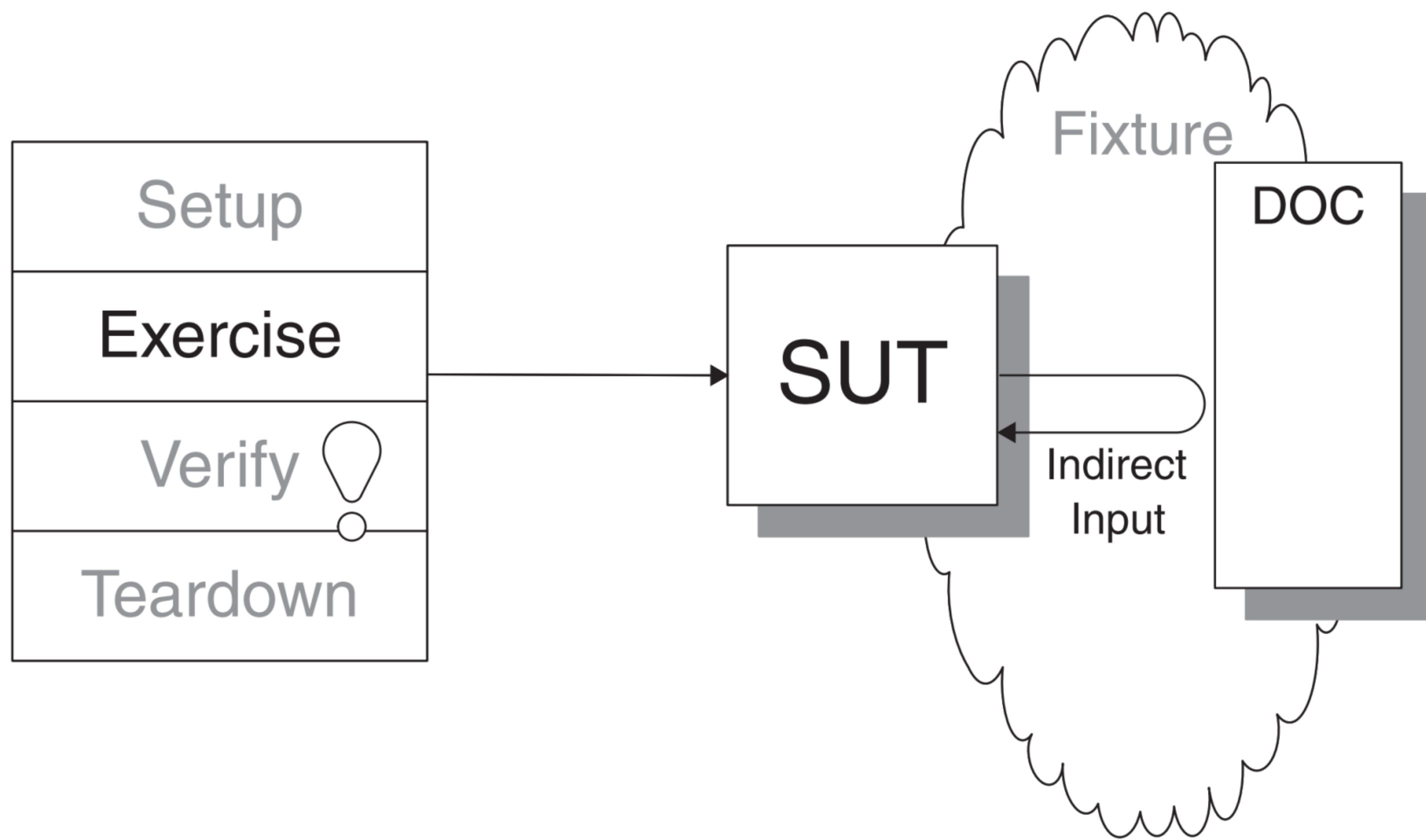
Verifikacija rezultata testova “teorije” (3)

- Primeri provera tvrđenja koja se mogu implementirati u testovima:
  - **Granice**: testirati da li rezultat uvek spada u određene poznate granice. Primer je ranije navedeni generativni test sa premijama osiguranja.
  - **Greške i izuzeci**: Da li se ulaz obrađuje bez grešaka i izuzetaka. Da li izlaz nije null.
  - **Kompleksne optimizacije**: ako neki kod radi nešto jednostavno na vrlo komplikovan način da to učini brzim, može se uključiti jednostavna implementacija u test i proveriti da li se izlazi podudaraju
  - **Komplementarni parovi**: ako imate par funkcija kao što su kodiranje i dekodiranje, može se proveriti da li kodiranje pa potom dekodiranje reprodukuje originalni ulaz.
  - **Idempotencija**: može se proveriti da li pozivanje funkcije više puta sa istim ulazom ne menja njen izlaz (osim ako je tako predviđeno).
  - **Komutativnost**: može se proveriti da li redosled argumenata menja rezultat ili ne. Na primer, funkcija sabiranja treba da se ponaša na ovaj način.
  - **Invarijante**: mogu se proveravati invarijante specifične poslovnoj logici (stvari koje uvek treba da važe, na primer da neki podatak u nekoj klasi nije null ili ako se pamti opseg datuma da posle svakog ažuriranja važi da je početak opsega raniji od kraja opsega).

