# Implementacione strategije

# Implementacija asocijacija

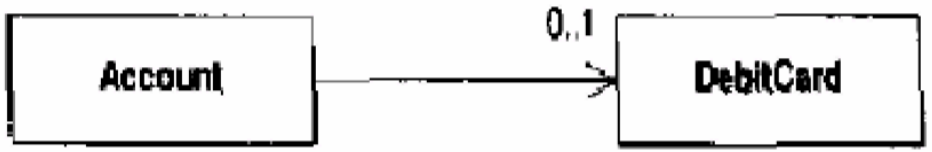
- Link vs. referenca
- Asocijacije
  - Jednosmjerna
  - Neusmjerena
- Deklaracija podatka člana u jednoj klasi koji će čuvati referencu na objekat druge klase
- Odgovarajući interfejs za manipulisanje referenom ili referencama

# Jednosmjerne asocijacije

- Multiplikativnost
  - 1
  - 0..1
  - *

# Multiplikativnost 0..1

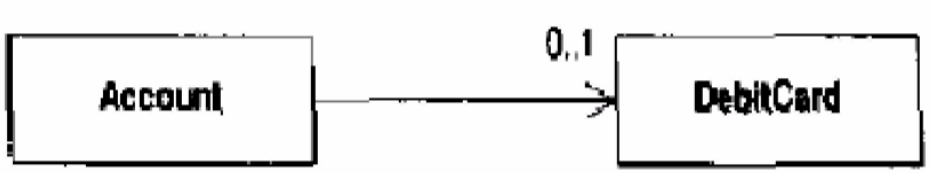
- Mutable association




```
public class Account
{
    public DebitCard getCard() {
        return theCard;
    }
    public void setCard(DebitCard card) {
        thecard = card;
    }
    public void removeCard(){
        theCard = null;
    }
    private DebitCard theCard;
}
```

# Multiplikativnost 0..1 (2)

- Immutable association




```
public class Account {
    public DebitCard getCard()    {
        return thecard ;
    }
    public void setCard(DebitCard card)   {
    if    (theCard    != null)     {
        //  throw ImmutableAssociationError
    }
    theCard = card ;
}
private DebitCard  theCard  ;
}
```

# Multiplikativnost 1


```
public class Account {
    public Account(Guarantor g) {
        if ( g == null ) {
            //throw NullLinkError
        }
        theGuarantor = g ;
    }
    public Guarantor getGuarantor(){
        return theGuarantor ;
    }
    private Guarantor theGuarantor ;
}
```

# Multiplikativnost *

- Objekat klase Manager mora da čuva kolekciju pokazivača na objekte klase Account i da implementira interfejs za upravljanje kolekcijom pokazivača
- Može li više

pokazivača na isti

Account objekat da

bude unijeto u niz

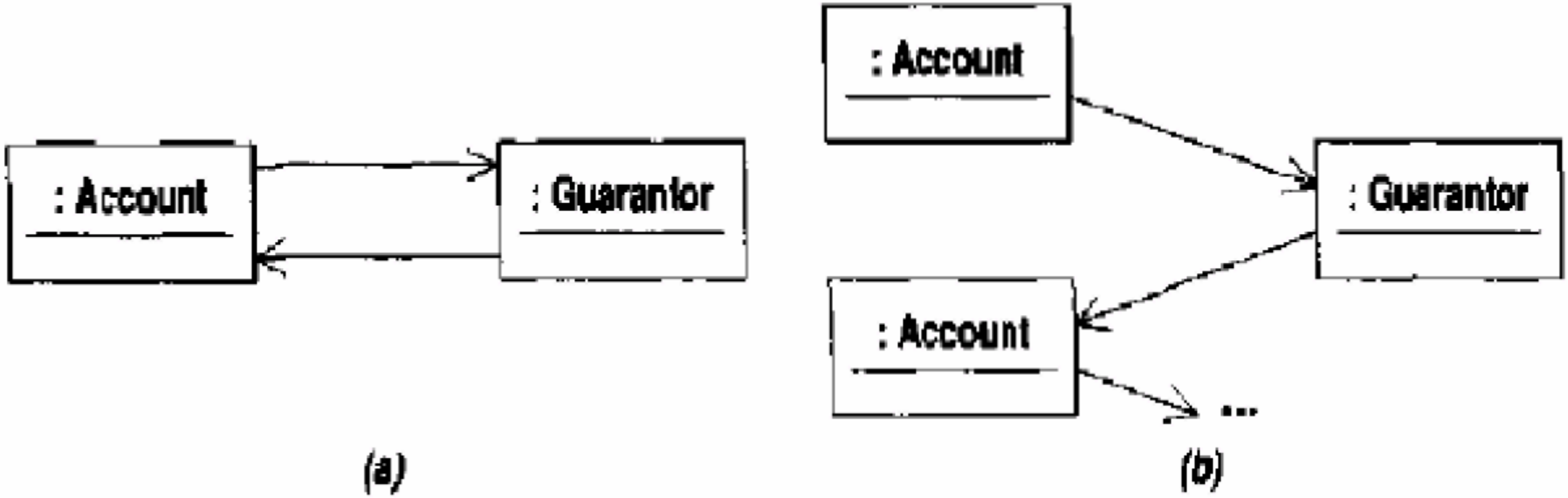
theAccounts?


```
public  class Manager {
    public void addAccount(Account acc)     {
        theAccounts.addElement (acc)    ;
    }
    public void removeAccount (Account acc) {
        theAccounts.removeElement(acc) ;

    }
    private Vector theAccounts ;
}
```

