Testiranje objektno-orijentisanog softvera

Jedinično testiranje OO softvera

- Konvencionalni softver:
  - Počinjemo sa jediničnim testiranjem (modula, procedura i/ili komponenata)
  - Sledi integracija komponenata (strategije integracionog testiranja i regresivno testiranje) da se otkriju greške u interfejsima modula i bočni efekti dodavanja novih modula
  - Na kraju se sistem testira kao celina da se otkriju eventualne nesaglasnosti u odnosu na početne zahteve (testiranja višeg reda)

Jedinično testiranje OO softvera

- Najmanja jedinica testiranja je klasa tj. objekti koji obuhvataju podatke i funkcije za manipulaciju tim podacima.
- Testiranje klase za OO softver je ekvivalent jediničnog testiranja kod konvencionalnog softvera.
- Metod klase ne može se testirati izolovano (kao kod konvencionalnog jediničnog testiranja) nego samo u kontekstu cele klase (i njenih osnovnih klasa).

Ilustrativni primer

```
class Base {
public:
     int func() { return x + vfunc();}
     int func2(){ return x+5; }
     virtual int vfunc() { return 0; }
     Base(): x(10) {}
private: int x;
};

class Derived: public Base {
public:
     virtual int vfunc() { return 1; }
};

Base b;
Derived d;
```

- Funkcija **func()** daje različite rezultate ako se pozove u kontekstu klase Base, odnosno Derived, iako postoji jedinstvena implementacija
- **b.func()** daje rezultat 10
- **d.func()** daje rezultat 11

Jedinično testiranje OO softvera

- U svakoj od podklasa metod X se poziva u kontekstu operacija i atributa koji su definisani za tu podklasu.
- Znači taj kontekst je različit u opštem slučaju, pa je potrebno X testirati u svim tim kontekstima.
- Za metode za koje smo sigurni da im je kontekst definisan u nekoj klasi i da je nepromenljiv, nije potrebno testiranje u kontekstu podklasa (prethodni primer **func2**).

Karakteristike OO softvera od uticaja na testiranje

- **Ponašanje zavisno od stanja**. Testiranje mora uzeti u obzir stanje u kome se pozivaju metodi. Tehnike testiranja koje to ne uzimaju u obzir (npr. tradicionalna pokrivenost kontrole toka) nisu efikasne u otkrivanju grešaka koje zavise od stanja objekta.

- **Enkapsulacija**. Efekti izvršavanja objektno orijentisanog koda mogu da uključuju izlaze, izmene stanja objekta, ili oba. Verifikacija rezultata može zahtevati pristup privatnim informacijama.
-
- **Nasleđivanje**. Testiranje da uzme u obzir efekte novih i redefinisanih metodama na ponašanje nasleđenih metoda, i napravi razliku između metoda koji zahtevaju nove test primere, nasleđenih metoda koji mogu da se testiraju ponovnim izvršavanjem postojećih test primere, kao i metoda koji ne moraju da budu ponovo testirani.

- **Polimorfizam i Dinamičko vezivanje**. Poziv jednog metoda može biti dinamički vezan za različite metode u zavisnosti od stanja objekta. Testovi moraju da izvrše različita vezivanja da otkriju propuste koje zavise od određenog vezivanja ili interakcija između vezivanja za različite pozive.

Karakteristike OO softvera od uticaja na testiranje

- **Apstraktne Klase**. Apstraktne klase ne mogu biti direktno instancirane i testirane, ali onr mogu biti važni elementi interfejsa u bibliotekama i komponentama. Neophodno je testirati ih bez punog znanja o tome kako one mogu biti instancirane.

- **Obrada izuzetaka**. Koristi se intenzivno u modernom objektno orijentisanom programiranju. Tekstualno rastojanje između tačke gde se baca izuzetak i tačke gde se hvata, i dinamičko vezivanje, čini bitnim da se eksplicitno testiraju izuzeci.

- **Konkurentnost**. Moderni objekat-orijentisani jezici i biblioteke podstiču, a ponekad čak i zahtevaju više niti kontrole (npr. java AWT i Swing korisnički interfejs). Konkurentnost uvodi nove vrste mogućih defekata, kao što su deadlock i racing i čini ponašanje sistema zavisnim od sistemskog planera koje nije pod kontrolom testera.

Primenljivost klasičnih tehnika jediničnog testiranja

- Metode crne kutije:
  - Ove tehnike su jednako upotrebljive kao i kod konvencionalnih sistema (uz vođenje računa da objekat klase sa sobom nosi stanje)

- Metode bele kutije:
  - Uzimajući u obzir prethodno rečeno, nisu sve primenljive ili efikasne tj. zahtevaju posebna razmatranja koja ćemo opisivati u nastavku

OO definicijie jediničnog i integracionog testiranja

- Proceduralni softver
  - jedinično = jedna komponenta, modul, funkcija ili procedura
- Objektno-orijentisani softver
  - jedinično = klasa ili (manja) grupa čvrsto povezanih klasa (npr. bazna i izvedena klasa)
  - jedinično testing = testiranje unutar klase (**intra-class testing**)
  - integraciono testiranje = **inter-class testing** (međuklasno testiranje)

Unutar klasno testiranje na bazi konačnog automata (primer jediničnog funkcionalnog testiranja)

- Osnovna ideja:
  Stanje objekta se menja operacijama nad njime.
  Metode se mogu modelovati kao prelazi među stanjima.
  Test slučajevi su nizovi poziva metoda koji predstavljaju određenu putanju u modelu stanja.

- Model stanja se može izvesti iz specifikacije, koda, ili primenom oba načina

Primer specifikacije: konfigurisanje računara za narudžbu

Slot: predstavlja mesto za komponentu u modelu (tj. konfiguraciji) računara.

.... Slotovi mogu biti popunjeni (bound) ili slobodni (unbound). Popunjenim slotovima dodeljena je kompatibilna komponenta, slobodni su prazni. Klasa slot pruža sledeće servise:

- **Install**: slot može biti instaliran na model kao obavezan (*required*) ili neobavezan (*optional*).

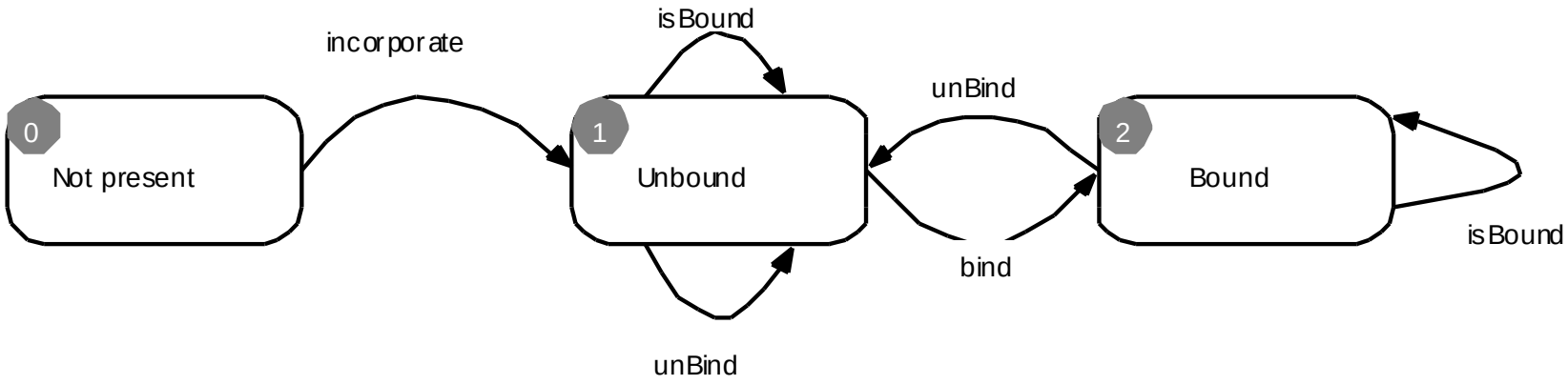
  ...
- **Bind**: slot može biti popunjen kompatibilnom komponentom.

  ...
- **Unbind**: popunjen slot može biti oslobođen uklanajnjem komponente.
- **IsBound**: vraća trenutnu vrednost slota, ako je popunjen; inače vraća specijalnu vrednost prazan (*empty*).