Test dvojnici

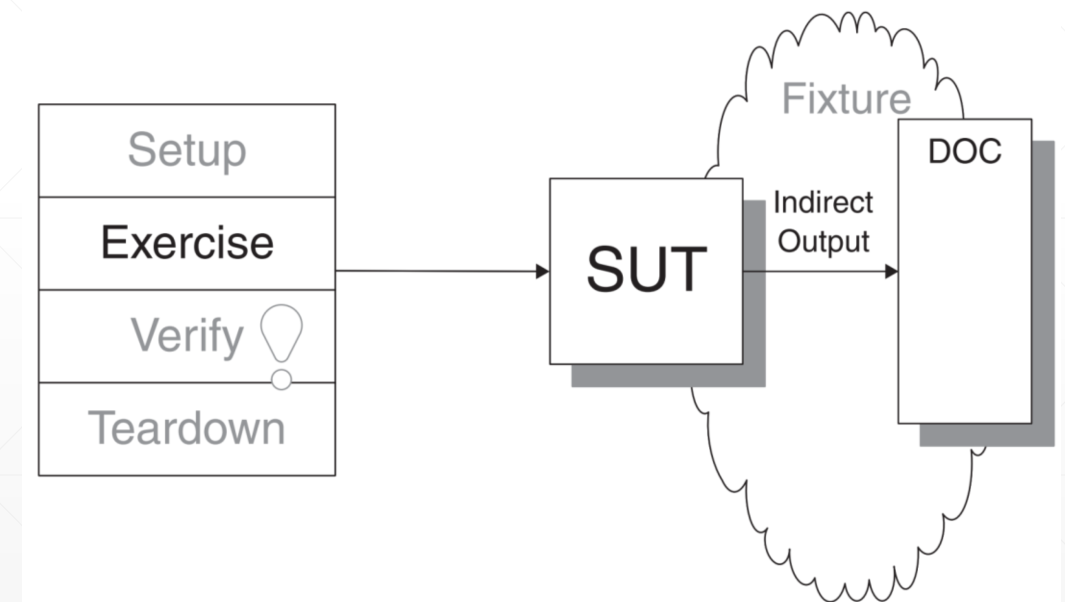
Uvod

- Klasa koju testiramo (SUT) može zavisiti od jedne ili više drugih klasa (tzv. depended-on component, DOC).



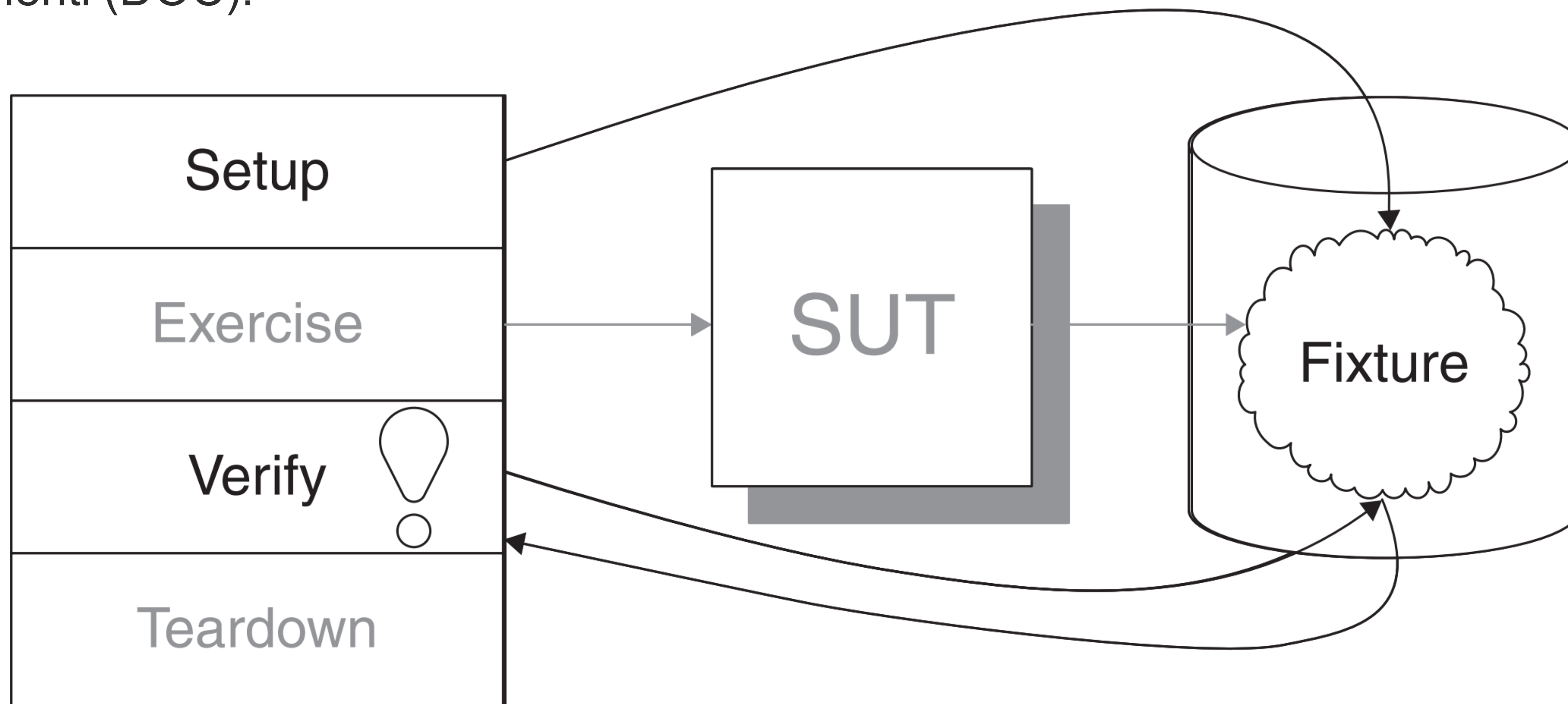
Problem indirektnih ulaza

- Klasa kolaborator (DOC) može vraćati vrednosti ili bacati izuzetke koji utiču na ponašanje SUT-a, ali kod nje može biti teško ili nemoguće prouzrokovati određene slučajeve ponašanja.
- Indirektni ulazi primljeni od DOC-a mogu biti nepredvidljivi (poput sistemskog sata ili kalendara).
- U drugim slučajevima, DOC možda neće biti dostupan u test okruženju ili možda čak i ne postoji.
- Nisu svi izlazi iz SUT direktno vidljivi test metodu. Ti takozvani indirektni izlazi šalju se drugim komponentama (DOC) u formi poziva metoda ili poruka.