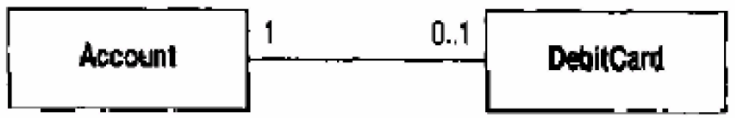
# Dvosmjerne asocijacije

- Potrebno je obezbijediti da par referenci za implementaciju budu konzistentni u run-time nalik referencijalnom integritetu



# Multiplikativnosti 1-1 i 1-0..1

```
public class Account {
    public Debitcard getCard() { ... }
    public void setCard (DebitCard card) { ... }
    public void removeCardC) { ... }
    private DebitCard theCard ;
}
```




```
public class DebitCard
{
    public DebitCard (Account a)    { ... }
    public Account  getAccount()    {  ... }
    private Account  theAccount   ;
}
```

- Immutable associjacija sa strane DebitCard
- Mutable i optional asocijacija sa strane Account

# Multiplikativnosti 1-1 i 1-0..1 (2)

- Primjer, napravljen je novi objekat klase DebitCard koji je potrebno povezati sa objektom klase Account

```
Account accl = new Account() ;
DebitCard cardl = new DebitCard(accl);
accl.setCard(cardl) ;
```

- Bolje rješenje je da se kreiranje potrebnih linkova enkapsulira u jendoj klasi. Npr. pogrešno je:

```
Account accl = new Account(), acc2 = new Account() ;
DebitCard cardl = new DebitCard(acc2) ;
accl,setCard(cardl) ;
```

# Multiplikativnosti 1-1 i 1-0..1 (3)

- DebitCard objekat se pravi u metodi addCard iz klase Account
- Konstruktor za klasu DebitCard nije public, objekti ove klase mogu da se kreiraju jedino unutar metoda klase Account, koncept friend iz C++

```
public class Account {
    public DebitCard getCard() {
        return theCard;
    }
    public void addCard()    {
        theCard = new DebitCard(this) ;
    }
    private Debitcarct thecard ;
}

public class Debitcard {
    DebitCard(Account a) {
        theAccount = a;
    }
    public Account getAccount() {
        return theAccount;
    }
    private Account theAccount;
}
```

# Multiplikativnosti 1-1 i 1-0..1 (4)

- Mutable asocijacija sa obje strane

```
public class Account {
    public DebitCard getCard() {...}
    public void addCard(DebitCard c) { ... }
    public void removecard() {
        theCard = null ;
    }
    private DebitCard theCard ;
}

public class DebitCsrd {
    public DebitCard(Account a) { ... }
    public Account getAccount() { ... }
    public void changeAccount(Account newacc) {
        if (newacc.getCardl) != null) {
        // throw AccountAlreadyHasACard
        }
    theAccount .removeCard ();
    newacc.addCard(this);
    }
    private Account  theAccount;
}
```

