DD1385

Programutvecklingsteknik

Några bilder till föreläsning 8

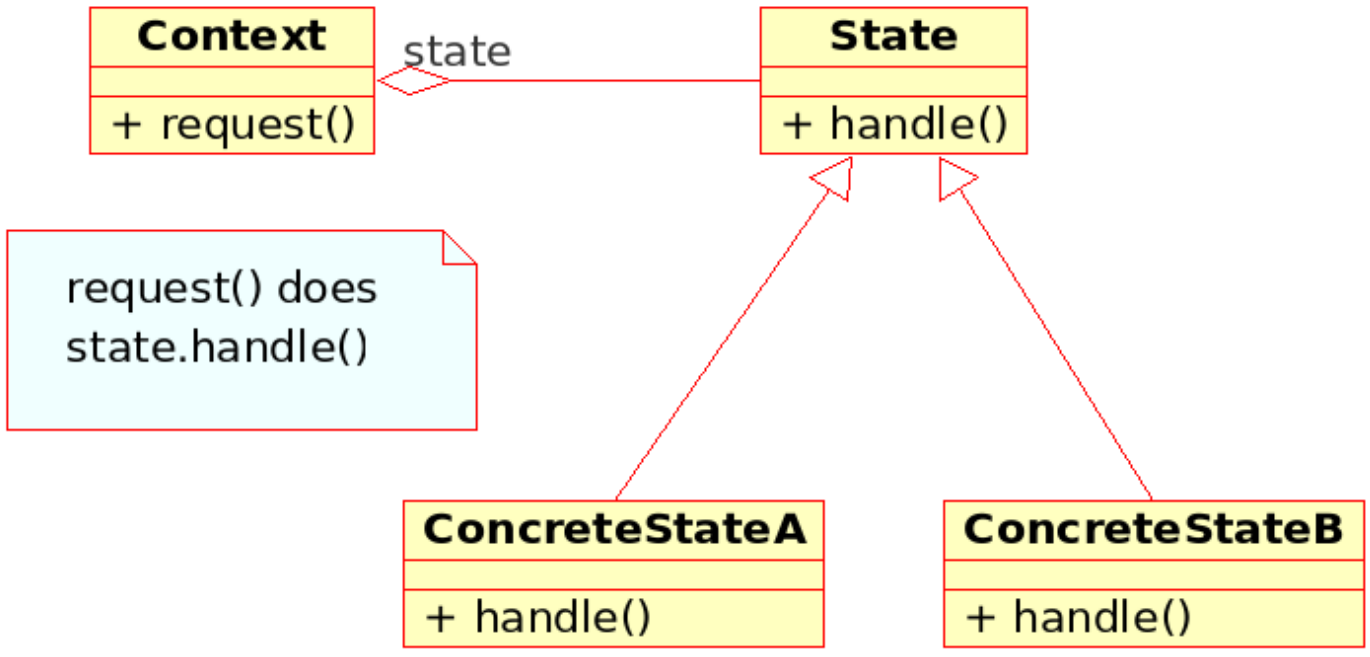
Innehåll

- Det som blev kvar från föreläsning 7
  - Designmönster: State
  - Enum, uppräkningstyp
- Testning med JUnit
- Refactoring

Mönstret State

- Ett objekts beteende ändras när dess tillstånd ändras
- Istället för if-satser som testar en tillståndsvariabel:
  Låt ett objekt ta hand om tillståndet.
- När tillståndet ändras, byt aktuellt tillståndsobjekt.