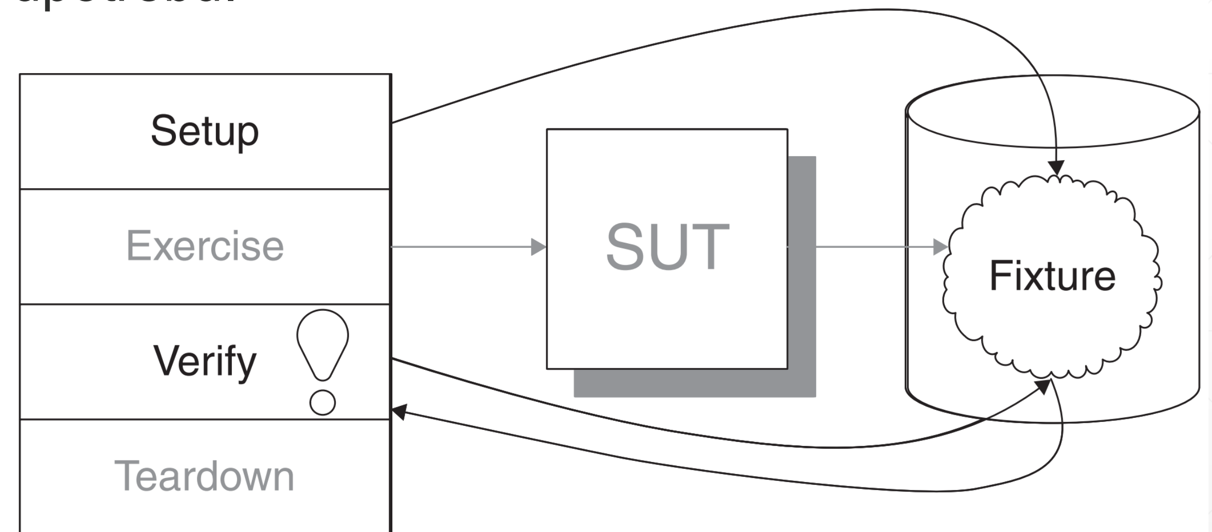
Kontrola indirektnih ulaza i izlaza

- **Back door tehnika**: ovo rešenje može se primeniti ako SUT pamti svoje stanje u drugoj komponenti (DOC).



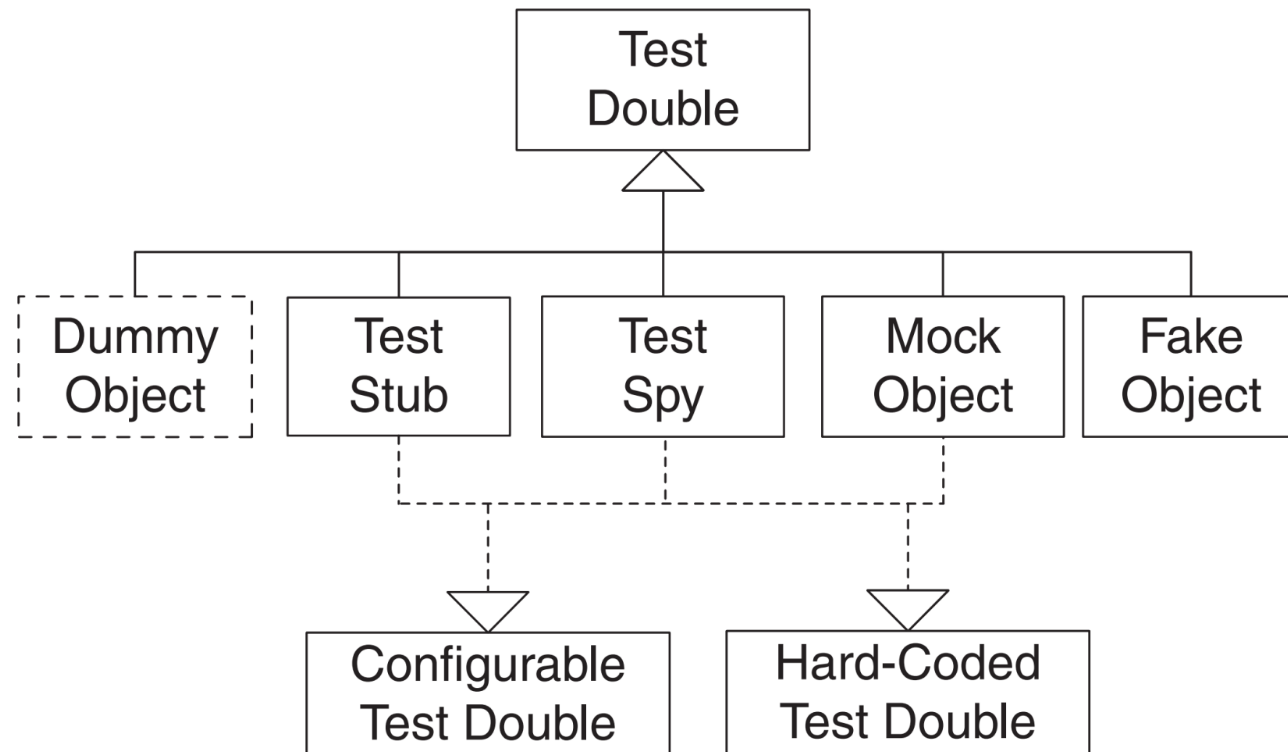
Kontrola indirektnih ulaza i izlaza (2)