Identikacija stanja i prelaza

- Iz neformalne specifikacije možemo identifikovati sledeća stanja:
  - Not_installed
  - Unbound
  - Bound
- i četiri prelaza
  - install: iz stanja Not_installed u stanje Unbound
  - bind: iz Unbound u Bound
  - unbind: ...u Unbound
  - isBound: ne menja stanje

Konstrukcija konačnog automata i test primera



Na primer, za pokrivanje svih prelaza uzimamo test primere:

- TC-1: incorporate, isBound, bind, isBound
- TC-2: incorporate, unBind, bind, unBind, isBound

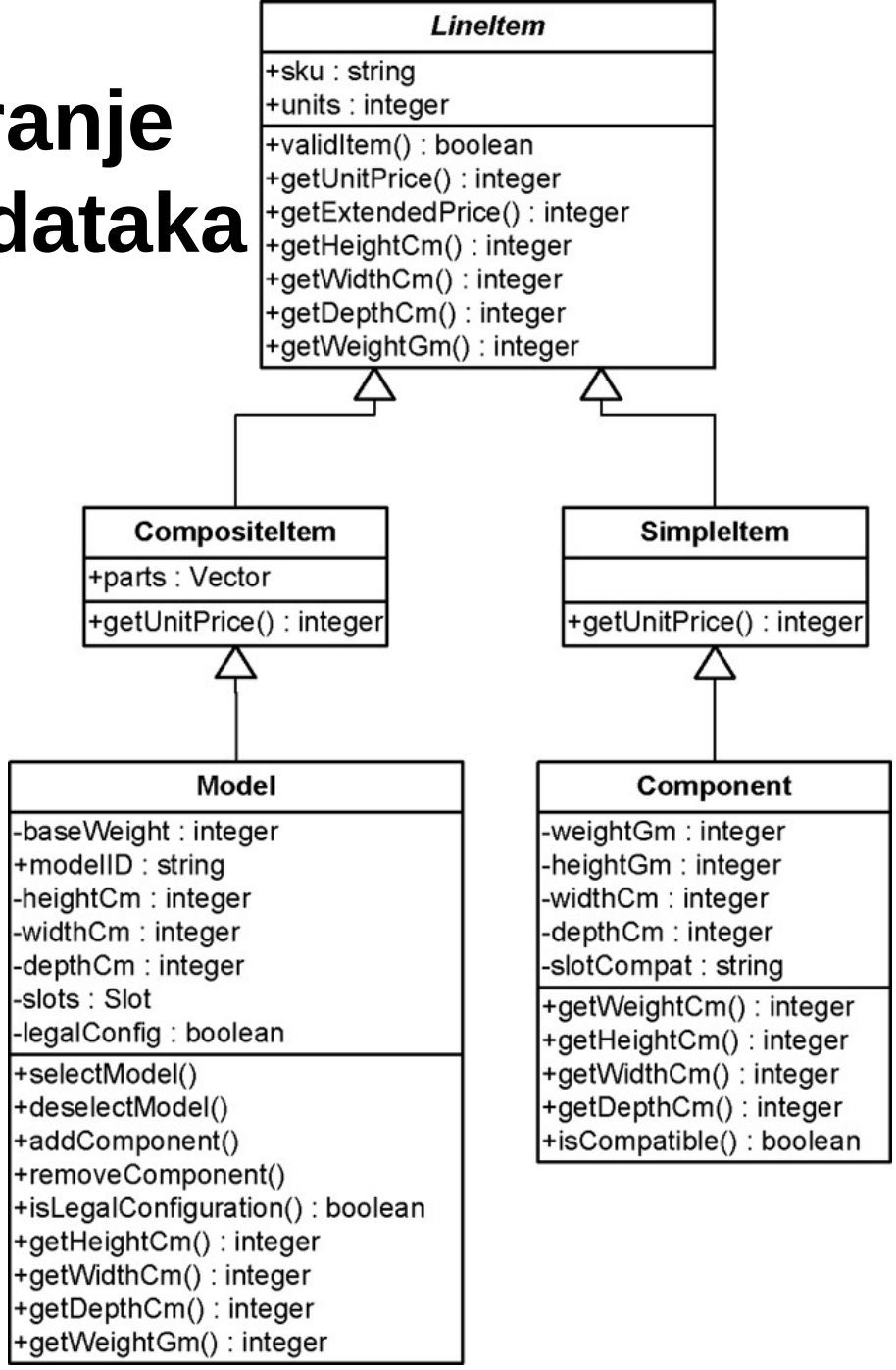
Unutarklasno testiranje tehnikom toka podataka (primer jediničnog strukturnog testiranja)

- Izvršavanja različitih sekvenci metoda (du lanci)
  - Od postavljanja ili modifikovanja vrednosti polja
  - Do upotrebe te vrednosti

- Potrebno je voditi evidenciju o toku kontrole koji obuhvata više od jednog poziva metoda ...

Unutarklasno testiranje na osnovu toka podataka

- Primer: Stavka narudžbe modela (konfiguracije) računara modela koji se sastoji od određenih komponenata. Hoćemo da testiramo klasu Model