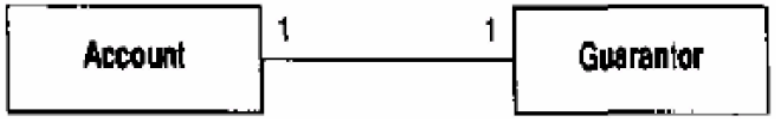
# Multiplikativnost 1-*

- Slično kao u slučaju jednosmjerne asocijacije
- Klasa Customer može da bude odgovorna za održavanje referencijalnog integriteta

# Dvosmjerne immutable asocijacije

```
public class Account
{
    public Account(Guarantor g)     { theGuarantor = g; }
    public Guarantor getGuarantor() { return theGuarantor; }
    private Guarantor theGuarantor ;
}

public  class Guarantor
{
    public Guarantor(Account a] { theAccount = a; }
    public Account getAccount0 { return theAccount; }
    private Account theAccount ;
}
```

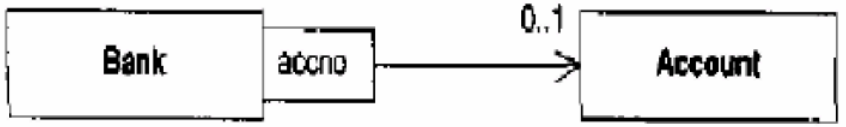


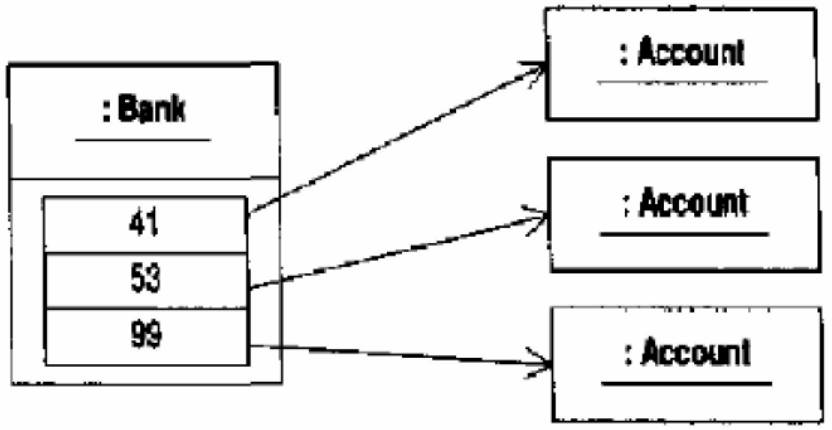

- Implementacija u C++
- U Javi nije moguće koristiti objekat prije nego bude kreiran (objekat a). Potreban je default konstruktor + seter metoda

```
Account a  = new Account(new Guarantor(a)) ;
Guarantor g = a.getGuarantor();
```

# Kvalifikovana asocijacija

- Banka sadrži više računa, svaki račun je identifikovan sa accno (broj računa)
- Struktura podataka koja implementira efikasno pristupanje objektima na osnovu njihovog identifikatora
  - lookup table umjesto običnog vektora pokazivača