- Međutim, u većini slučajeva ova tehnika nije primenjliva:
  - realna komponenta DOC ne može se manipulisati da proizvede željeni indirektni ulaz
  - realna komponenta DOC može se manipulisati za željeni indirektni ulaz, ali to bi proizvelo neprihvatljive bočne efekte
  - realna komponenta još i nije raspoloživa za upotrebu.
- U ovim slučajevima rešenje je upotreba **test dvojnika** realne DOC komponente.



Vrste test dvojnika

- **Test dvojnik** je bilo koji objekat illi komponenta kojom možemo zameniti realnu komponentu u svrhu izvršavanja testa.



Vrste test dvojnika (2)

- **Dummy (prazan?) objekat** koristi se da popuni mesto nekog parametra koji se prosleđuje SUT prilikom testiranja, ali se taj parametar ne upotrebljava (ne utiče na test).
- **Test talon** (**test stub**) je objekat koji menja realnu komponentu da bi se kontrolisali indirektni ulazi u SUT.
- **Test špijun** (**test spy**) je naprednija verzija test talona koja omogućava i verifikaciju indirektnih izlaza SUT na taj način što omogućava test metodu da ih proveri posle izvršavanja SUT.
- **Probni (mock) objekat** menja realnu komponentu da bi se verifikovali indirektni izlazi. **.**
  U probni objekat su ugrađena tvrđenja da porede pozive iz SUT i njihove stvarne parameter sa očekivanim.
- **Lažni** (**fake**) objekat menja funkcionalnost realnog DOC alternativnom jednostavnijom implementacijom.