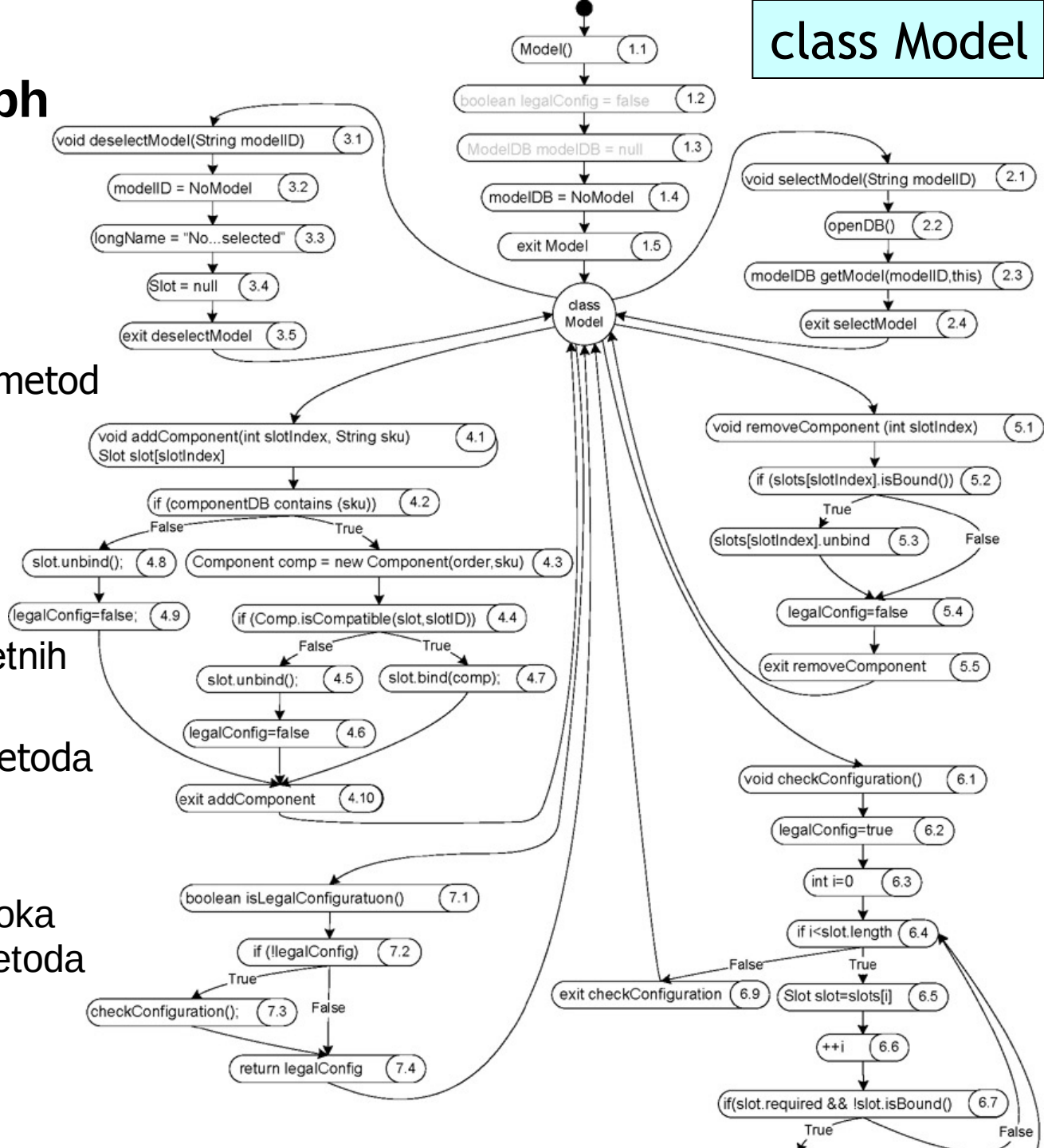


Implementacija klase Model

```
1  public class Model extends Orders.CompositeItem {
2      public String modelID; // Database key for slots
3      private int baseWeight; // Weight excluding optional components
4      private int heightCm, widthCm, depthCm; // Dimensions if boxed
5      private Slot[] slots; // Component slots
6
7      private boolean legalConfig = false; // result of isLegalConf
8      private static final String NoModel = "NO MODEL SELECTED";
12
13     /** Constructor, which should be followed by selectModel */
14     public Model(Orders.Order order) {
15         super( order);
16         modelID = NoModel;
17     }
61 ...
62     /** Bind a component to a slot.
63       * @param slotIndex Which slot (integer index)?
64       * @param sku Key to component database.
65       * Choices should be constrained by web interface, so we don't
66       * need to be graceful in handling bogus parameters.
67       */
68     public void addComponent(int slotIndex, String sku) {
69         Slot slot =slots[slotIndex];
70         if (componentDB.contains(sku)) {
71             Component comp = new Component(order, sku);
72             if (comp.isCompatible(slot.slotID)) {
73                 slot.bind(comp);
74                 // Note this cannot have made the
75                 // configuration illegal.
76             } else {
77                 slot.unbind();
78                 legalConfig = false;
79             }
80         } else {
81             slot.unbind();
82             legalConfig = false;
83         }
84     }
86
87     /** Unbind a component from a slot. */
88     public void removeComponent(int slotIndex) {
89         // assert slotIndex in 0..slots.length
90         if (slots[slotIndex].isBound()) {
91                 slots[slotIndex].unbind();
92             }
93          legalConfig = false; 94 }
94 ...
100    /** Is the current binding of components to slots a legal
101    * configuration? Memo-ize the result for repeated calls */
102    public boolean isLegalConfiguration() {
103        if (! legalConfig) {
104            checkConfiguration();
105        }
106        return legalConfig;
107    }
108
109    /** Are all required slots filled with compatible components?
110      * It is impossible to assign an incompatible component,
111      * so just to check that every required slot is filled. */
112    private void checkConfiguration() {
113        legalConfig = true;
114        for (int i=0; i < slots.length; ++i) {
115            Slot slot = slots[i];
116            if (slot.required && ! slot.isBound()) {
117                legalConfig = false;
118            }
119        }
120    }
241 ...
242 }
```

The intraclass control flow graph

class Model



Graf kontrole toka za svaki metod

+

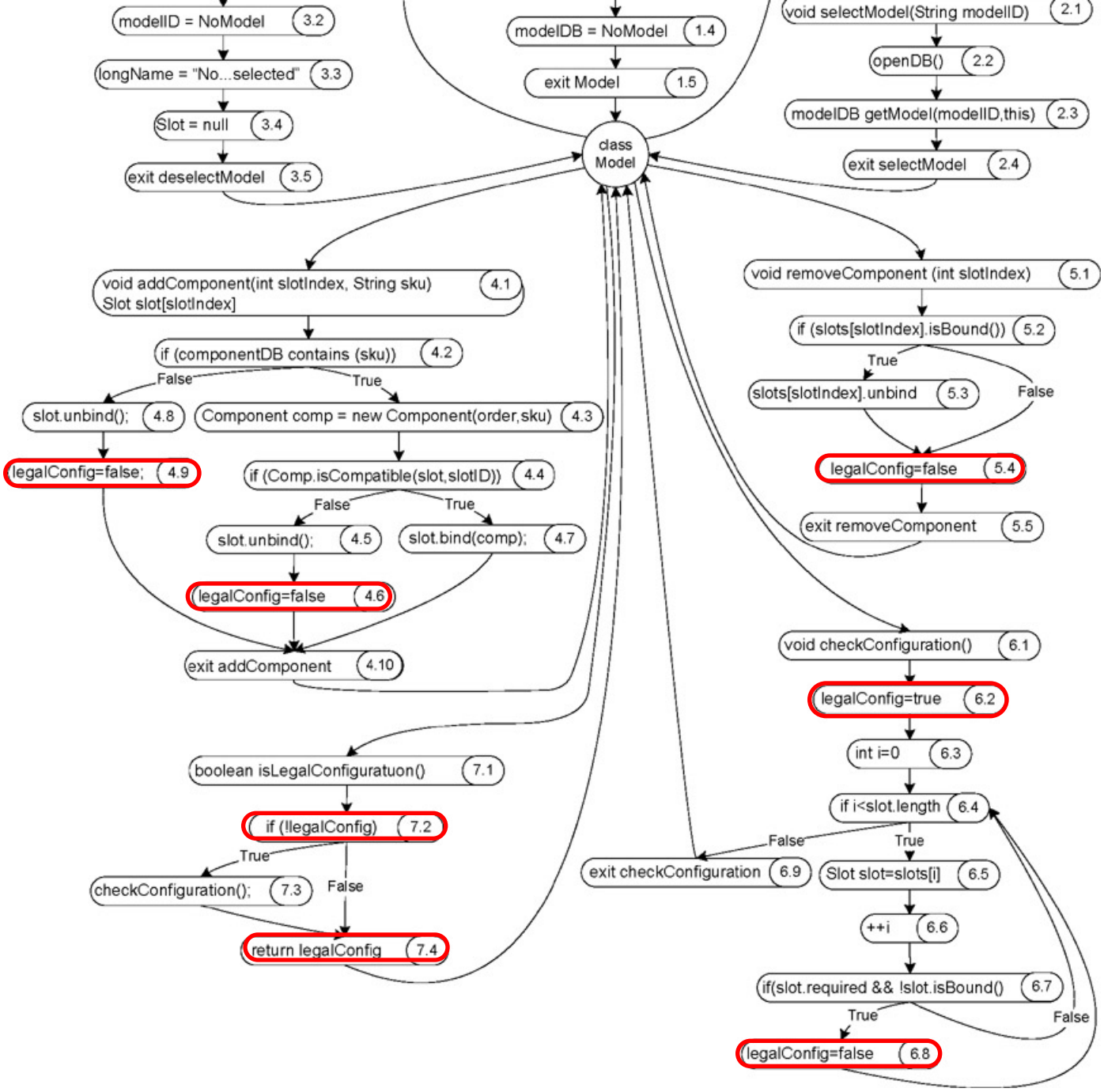
Čvor koji reprezentuje klasu

+

grane

- Od čvora *klase* do početnih čvorova metoda
- Od završnih čvorova metoda do čvora *klase*

=> Reprezentuje kontrolu toka kroz *sekvence* poziva metoda



Zumirano, analiziramo dodele i upotrebe promenljive legalConfig

Definition-Use (DU) parovi

Za promenljivu instance **legalConfig** DU parovi se definišu parovima metoda (u prvom postoji dodela vrednosti, u drugom korišćenje bez prethodne nove dodele).

<model (1.2), isLegalConfiguration (7.2)>
<addComponent (4.6), isLegalConfiguration (7.2)>
<removeComponent (5.4), isLegalConfiguration (7.2)>
<checkConfiguration (6.2), isLegalConfiguration (7.2)>
<checkConfiguration (6.3), isLegalConfiguration (7.2)>
<addComponent (4.9), isLegalConfiguration (7.2)>

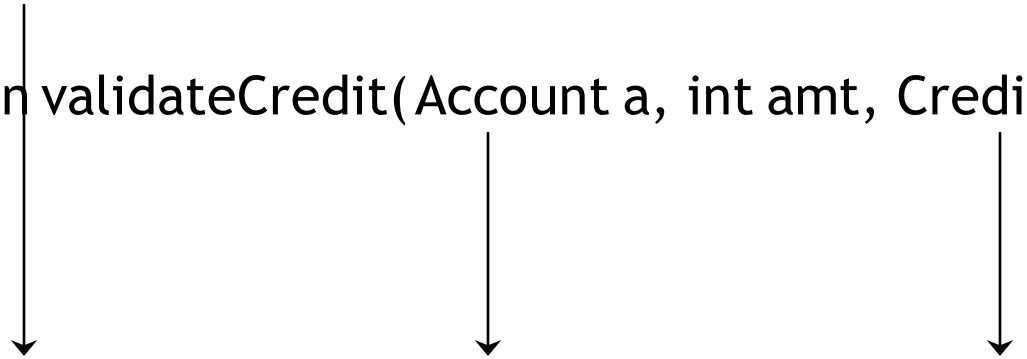
Svaki par treba pokriti bar jednim test primerom. Test primeri su u opštem slučaju kompleksne sekvence poziva metoda.

Napomena: neke parove je u opštem slučaju nemoguće pokriti testom.

Polimorfizam i dinamičko vezivanje

“Izolovani” pozivi: problem kombinatorne eksplozije

```
abstract class Credit {
...
  abstract boolean validateCredit(Account a, int amt, CreditCard c);
...
}
```



EduCredit
BizCredit
IndividualCredit

USAccount
UKAccount
EUAccount
JPAccount
OtherAccount

VISACard
AmExpCard
StoreCard

The kombinatorni problem: 3 x 5 x 3 = 45 mogućih kombinacija dinamičkih vezivanja (samo za ovaj jedan metod!)