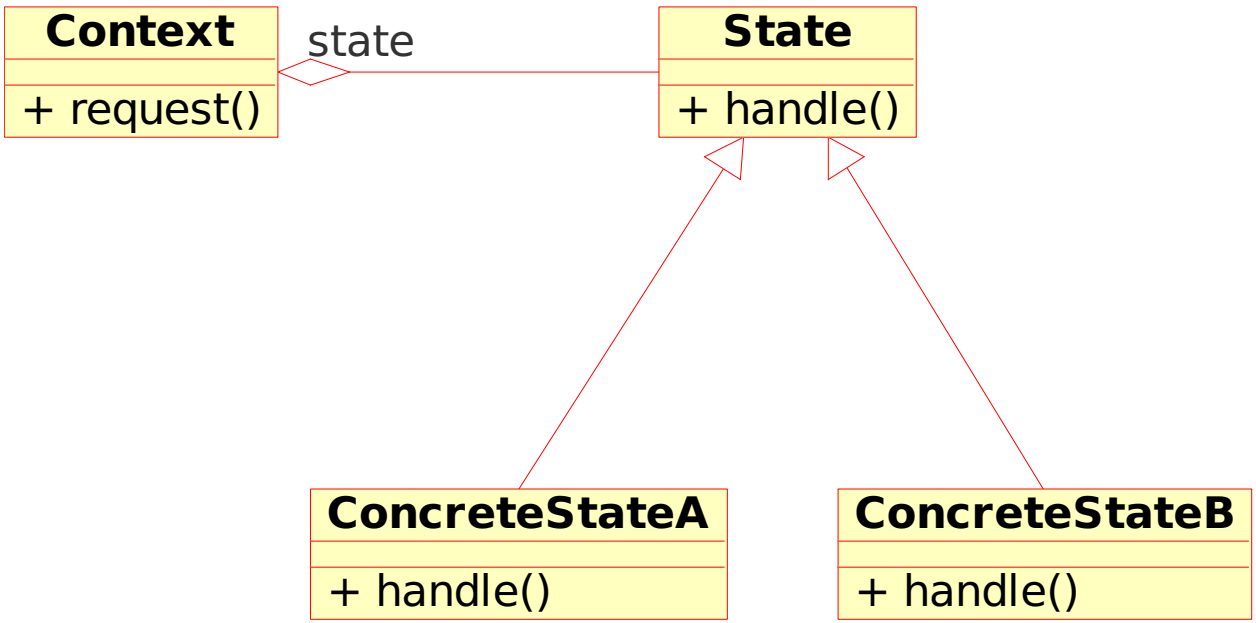
State


Context
+ request()
state
State
+ handle()
request() does
state.handle()
ConcreteStateA
+ handle()
ConcreteStateB
+ handle()

```
class Game {

  int level;

  void move1() {
    if (level == 1) {...}       // simple action
    else if (level == 2) {...}// med. level action
    else if (level == 3) {...}// advanced action
  }

  void move2() {
    if (level == 1) {...}       // simple action
    else if (level == 2) {...}// med. level action
    else if (level == 3) {...}// advanced action
  }

}
```

```
class Game {
  int level;
  State currentState = new Level1State(this);

  void move1() {
    currentState.move1();
  }

  void move2() {
    currentState.move2();
  }

  void checkstate() {
    // if state change is required, change the
    // state object. If-clauses needed here!

      currentState = ... //change the state object
  }
}
```

State


Context
+ request()
state
State
+ handle()
ConcreteStateA
+ handle()
ConcreteStateB
+ handle()

Uppräkningstyper – enum

Definiera egen typ med *egna* värden:

```
enum Day {Monday, Tuesday, Wednesday,
          Thursday, Friday, Saturday,
          Sunday}
```

Day är en klass med konstanta värden.
Värdena refereras med

```
Day.Wednesday      Day.Saturday
```

```
Day theDay = Day.Saturday;
```

```
Day[] specialdays =
{Day.Tuesday, Day.Thursday, Day.Sunday};
```