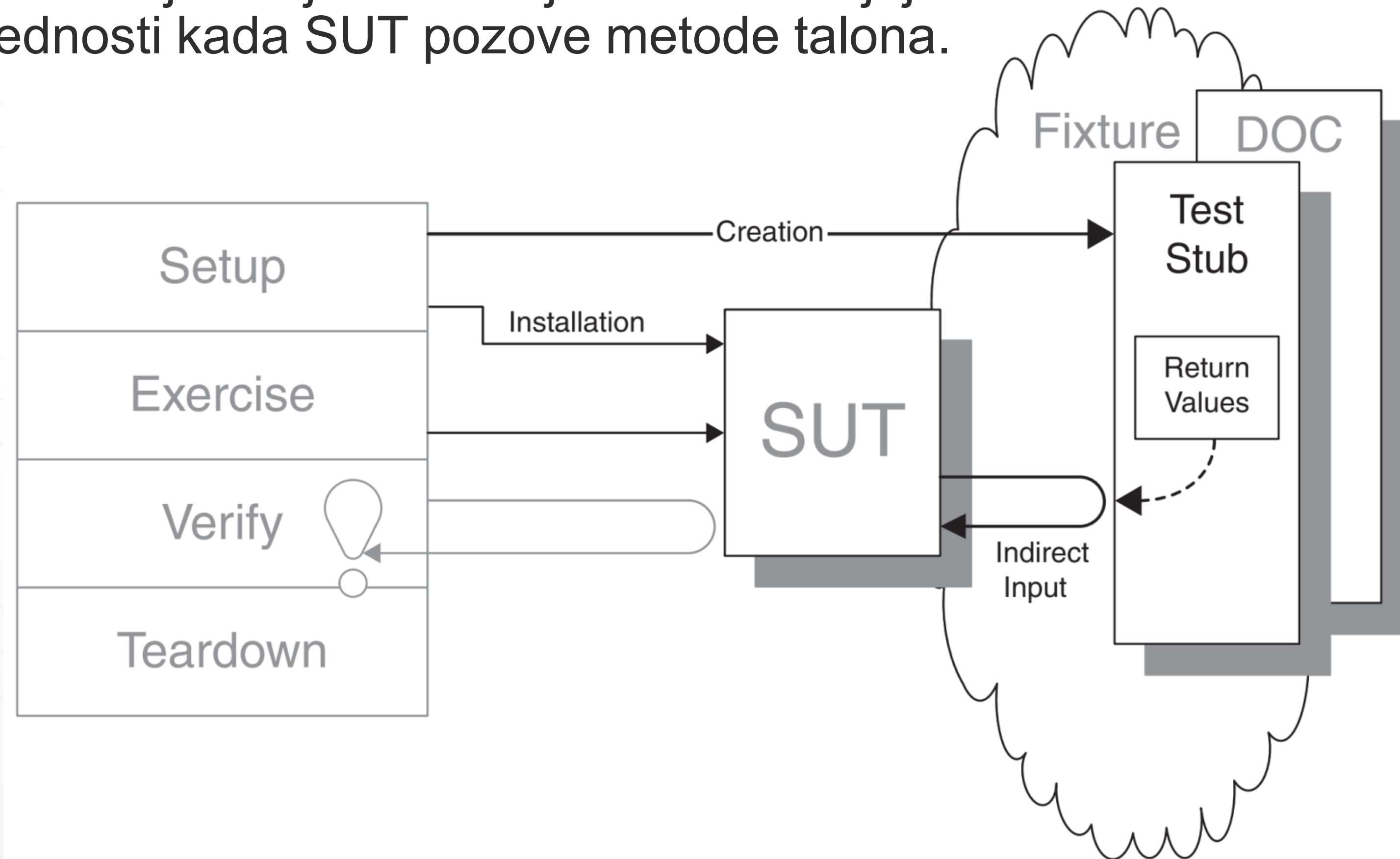
Dummy objekat

- **Dummy objekat** koristi se da popuni mesto nekog parametra koji se prosleđuje SUT prilikom testiranja, ali se taj parametar ne upotrebljava (ne utiče na test).

```
public void testInvoice_addLineItem_DO() {
  final int QUANTITY = 1;
  Product product=new Product("Dummy Product Name", getUniqueNumber());
  Invoice inv = new Invoice( new DummyCustomer() );
  LineItem expItem = new LineItem(inv, product, QUANTITY);
  // Exercise
  inv.addItemQuantity(product, QUANTITY);
  // Verify
  List lineItems = inv.getLineItems();
  assertEquals("number of items", lineItems.size(), 1);
  LineItem actual = (LineItem)lineItems.get(0);
  assertLineItemsEqual("", expItem, actual);
}
```

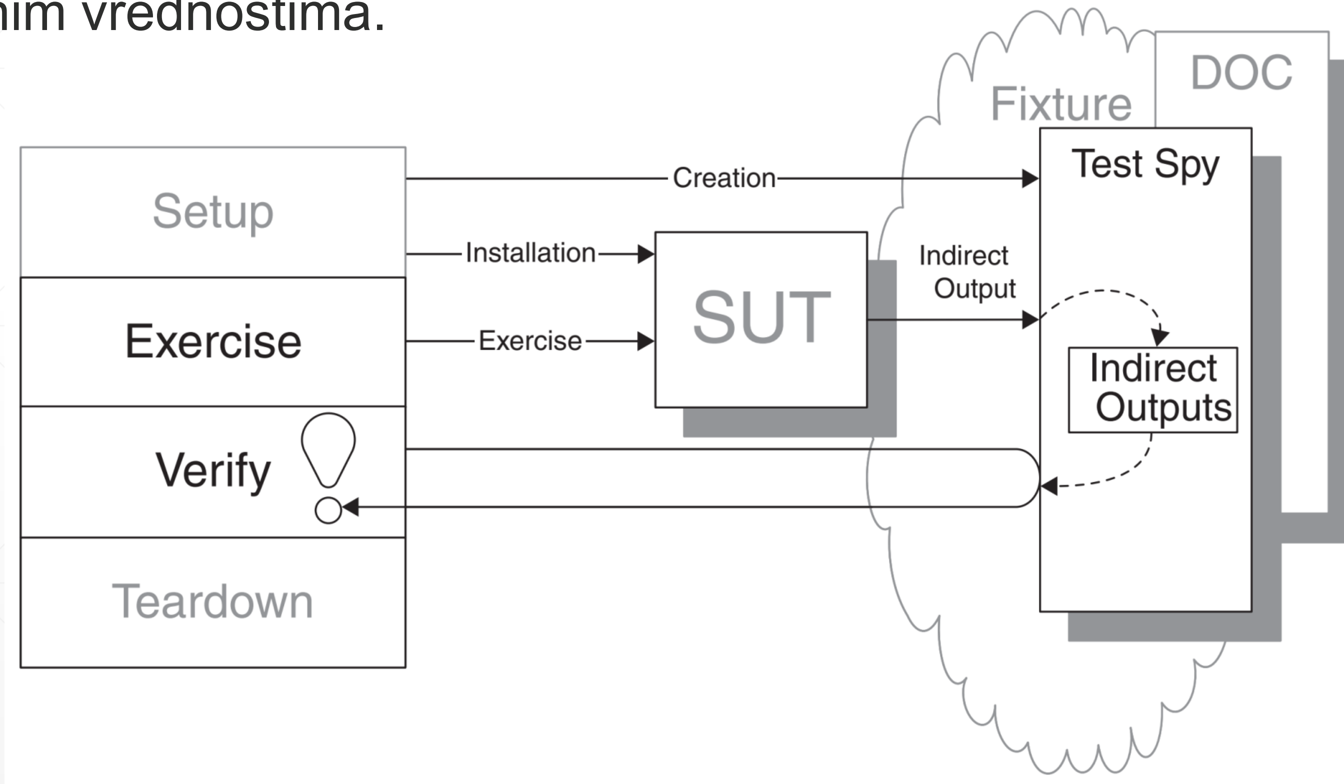
Test talon (*stub*)

- Test talon je objekat koji menja DOC kojim se dostavljaju vrednosti indirektnih ulaza SUT kao povratne vrednosti kada SUT pozove metode talona.