Kombinatorno testiranje

Identifikovati skup kombinacija koji pokriva sve parove dinamičkih vezivanja

Account	Credit	creditCard
USAccount	EduCredit	VISACard
USAccount	BizCredit	AmExpCard
USAccount	individualCredit	ChipmunkCard
UKAccount	EduCredit	AmExpCard
UKAccount	BizCredit	VISACard
UKAccount	individualCredit	ChipmunkCard
EUAccount	EduCredit	ChipmunkCard
EUAccount	BizCredit	AmExpCard
EUAccount	individualCredit	VISACard
JPAccount	EduCredit	VISACard
JPAccount	BizCredit	ChipmunkCard
JPAccount	individualCredit	AmExpCard
OtherAccount	EduCredit	ChipmunkCard
OtherAccount	BizCredit	VISACard
OtherAccount	individualCredit	AmExpCard

Nepoželjni efekti kombinovanih poziva (ovakve stvari treba detektovati testiranjem)

```
public abstract class Account { ...
    public int getYTDPurchased() {
     if (ytdPurchasedValid) { return ytdPurchased; }
     int totalPurchased = 0;
     for (Enumeration e = subsidiaries.elements() ; e.hasMoreElements(); )
          {   Account subsidiary = (Account) e.nextElement();
            totalPurchased += subsidiary.getYTDPurchased();   }
     for (Enumeration e = customers.elements(); e.hasMoreElements(); )
        {   Customer aCust = (Customer) e.nextElement();
            totalPurchased += aCust.getYearlyPurchase();   }
     ytdPurchased = totalPurchased;
     ytdPurchasedValid = true;
     return totalPurchased;
    }  ...  }
```

Ukupna godišnja suma nabavke za račun određuje se metodom getYTDPurchased, koji sumira nabavke od strane svih kupaca preko računa i iz svih filijala. Iznosi se uvek evidentiraju u lokalnoj valuti na računu, ali getYTDPurchased sumira nabavke iz filijala, čak i kada one koriste različite valute (na primer, kada neke su vezane za potklasu USAccount a druge za EUAccount). Tehnike unutarklasnog i međuklasnog testiranja predstavljene u prethodnim slajdovima možda neće uspeti da otkriju ovu vrstu defekta. Problem se može rešiti tako što će se izabrati test primeri koji pokrivaju kombinacije polimorfnih poziva za različite instance objekata.

Pristup putem toka podataka

```
public abstract class Account {
...
    public int getYTDPurchased() {
                if (ytdPurchasedValid) return ytdPurchased; }
                int totalPurchased = 0;
                for (Enumeration e = subsidiaries.elements() ; e.hasMoreElements(); )
                    {
                                Account subsidiary = (Account) e.nextElement();
                                totalPurchased += subsidiary.getYTDPurchased
                    }
                for (Enumeration e = customers.elements(); e.hasMoreElements(); )
                    {
                                Customer aCust = (Customer) e.nextElement();
                                totalPurchased += aCust.getYearlyPurchas
                    }
                ytdPurchased = totalPurchased;
                ytdPurchasedValid = true;
                return totalPurchased;
    }
...
}
```

totalPurchased defined

korak 1: identifikacija polimofnih poziva, skupa vezujućih metoda, definicija i upotreba

totalPurchased used and defined

totalPurchased used and defined

totalPurchased used

totalPurchased used

Da bismo identifikovali sekvencijalne kombinacije vezivanja, prvo moramo da identifikujemo individualne polimorfne pozive i skupove vezivanja, a zatim izaberemo moguće sekvence.

Def-Use (dataflow) testiranje polimorfnih poziva

- Napraviti test primer za svaki mogući polimorfni <def,use> par
  - Svako vezivanje mora zasebno da se razmatra
  - Jedan par dodele-upotrebe postaje $n \times m$ parova ako se tačka dodele može vezati na n načina a tačka upotrebe na m načina.
  - Kombinatorna selekcija parova može pomoći da se redukuje broj test primera
  - Slabija ali ipak korisna alternativa je da se svako vezivanje posmatra nezavisno do drugog, što rezultuje u m ili n parova (koji god da je veći) umesto njihovog proizvoda.

Obrada izuzetaka

```
void addCustomer(Customer theCust) {
   customers.add(theCust);
     }
     public static Account
   newAccount(...)
   throws InvalidRegionException
     {
   Account thisAccount = null;
   String regionAbbrev = Regions.regionOfCountry(
                      mailAddress.getCountry());
   if (regionAbbrev == Regions.US) {
       thisAccount = new USAccount();
   } else if (regionAbbrev == Regions.UK) {
       ....
   } else if (regionAbbrev == Regions.Invalid) {
       throw new
   InvalidRegionException(mailAddress.getCountry());
   }
...
     }
```

Izuzeci kreiraju implicitnu kontrolu toka i mogu biti obrađivani od strane različitih hendlera

Testiranje obrade izuzetaka

- Nepraktično da se izuzeci tretiraju na isti način kao normalni tok kontrole
  - Previše putanja: svako indeksiranje elementa niza, svaka dodela memorije, svaka konverzija tipa, ...
  - Pomnožiti to sa brojem hendlera koji se mogu pojaviti neposredno iznad u steku poziva.
  - Mnogi od njih su u stvarnosti nemogući
- Zbog toga posebno testiramo korisničke izuzetke
  - a ignorišemo izuzetke usled programskih grešaka (tj. drugim testovima testiramo korektnost programa)
- Šta testirati: Svaki hendler izuzetka i svako eksplicitno bacanje (ili ponovno bacanje) izuzetka

Testiranje obrade izuzetaka

- Lokalni hendleri izuzetaka
  - Testirati hendler izuzetaka (razmotriti podskup tačaka bacanja vezanih za hendler)
- Ne-lokalni hendleri izuzetaka
  - Teško je ustanoviti sva uparivanja <bacanje, obrada>
  - Treba se pridržavati sledećeg projektnog pravila (i testirati ovo):
    ako metod prosleđuje izuzetak, poziv metoda ne treba da ima *nikakav drugi efekat*

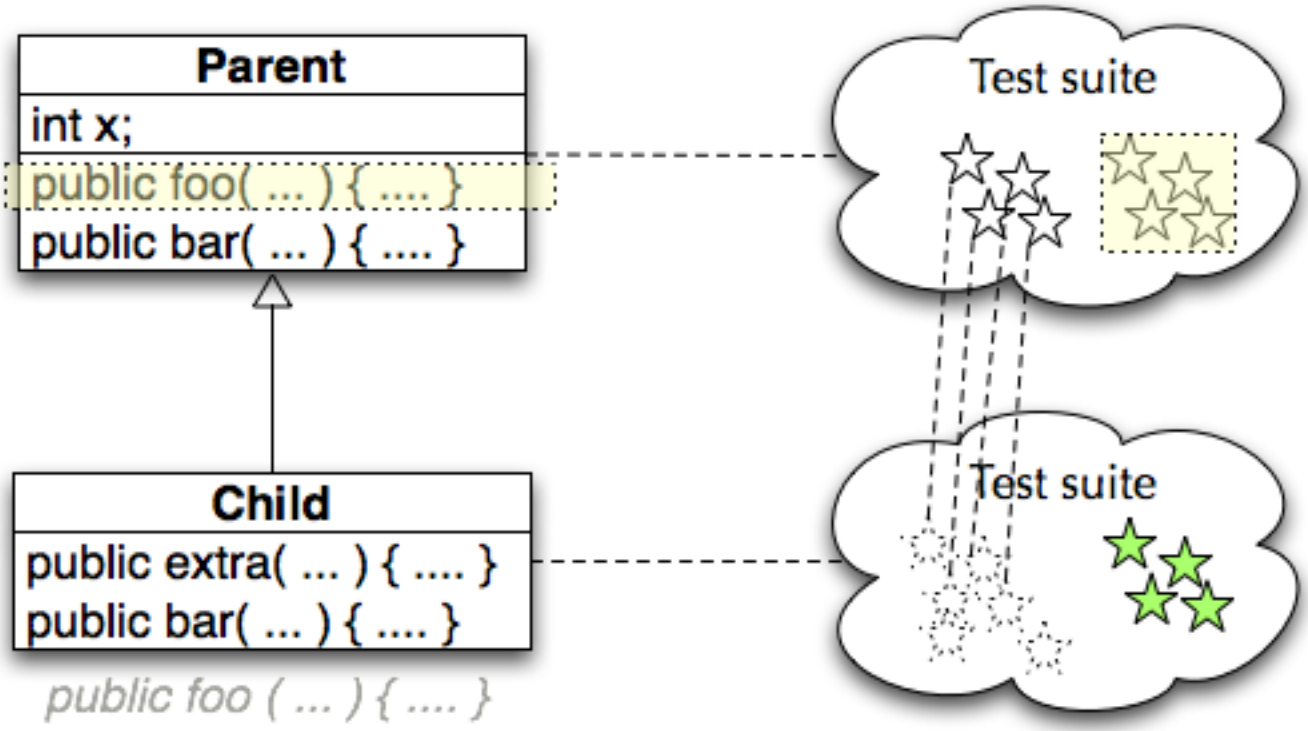
Nasleđivanje

- Kada testiramo podklasu...
  - Želimo da testiramo samo delove koji nisu već temeljno istestirani u roditeljskoj klasi
    - Na primer, nema potrebe testirati metode hashCode i getClass nasleđene iz klase Object u Javi
  - Ali treba da ponovo testiramo svaki metod čije ponašanje je moglo da bude promenjeno (setiti se uvodnog primera lekcije).