Uppräkningstyper – enum

Alla värden inom typen:

```
for (Day d: Day.values())
    System.out.println(d);
```

ger utskrift

```
Monday
Tuesday
Wednesday
Thursday
...
```

Uppräkningstyper – enum

Ordningsnummer med ordinal():

```
for (Day d: Day.values())
    System.out.print(d.ordinal() + " ");
```

ger utskrift

```
0 1 2 3 4 5 6
```

Går att jämföra: `implements Comparable`

```
System.out.println
  (Day.Thursday.compareTo(Day.Saturday));
```

ger utskrift

```
-2
```

Spelkort med enum

```
class Spelkort {
    Farg farg;
    Valor valor;

    Spelkort(Farg f, Valor v){
        farg = f;
        valor = v;
    }

    public String toString () {
        return farg + "-" + valor;
    }
}
```

Kortlek

```
class Kortlek {
  Spelkort[] lek = new Spelkort[52];

  Kortlek() {
    int i=0;
    for (Farg farg: Farg.values())
      for (Valor valor:Valor.values())
        lek[i++] =
          new Spelkort(farg,valor);
  }
}
```

Metoder att lägga till i Kortlek : toString()

En lång String med 4 kort per rad

```
public String toString() {
    StringBuilder allt = new StringBuilder();
    int rad = 0;
    for (Spelkort s : lek){
        allt = allt.append(s + " ");
        if (rad++ == 4){
            allt.append("\n");
            rad = 0;
         }
    }
    allt.append("\n");
    return allt.toString();
}
```

Blanda korten

`Arrays.asList(kortlek)`

gör lista intimt kopplad till en array (här kortlek).

Ändringar i listan görs även i arrayen.

```
Collections.shuffle(Arrays.asList(kortlek));
```

Listan blandas $\rightarrow$ Kortleken blandas

Metoder att lägga till i Kortlek : blanda()

```
void blanda() {
    Collections.shuffle(Arrays.asList(lek));
}
```

Testa kortleken: skapa, skriv ut, blanda, skriv ut

```
public static void main(String[] u) {
    Kortlek lek = new Kortlek();
    System.out.println("Kortleken fran borjan:");
    System.out.println(lek);
    lek.blanda();
    System.out.println("\nKortleken blandad:");
    System.out.println(lek);
}
```

Gamla Spelkort

```
class Spelkort {
    String farg;
    int valor;

    Spelkort(String f, int v){
        farg = f;
        valor = v;
    }

    public String toString () {
        return farg + "-" + valor;
    }
}
```

Varför är nya Spelkort bättre än gamla ?

- Med enum så är Farg och Valor väldefinierade
- Största och minsta värden är säkert tillgängliga
  `values()` `ordinal()`
- Med gamla Spelkort kan man skapa t.ex.
  `new Spelkort("VIRUS",-1729)`

Enum är mer än enstaka värden

- Enum är klass med konstruktor och metoder (om man vill)
- Enum - klasser är `final`, går ej att ärva från
- Enum med ett enda värde $\rightarrow$ `Singleton`

```
public enum Singleton {THEONLY}
```

Singleton med lite innehåll

```
public enum Snowwhite {
  THEONLY;

  boolean married = false;
  String occupation;
  int age;

  void incrAge () {
    age++;
  }

  public String toString() {
    return    "Snowwhite " + age;
  }
}
```

Sju dvärgar

```
public enum Dwarf {
    Blick(131), Flick(112), Glick(142),
    Snick(115), Plick(108), Whick(120), Quee(99);

    int age;

    Dwarf (int a) {
        age = a;
    }

    public String toString() {
        return super.toString() + " " + age;
    }

    // fler metoder
}
```

Varje dvärg (konstant instans av klassen) definieras med ett namn och ett kontruktoranrop

```
Plick(108)    //age=108 is set in constructor
```

Sju dvärgar, mainmetod

```
public static void main (String[] u) {
    System.out.println("All dwarfes");
    for (Dwarf d : Dwarf.values())
        System.out.print(d + "  ");
    System.out.println();
}
```

Spelkort med enum

```
public enum Farg
  {HJARTER, KLOVER, RUTER, SPADER}


public enum Valor
  {ESS, TVA, TRE, FYRA, FEM, SEX,
   SJU, ATTA, NIO, TIO, KNEKT,
   DAM, KUNG}
```