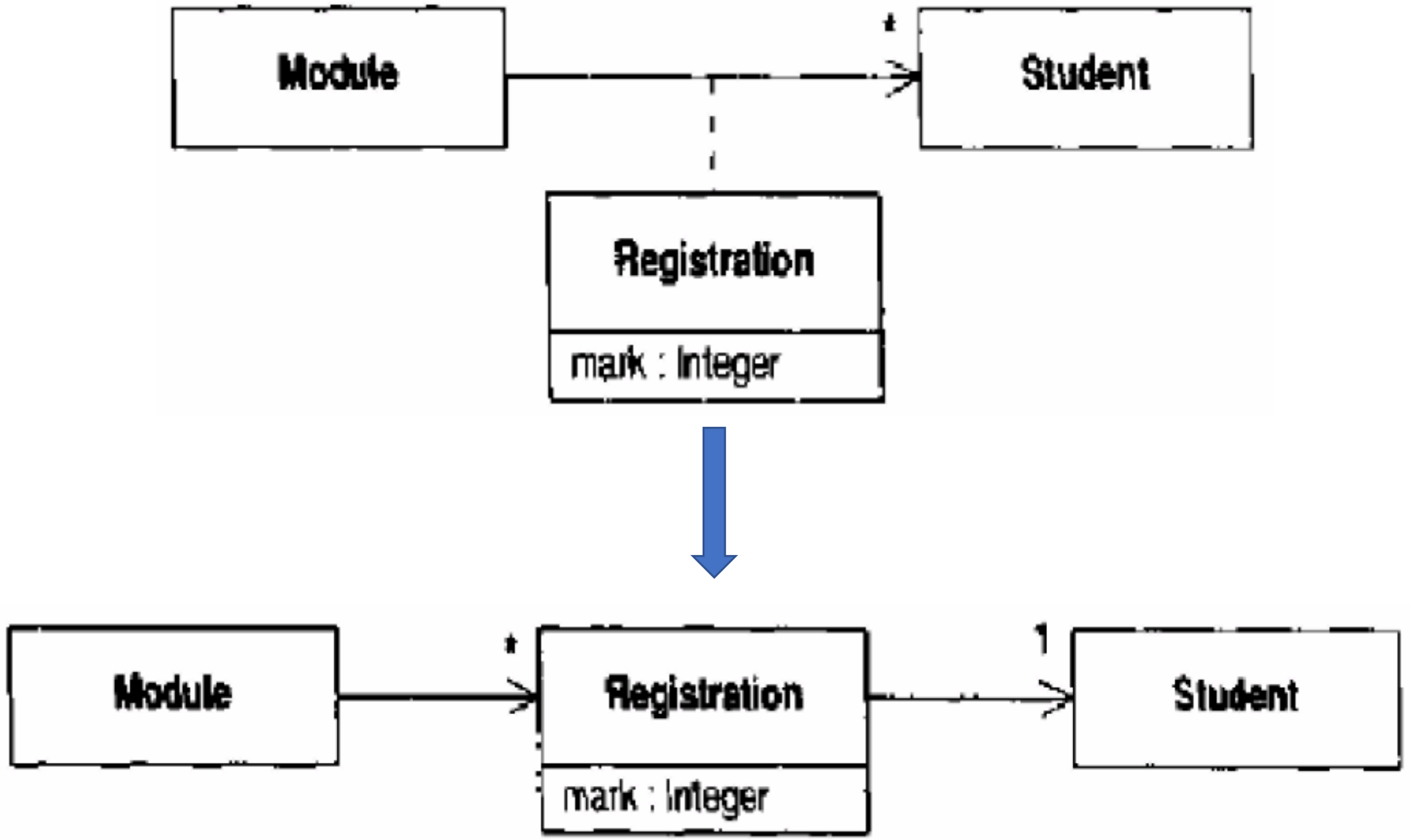


# Kvalifikovana asocijacija (2)

```
public class Bank {
    public void addAccount (Account a) {
        theAccounts.put(new Integer(a.getuumber()), a);
    }
    public void removeAccount(int accno) {
        theAccounts.remove(new Integer(accno));
    }
    public Account lookupAccount(int accno) {
        return (Account) theAccounts.get(new Integer(accno));
    }
    private Hashtable theAccounts ;
}
```