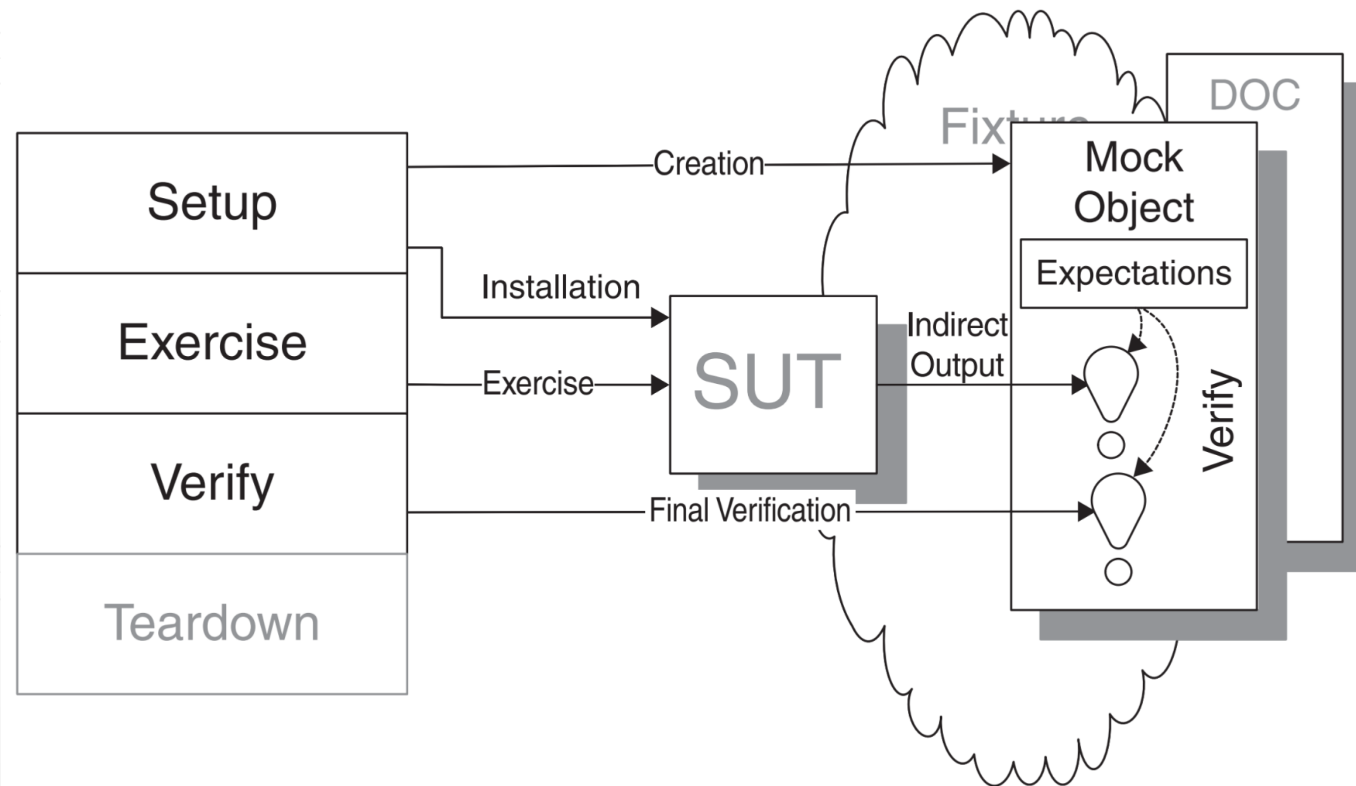
Test špijun (*spy*)

- Test špijun koji menja DOC tokom testiranja beleži pozive metoda koje vrši SUT tokom izvršavanja testa. U fazi verifikacije rezultata, porede se vrednosti zabeležene od strane test špijuna sa očekivanim vrednostima.