Testning med JUnit

- Ramverk i Java för testning av Java-klasser
- Utvecklat av Gamma & Beck,
  första versionen kom 2002
- `@Test` = annotering för testmetod
- `@Suite.SuiteClasses(...)`
  Klasser med testmetoder bygger upp testsviter
  enligt mönstret *Composite*
- Textgränssnitt eller grafiskt
- (Ganska) enkelt att använda
- Gratis på www.junit.org

Refactoring

- Förbättra programkoden
  - Minskad koduppprepning
  - Ökad flexibilitet
  - Tydligare
  - Effektivare
- ”Utsidan”/gränssnitt mot användare ändras ej
- Funktionaliteten ändras ej
- Svårt
- Några enkla exempel visas

Refactoring - koduppprepning inom klass

```
class C {
  void m1() {
    <A> do1(); do2(); do3(); <B>
  }
  void m2() {
    <C> do1(); do2(); do3(); <D>
  }
}
```

förbättras till

```
class C {
  void doAll { do1(); do2(); do3(); }
  void m1() { <A> doAll(); <B>}
  void m2() { <C> doAll(); <D>}
}
```

Refactoring - koduppprepning i olika klasser

```
class A {
  void m1() {
    <A> do1(); do2(); do3(); <B>
  }
}

class B {
  void m2() {
    <C> do1(); do2(); do3(); <D>
  }
}
```

förbättras genom arv eller delegering

Arv

```
class C {
  void doAll { do1(); do2(); do3(); }
}

class A extends C {
  void m1() { <A> doAll(); <B>}
}

class B extends C {
  void m2() { <C> doAll(); <D>}
}
```

Delegering

```
class Helper {
  void doAll() {
     do1(); do2(); do3();
  }}

class A {
  Helper helper = new Helper();
  void m1() {<A> helper.doAll(); <B> }
  ..
}

class B {
  Helper helper = new Helper();
  void m2() {<B> helper.doAll(); <D>}
  ..
}
```

Koduppprepning igen

```
class A {
  void m() {
    <common 1>
    <A–spec>
    <common 2>
  }
}

class B {
  void m() {
    <common 1>
    <B–spec>
    <common 2>
  }
}
```