Ponovno korišćenje testova za podklase

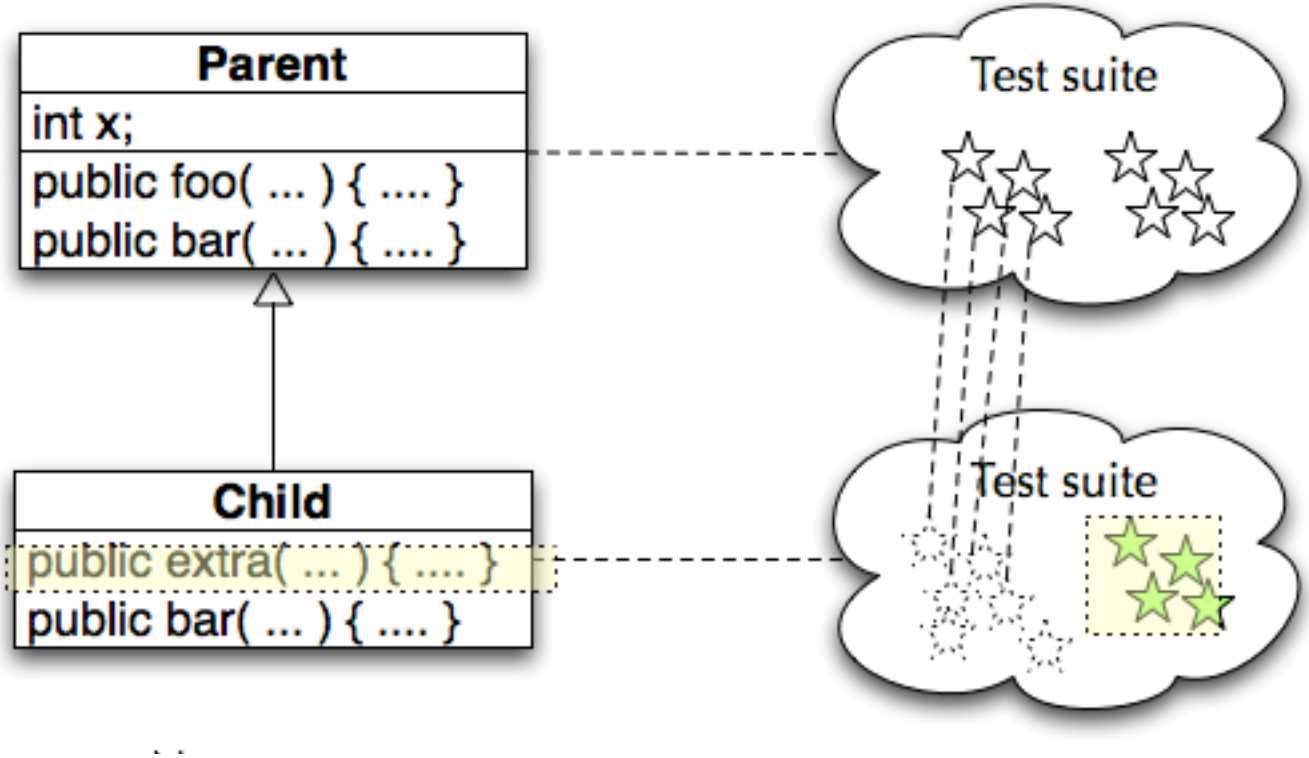
- Voditi evideciju o skupovima testova i izvršavanjima tih testova (tzv. istorijat testiranja)
  - Odrediti koje nove testove je potrebno razviti
  - Odrediti koje stare testove treba ponovo izvršiti
- Novo i promenjeno ponašanje uslovljava da:
  - treba testirati nove metode
  - redefinisani metodi moraju se testirati, ali se delimično mogu iskoristiti skupovi testova nadklase
  - drugi nasleđeni metodi ne moraju se ponovo testirati