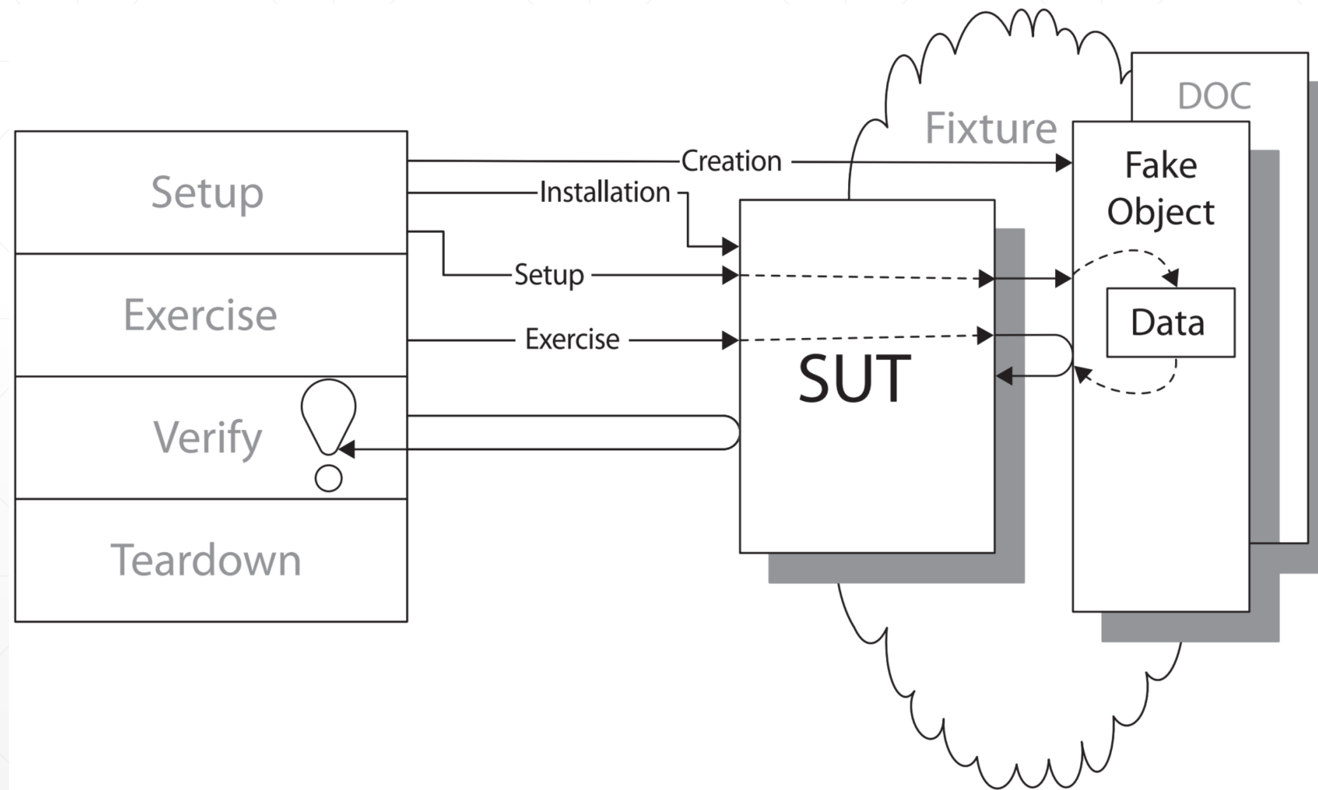
Mock (probni) objekat

- Probni objekat menja DOC tokom testiranja da bi se mogli pratiti indirektni izlazi.
  U probni objekat su ugrađena tvrđenja da porede pozive iz SUT i njegove stvarne parametre sa očekivanim.



Lažni (*fake*) objekat

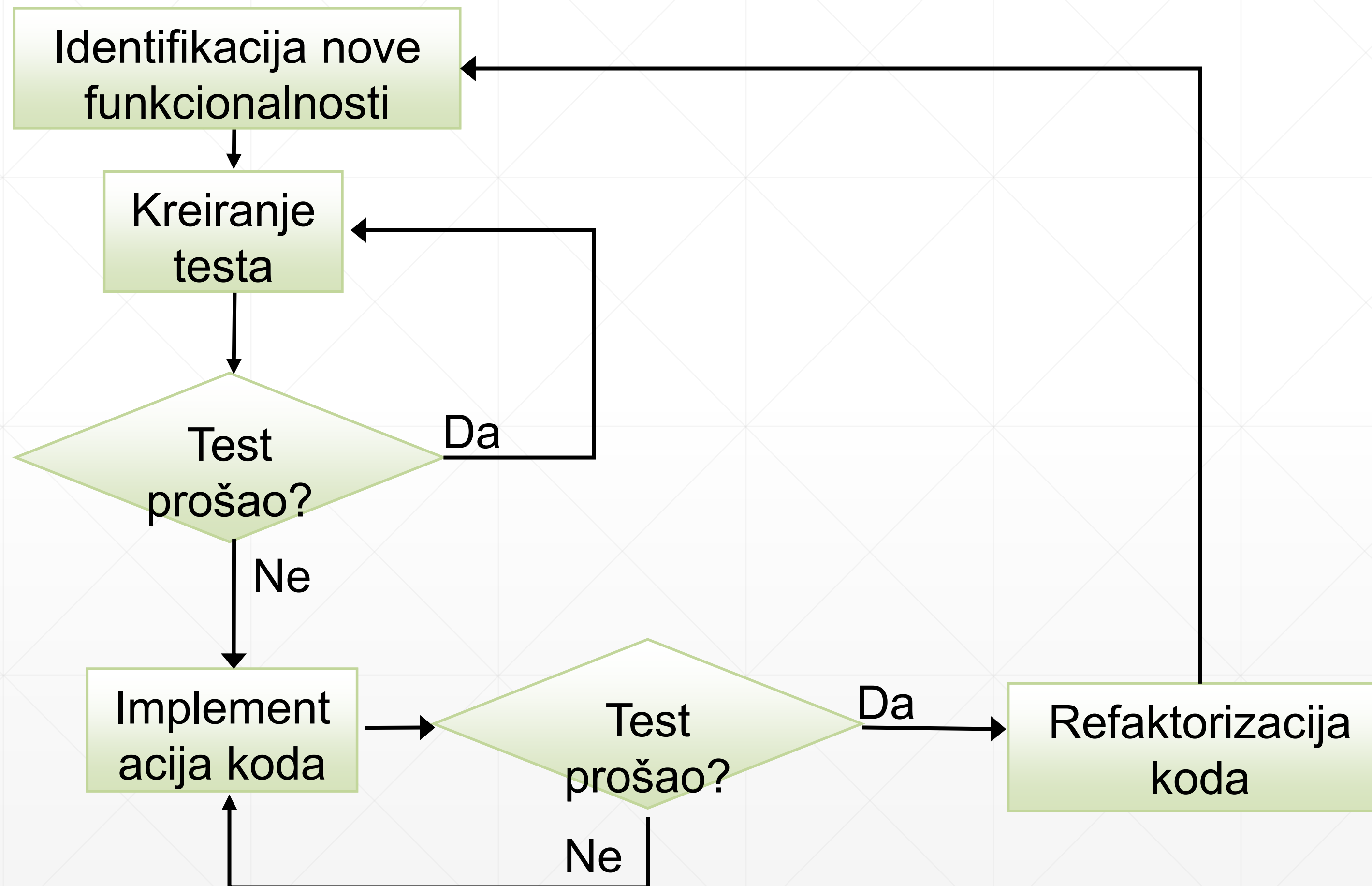
- Lažni objekat je jednostavnija i često brža implementacija realnog DOC koja predstavlja funkcionalnosti DOCa bez bočnih efekata koje želimo da uklonimo.