# Asocijativna klasa



# Implementacija ograničenja

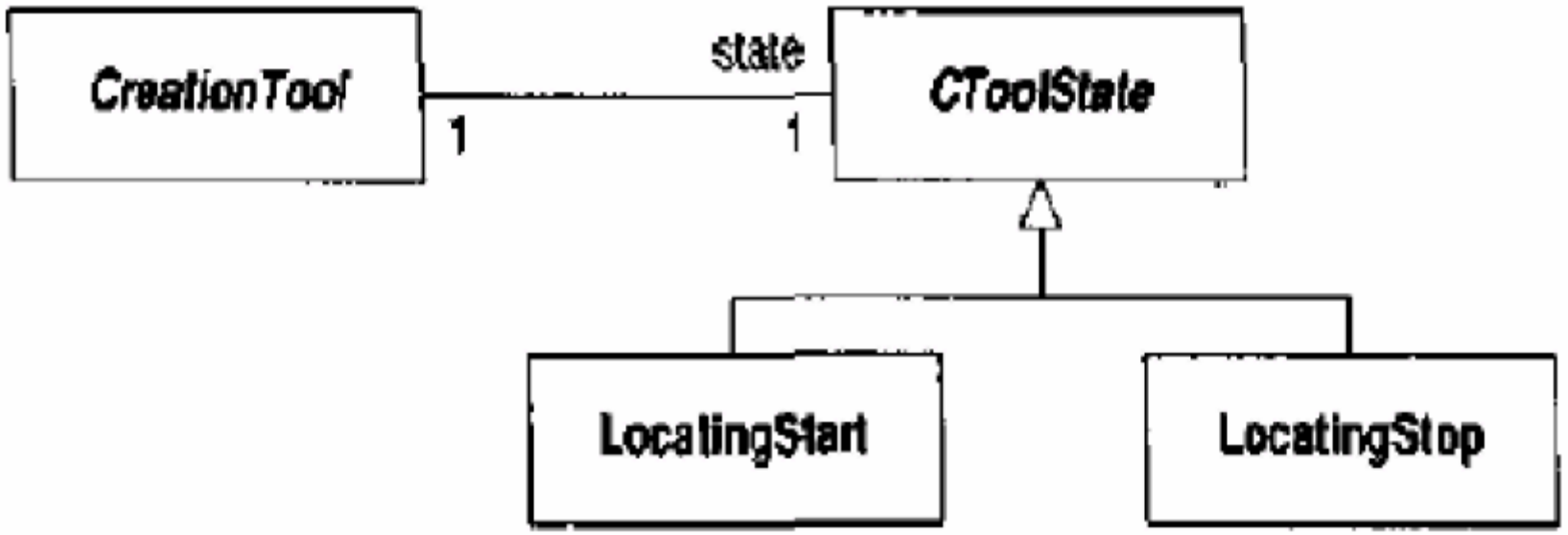
```
public class SavinqsAc count
{
    public void withdraw(double amt) {
    if (amt >= balance) {
        // throw PreconditionUnsatisfied ) balance -= amt ;
    }
    private double balance ;
}
```

# Implementacija dijagrama stanja

- Osnovni metod
  - Stanja se predstavljaju podatkom članom koji je nabrojivog tipa
  - Metode su implementirane sa swich naredbom, svaki case predstavlja stanje iz dijagrama i tranzicije
- Ako se dodaje novo stanje mora da se mijenja implementacija svih metoda
- Redudantnost koda, mnogo praznih case-ova

# Implementacija dijagrama stanja (2)

- Reprezentacija stanja klasama
  - Objekat klase CreationTool sadrži referencu na objekat koji predstavlja njegovo stanje



```
public class CreationTool {
    public void press() {
        state.press();
    }
    private CToolState state;
}
```

```
public  abstract  class CToolState {
    public abstract void press();
}
```

```
public class LocatingStart extends CToolState
{
    public void press() {
        set start position to current ;
        draw faint image of shape ;
        set current state to 'LocatingStop' ;
    }
}
```

# Implementacija dijagrama stanja (3)

- Klase koje predstavljaju stanja obezbjeđuju implementaciju interfejsa iz klase CreationTool
  - Kada objekat klase CreationTool primi poruku, prosto je prosljeđuje objektu koji predstavlja trenutno stanje
- Interfejs klase CToolState sadrži sve poruke koje mogu na prethodni način biti proslijeđene
- Kako se implementira promjena stanja? (set current state to LocatingStop)

# Perzistentnost

- Snimanje kreiranih objekata i njihovo ponovno učitavanje
- Osnovna jedinica za definisanje perzistentnosti je klasa
  - Asocijacija između perzistentnih klasa je perzistentna
  - Asocijacija između perzistentne i tranzijentne klase je tranzijentna
  - Atributi perzistentne klase su perzistentni
- Problem sa referencama koje su tranzijentne
- Serijalizacija
  - Java: interfejs Serializable, writeObject, readObject
- Mapiranje na relacioni model i upotreba relacionih SUBP kroz odgovarajuće API