Nasleđeni, nepromenjeni



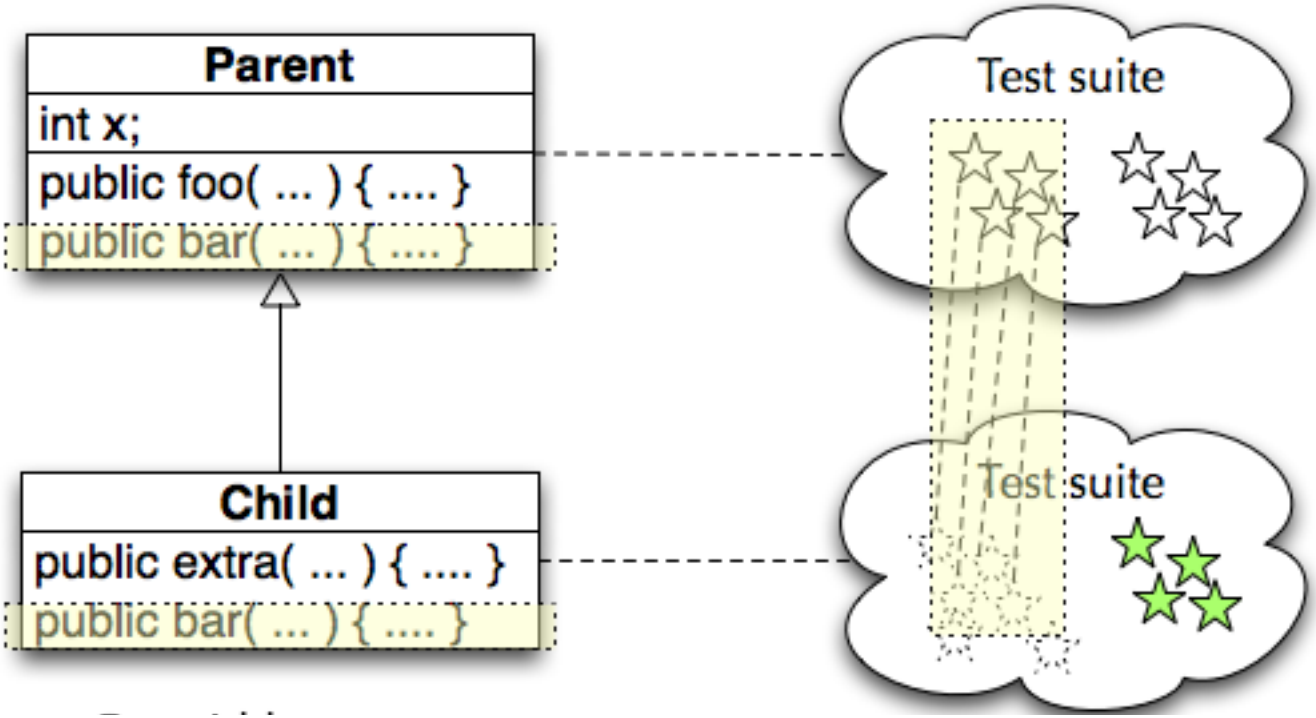
Nije potrebno ponovo testirati

Novo dopisani metodi



Projektovati i izvršiti nove test primere

Redefinisani metodi

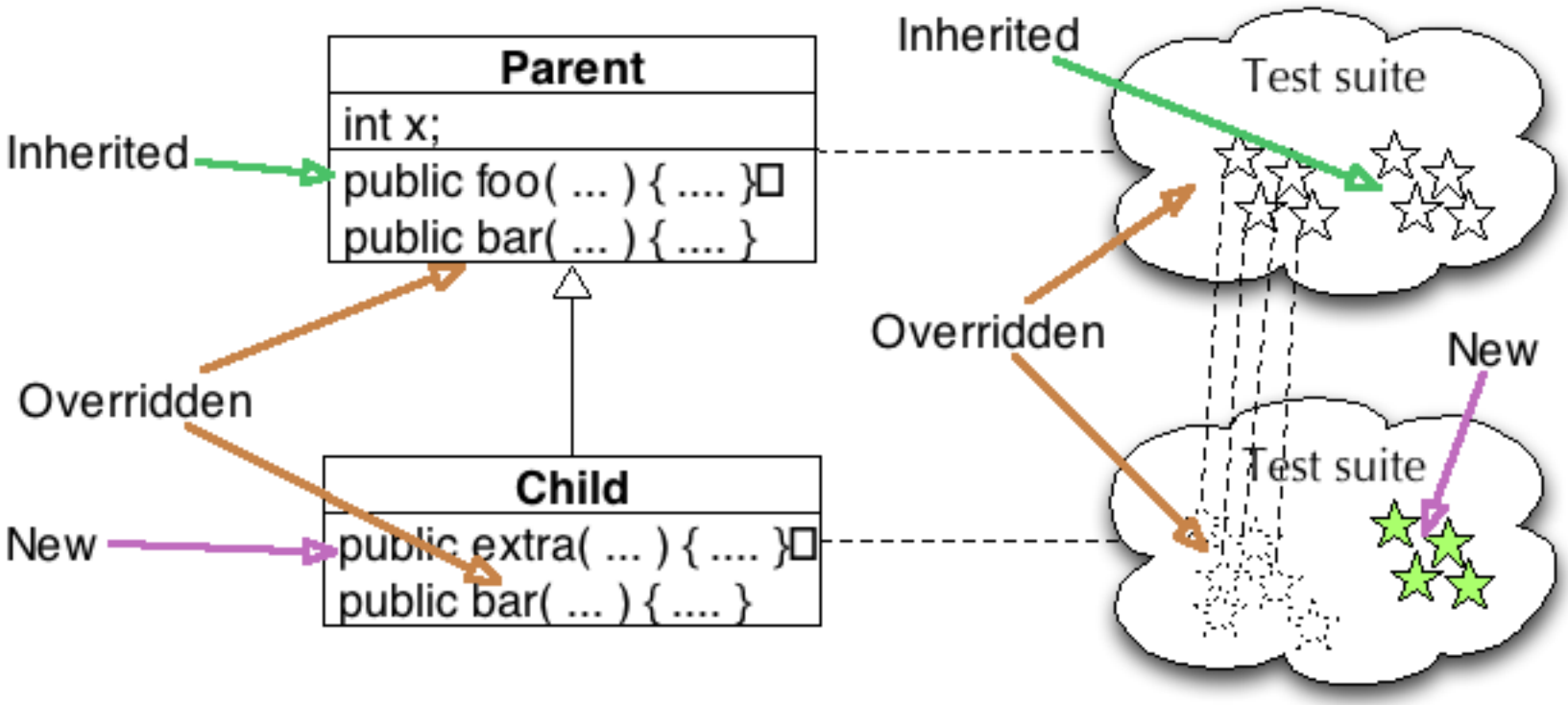


Ponovo izvršiti roditeljske test primere, dodati nove test primere po potrebi

Ostalo

- Abstraktne klase i metodi
  - Preporuka je da se projektuju test primeri čim nastane apstraktan metod (čak i kada još uvek ne mogu da se izvršavaju)
- Promena ponašanja
  - Smatrati metod "redefinisanim" ako drugi novi ili redefinisani metod izmeni ponašanje prvog