Razvoj vođen testovima

Test driven development

Razvoj vođen testovima



Proces razvoja vođenog testovima

Koraci u procesu TDDa su sledeći:

1. Izbor inkrementa zahtevane funkcionalnosti. On treba normalno da bude mali i da može da se isprogramira u nekoliko linija koda.
2. Piše se test za ovu funkcionalnost i implementira u vidu automatizovanog testa. To znači da test može biti izvršen i daće izveštaj da li je prošao ili nije.
3. Zatim se pokrene test, zajedno sa svim ostalim testovima koji su već napisani. U početku, nije realizovana funkcionalnost tako da novi test neće uspeti. To je tako namerno, jer pokazuje da test dodaje nešto skupu testova. Ovo je takozvana crvena faza.
4. Zatim se isprogramira funkcionalnost i ponovo pokrene test. Cilj je da test se test sada uspešno izvrši. Ovo je takozvana zelena faza.
5. Zatim se radi refaktorisanje postojećeg koda. Rastuća baza koda mora se redovno čistiti tokom razvoja vođenog testovima. Novi kod se može premestiti sa mesta gde je bilo zgodno za zadovoljenje testa tamo gde logičnije pripada. Duplikat mora biti uklonjen. Imena objekata, klasa, modula, promenljivih i metoda treba jasno da predstavljaju njihovu trenutnu namenu i upotrebu, jer se dodaje dodatna funkcionalnost. Kako se funkcije dodaju, tela metoda mogu da se povećavaju i članovi klase dodaju. Korisno je razdeliti i pažljivo imenovati njihove delove radi poboljšanja čitljivosti i pogodnosti koda za održavanje kasnije u životnom ciklusu softvera. Nasledne hijerarhije mogu se preurediti kako bi bile logičnije, a i da bi se ubacili prepoznati obraci dizajna (*design patterns*).
6. Kada svi testovi uspešno prođu, prelazi se na implementaciju sledećeg inkrementa funkcionalnosti.

Preporuke pisanja testova

Poželjno je testove nove funkcionalnosti pisati sledećim redom:

1. **Granični slučaj** - započnite sa testom koji radi na „praznoj“ vrednosti poput nula, null, prazan niz ili slično. Pomoći će vam da zadovoljite interfejs osiguravajući pritom da se vrlo brzo može doneti.

2. **Jedan ili nekoliko testova osnovnog uspešnog scenarija** tzv. *happy path* tests - Takav test / testovi postaviće temelje implementacije, dok ostaju fokusirani na osnovnu funkcionalnost.

3. **Testovi koji pružaju nove informacije ili znanje** - Ne kopajte na jednom mestu. Pokušajte da pristupite rešenju iz različitih uglova tako što ćete napisati testove koji aktiviraju različite njegove delove i koji će vas naučiti nečemu novom o problemu.

4. **Rukovanje greškama i negativni testovi** - Ovi testovi presudni su za ispravnost, ali retko za dizajn. U mnogim slučajevima mogu se bez opasnosti pisati na kraju.

Strategije prelaska iz crvenog u zeleno

1. **Lažiranje** - Ovo je najjednostavniji način da test prođe. Samo se vrati ili uradi šta god određeni test očekivao. Ako test očekuje određenu vrednost, onda se samo vrati konstanta. Testovi koji prođu nakon lažiranja obično padaju kada se doda sledeći test želi nešto što nije konstantna vrednost ili fiksno ponašanje.

2. **Očigledna implementacija** - Ako znamo šta treba da se implementira, onda to prosto treba uraditi. Očigledna implementacija obično podrazumeva uzimanje malo većeg razvojnog koraka. Međutim, ako se ode do krajnjih granica kucanjem čitavog algoritma odjednom, to može dati crvenu fazu tj. da neki testovi ne prođu. Onda se mora pribegavati razvoju sa uklanjanjem grešaka degagovanjem, što je vraćanje na stare, loše navike.

3. **Triangulacija** - Neki algoritmi se mogu implementirati davanjem određenog broja konkretnih primera i zatim uopštavanjem rešenja. Ovo se naziva triangulacija i ima svoje korene u geometriji. Razumno je da se jedan test lažiranjem učini zelenim, dok će veći broj testova sa različitim parametrima i očekivanim rezultatima potisnuti kod u smeru opšteg algoritma.

Primer primene stategije triangulacije

- Želimo da napišemo program koji će izračunavati površinu kvadrata dužine stranice *edge*.
- Počinjemo od graničnog testa, za dužinu stranice 0.

```
@Test
public void testAreaWithZeroEdge() throws Exception {
   Square s = new Square(0.0);
   assertEquals(0.0, s.area(), 0.0); // treći parametar je delta
}
```

- Lažiranjem zadovoljavamo test:

```
class Square {
   Square(double edge) {}
   public double area() {
         return 0.0;
   }
}
```

Primer primene stategije triangulacije (2)

- Sada dodajemo novi granični test, za dužinu stranice 1.

```
@Test
public void testAreaWithEdgeOfOne() throws Exception {
  Square s = new Square(1.0);
  assertEquals(1.0, s.area(), 0.0);
}
```

- Test pada za originalno rešenje. Moramo smisliti da rešenje zadovolji oba testa. Edge jednako 0 daje površinu 0, edge jednako 1 daje površinu 1. Možemo napisati da je površina jednaka edge, što će zadovoljiti oba testa.

```
class Square {
  private double edge;
  Square(double edge) { this.edge = egde; }
  public double area() {
        return edge;
  }
}
```

Primer primene stategije triangulacije (3)

- Nastavljamo, sada dodajemo novi test, za dužinu stranice 2.

```
@Test
public void testAreaWithEdgeOfTwo() throws Exception {
  Square s = new Square(2.0);
  assertEquals(4.0, s.area(), 0.0);
}
```

- Test pada za tekuće rešenje. Dakle korak po korak, krenuli smo od ničega ka 0.0, zatim od 0.0 ka edge, i konačno ćemo edge da zamenimo formulom edge “ edge, čime dobijamo finalno generalno rešenje.

```
class Square {
  private double edge;
  Square(double edge) { this.edge = egde; }
  public double area() {
        return edge * edge;
  }
}
```

Pitanja?

Hvala na pažnji.