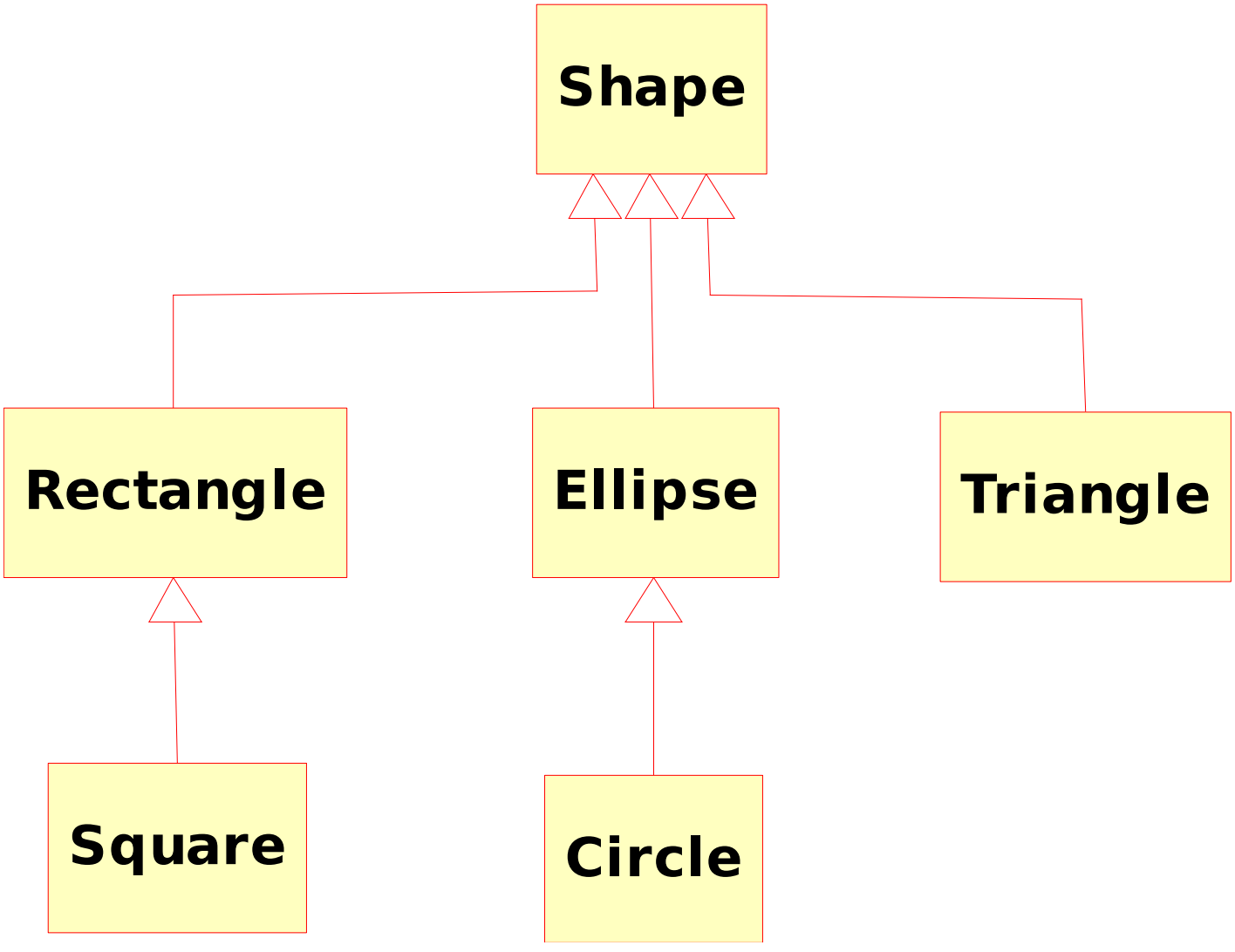
Koduppprepning igen, forts.

```
class C {
  void specm(){}; // tom eller abstrakt
  void m() {
    <common 1>
    specm();
    <common 2>
  }}

class A extends C {      // Vilket designpattern
  void specm() {         // ar detta ?
    <A–spec>
  }}

class B extends C {
  void specm() {
    <B–spec>
  }}
```

LSP
Liskov Substitution Principle

- "Subtypes must be substitutable for their base types"
- Ett objekt av en subklass ska kunna användas där objekt av basklassen/superklassen används

LSP motiveras med ett exempel


Shape
Rectangle
Ellipse
Triangle
Square
Circle

```
public class Rectangle extends Shape {
    private int width, height;
    public void setWidth (int w) {
        width = w;
    }
    public void setHeight (int h) {
        height = h;
    }
    public int getWidth () {
        return width;
     }
    public int getHeight () {
        return height;
    }
    public int area () {
        return width*height;
    }
}
```

Square ärver från Rectangle, där width == height

```
public class Square extends Rectangle{
    public void setWidth (int w) {
        super.setWidth(w);
        super.setHeight(w);
    }
    public void setHeight (int h) {
        super.setHeight(h);
        super.setWidth(h);
    }
}
```

Lite slösaktigt att alla kvadrater har ett extra datafält!