Istorijat testiranja - rezime

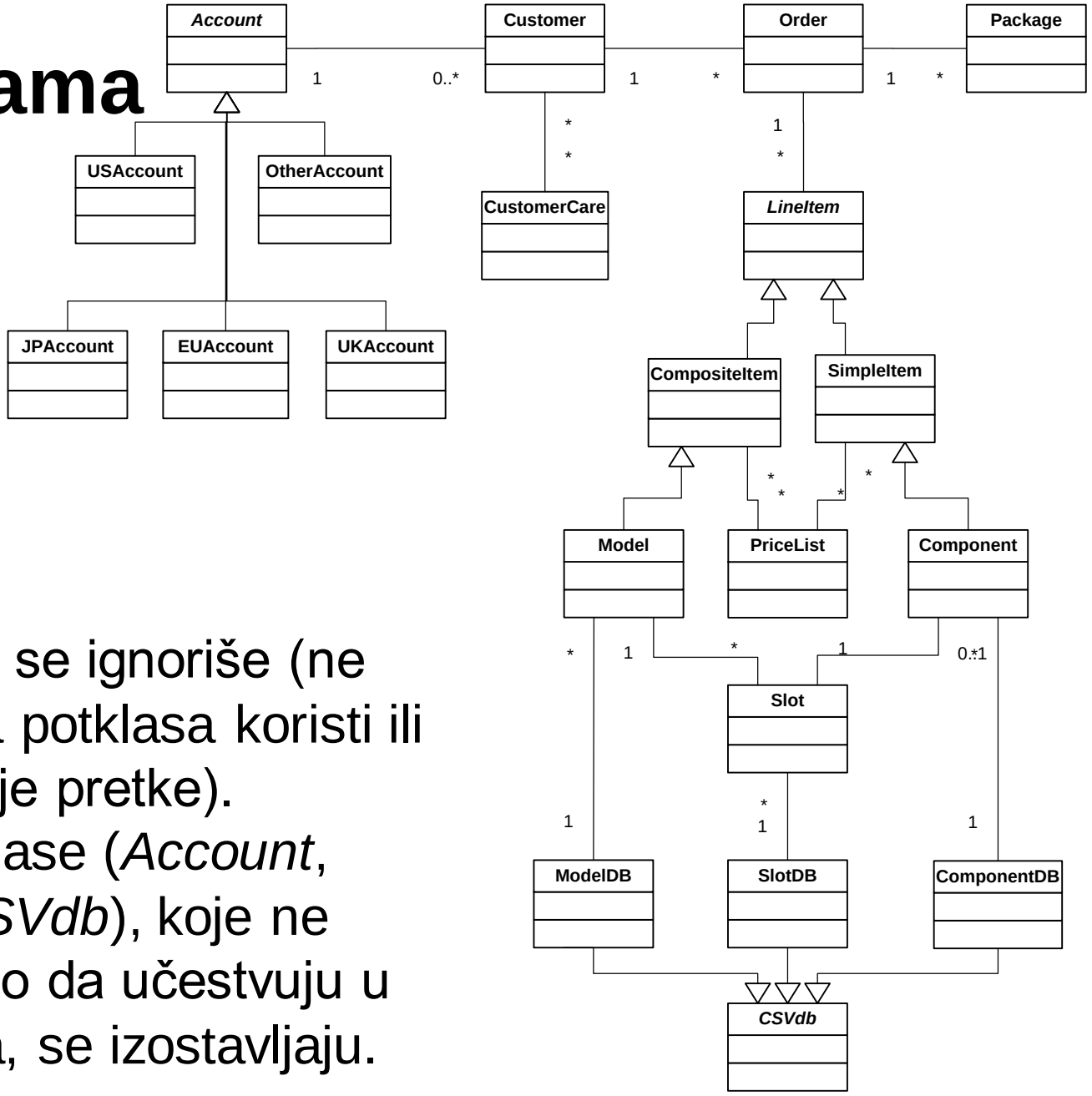


OO integraciono testiranje

Međuklasno testiranje

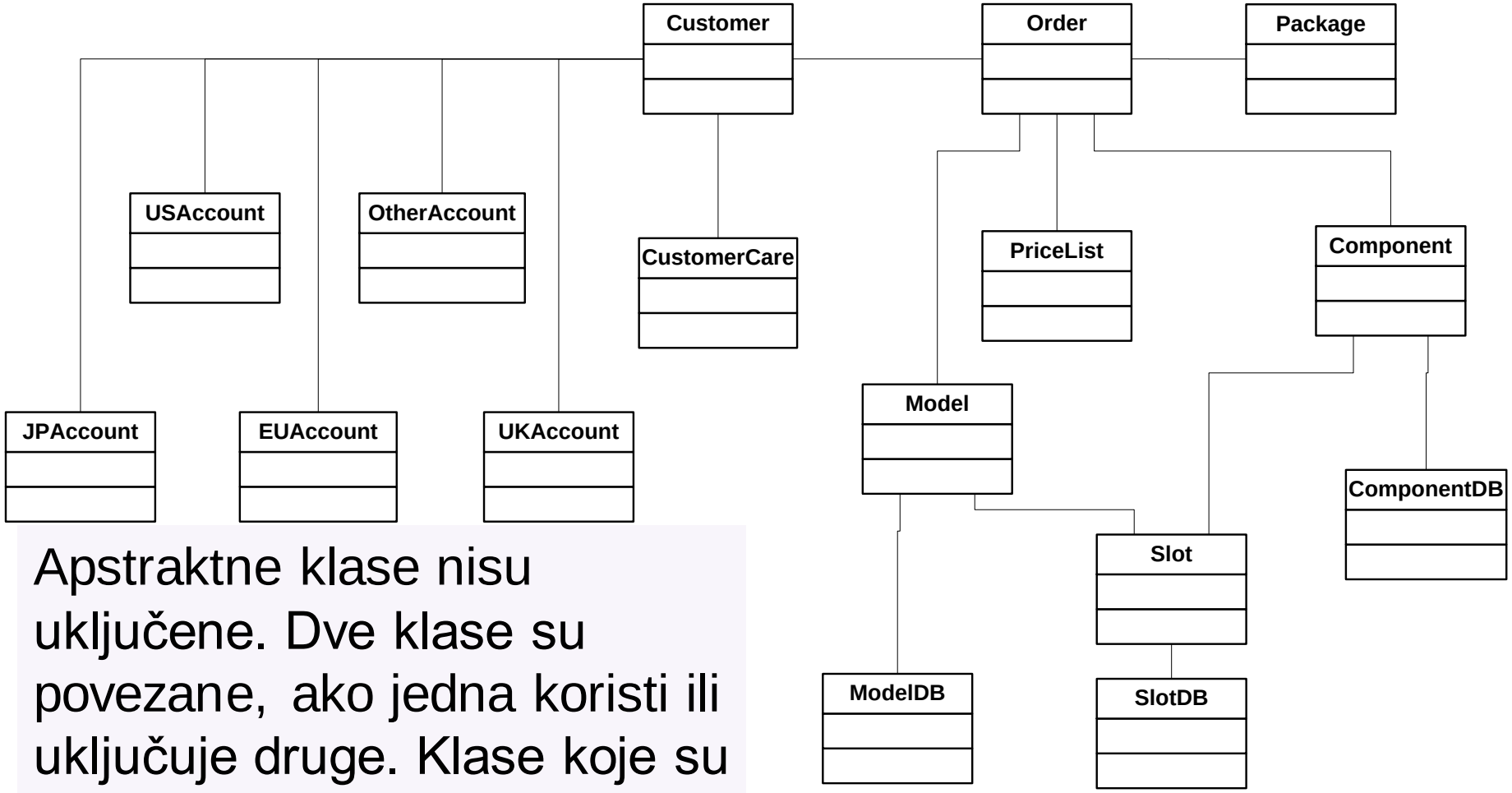
- Predstavlja *integraciono testiranje* za objektno-orijentisani softver
  - Fokusira se interakcije između klasa
- Itegracija odozdo-na gore prema relaciji "use/include" (koristi/uključuje)
  - A use B ili A include B: Prevesti i testirati prvo B, onda dodati i A
- Use/include hijerarhija
    - Klasa A poziva metode klase B (*use*)
    - Objekti klase A imaju reference na objekte klase B (*include*)
      - Ali samo ako referenca znači "je deo od"

Polazi se od dijagrama klasa...



Nasleđivanje se ignoriše (ne smatramo da potklasa koristi ili uključuje svoje pretke). Apstraktne klase (*Account*, *LineItem* i *CSVdb*), koje ne mogu direktno da učestvuju u interakcijama, se izostavljaju.

Use/include hijerarhija za dati dijagram klasa



Apstraktne klase nisu uključene. Dve klase su povezane, ako jedna koristi ili uključuje druge. Klase koje su više u dijagramu uključuju ili koriste klase koje su niže u dijagramu.

Use/include hijerarhija

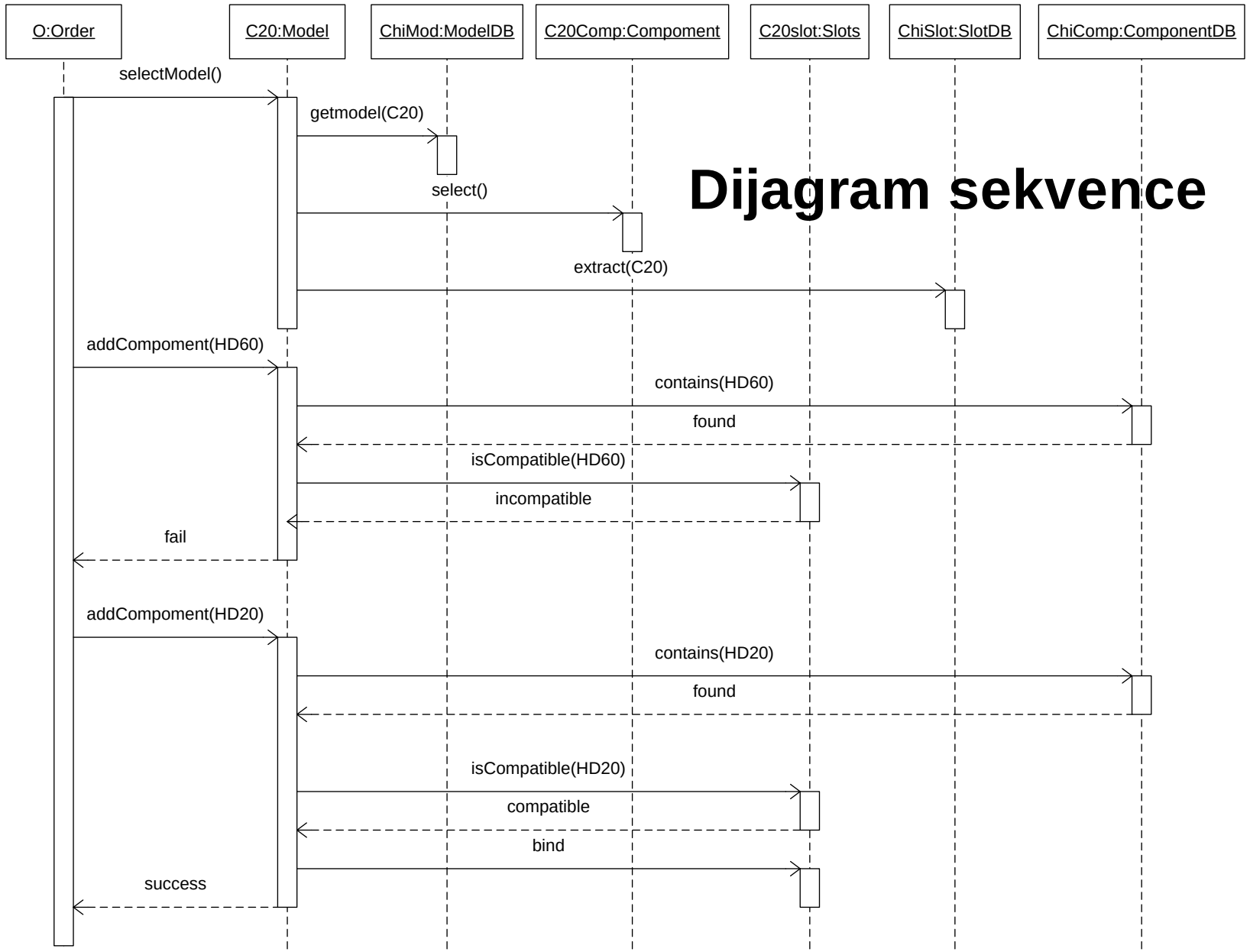
- Kada use/include relacija među klasama pravi ciklus u grafu (npr. "poziv na gore" do pretka u grafu), petlja može biti uklonjena zamenom pretka stabom. Tako se uvek može postići aciklički graf.

- Međuklasne strategije testiranja obično primenjuju strategiju odozdo na gore, počevši od klasa koje ne zavise od drugih.

- Na primer, možemo da počnemo integrišući klasu SlotDB sa klasom Slot i klasu Component sa klasom ComponentDB, a zatim nastavimo postepeno integrisanjem klasa ModelDB i Model, do klase Order.

Interakcije u međuklasnim testovima

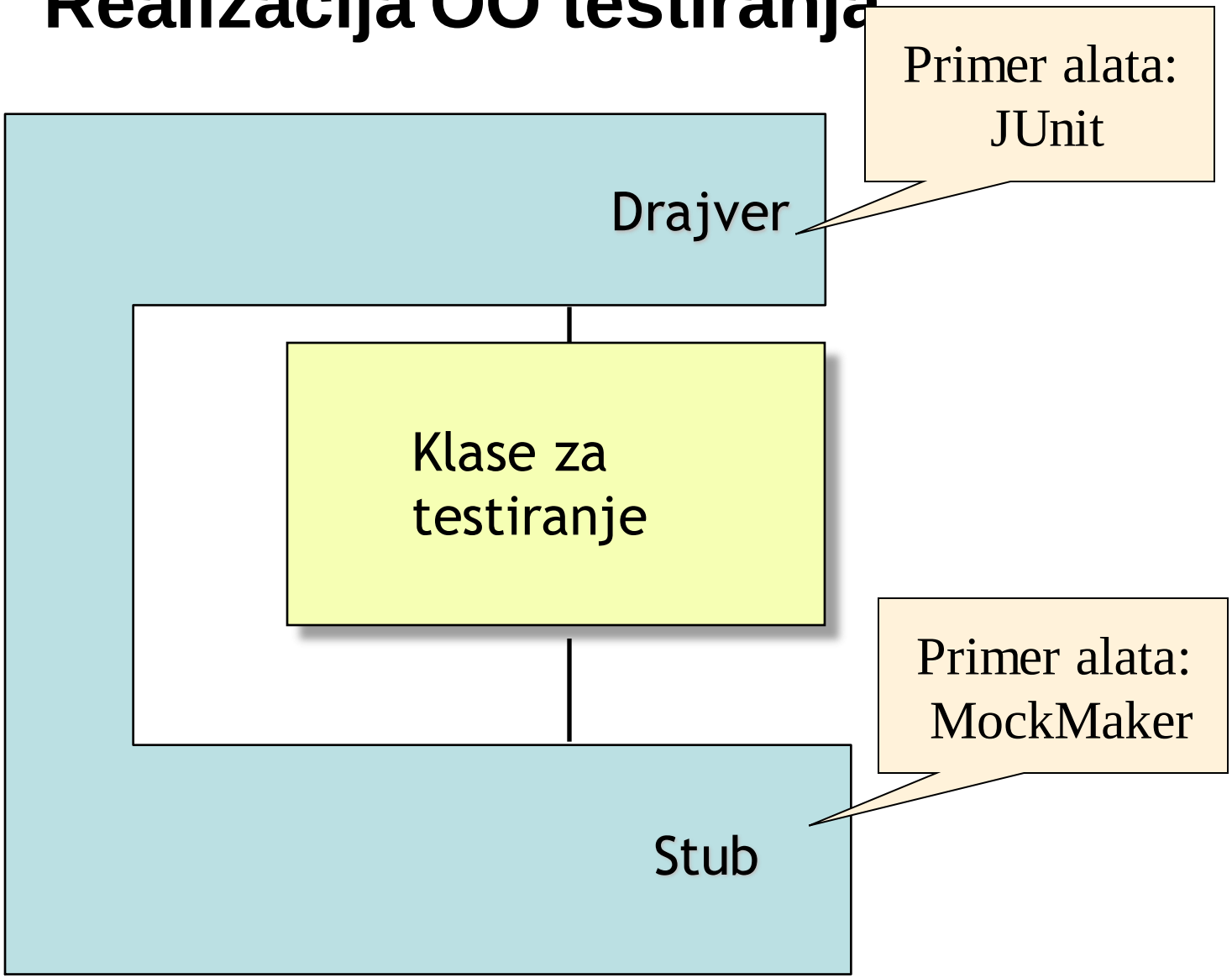
- Dok se penjemo duž use/include relacije,
- Razmotriti sve kombinacije interakcija
  - Primer: test primer za klasu *Order* uključuje poziv metoda klase *Model* i pozvani metod poziva metod klase *Slot, treba pokriti sva moguća stanja različitih klasa*
  - problem: kombinatorna eksplozija testova
  - rešenje: izabrati podskup interakcija:
    - Proizvoljna (slučajna) selekcija
    - plus svi važni scenariji identifikovani u fazi analize i projektovanja: dijagrami sekvence + dijagrami kolaboracije



Dijagram sekvence

(Delimični) dijagram sekvence koji definiše interakcije između objekata klasa *Order, Model, ModelDB, Component, ComponentDB, Slot* i *SlotDB* da bi se izabrao računar, dodala nelegalna komponenta, a zatim dodala legalna.

Realizacija OO testiranja



Predikcija rezultata testa

- Prediktor testa mora biti u stanju da proveri ispravnost ponašanja objekta, kada se izvršava za zadati ulaz
- Izvršavanje proizvodi ***izlaze*** i dovodi objekat u ***novo stanje***
  - Mogu se koristiti tradicionalne pristupe za proveru korektnosti izlaza
  - Da bi se proverilo finalno stanje mora mu biti omogućen pristup

Pristup stanju objekta

- Invazivni pristupi
  - Upotreba jezičkih konstrukcija (C++ friend)
  - Dodavanje inspekcijskih metoda
  - *U oba slučaja narušava se enkapsulacija i mogu se dobiti neželjeni rezultati*
- Pristup putem ekvivalentnih scenarija:
  - generisati ekvivalentne i ne-ekvivalentne sekvuence poziva metoda
  - uporediti finalno stanje objekta posle ekvivalentne ili neekvivalentne sekvence

Pristup putem ekvivalentnih sekvenci

selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()
deselectModel()
selectModel(M2)
addComponent(S1,C1)
isLegalConfiguration()

EKVIVALENTNA
selectModel(M2)
addComponent(S1,C1)
isLegalConfiguration()

NEEKVIVALENTNA
selectModel(M2)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

Pretpostavka je da deselekcija modela za narudžbu treba automatski da ukloni sve komponente iz modela

Generisanje ekvivalentnih sekvenci

- ukloniti nepotrebne ("cirkularne") metode (vraćaju u prethodno stanje)

  selectModel(M1)

  addComponent(S1,C1)

  addComponent(S2,C2)

  isLegalConfiguration()

  deselectModel()

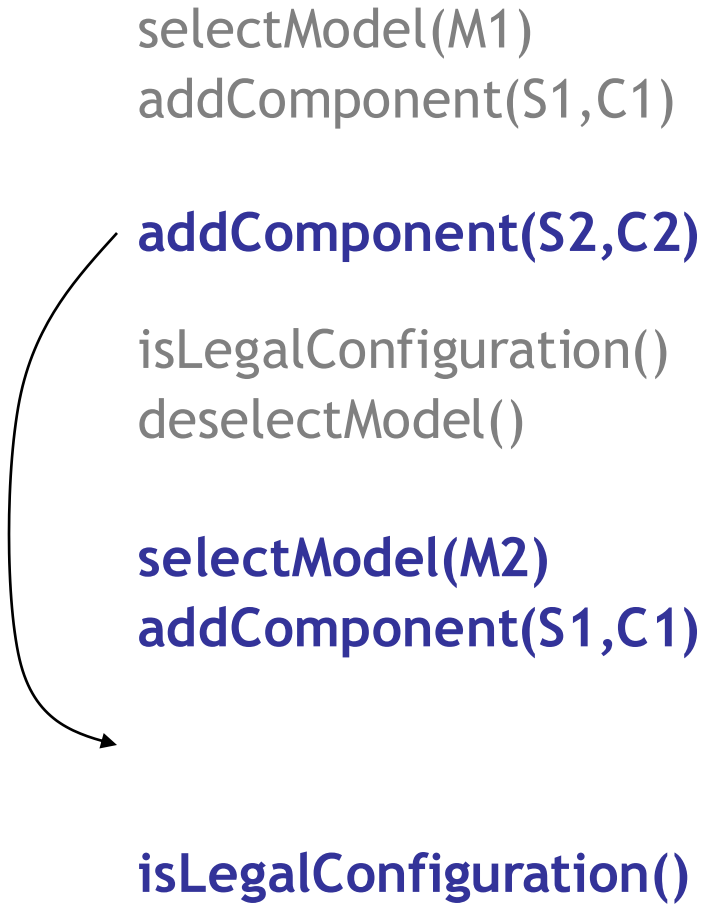
  selectModel(M2)

  addComponent(S1,C1)

  isLegalConfiguration()

Generisanje neekvivalentnih sekvenci

- Ukloniti i/ili izmešati osnovne akcije
- Pokušati sa generisanjem sekvenci koje podsećaju na realne defekte



Verifikacija ekvivalentnosti stanja

U teoriji: Dva stanja su ekvivalentna ako sve moguće sekvence metoda koje počinju iz tih stanja proizvode iste rezultate

U praksi:

- Dodati inspektore koji otkrivaju skriveno stanje i uporediti rezultate
  - ovim se narušava enkapsulacija
- Dodati klasi metod "compare" koji specijalizuje default metod *equal*
  - poseban metod je često neophodan jer ekvivalenta stanja ne podrazumevaju identičnost objekata do poslednjeg bita
  - ovo spada u projektovanje za mogućnost testiranja (design for testability)