Ett testprogram

```
class Test {
  static void test (Rectangle r) {
    r.setWidth(4);
    r.setHeight(5);
    System.out.println
      ("Area is " + r.area() +
       "    should be 20");
  }

  public static void main (String[] a) {
    test (new Rectangle());
    test (new Square());
  }
}
```

Utskrift från Test blir

```
Area is 20 should be 20
Area is 25 should be 20
```

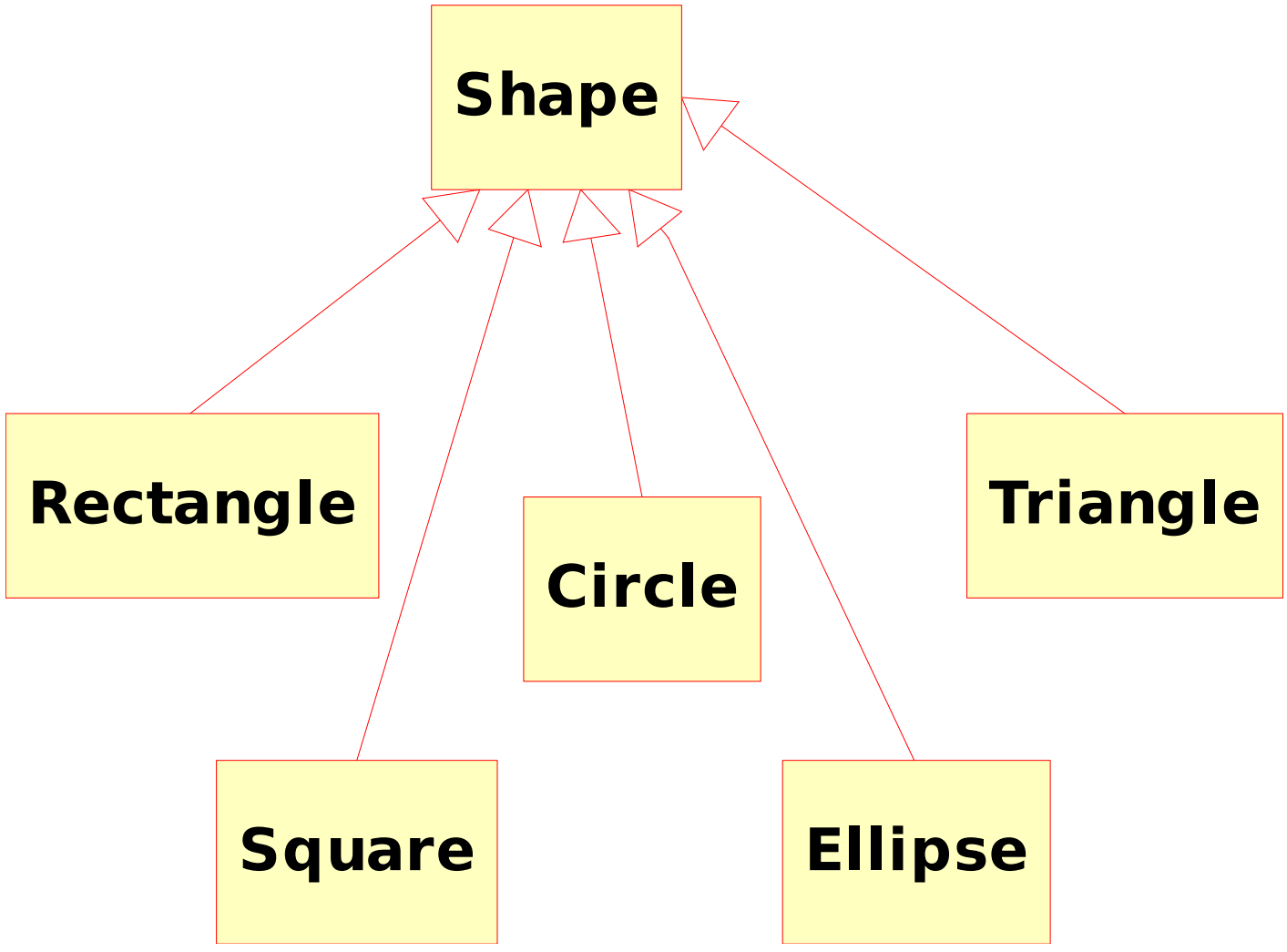
LSP
Liskov Substitution Principle

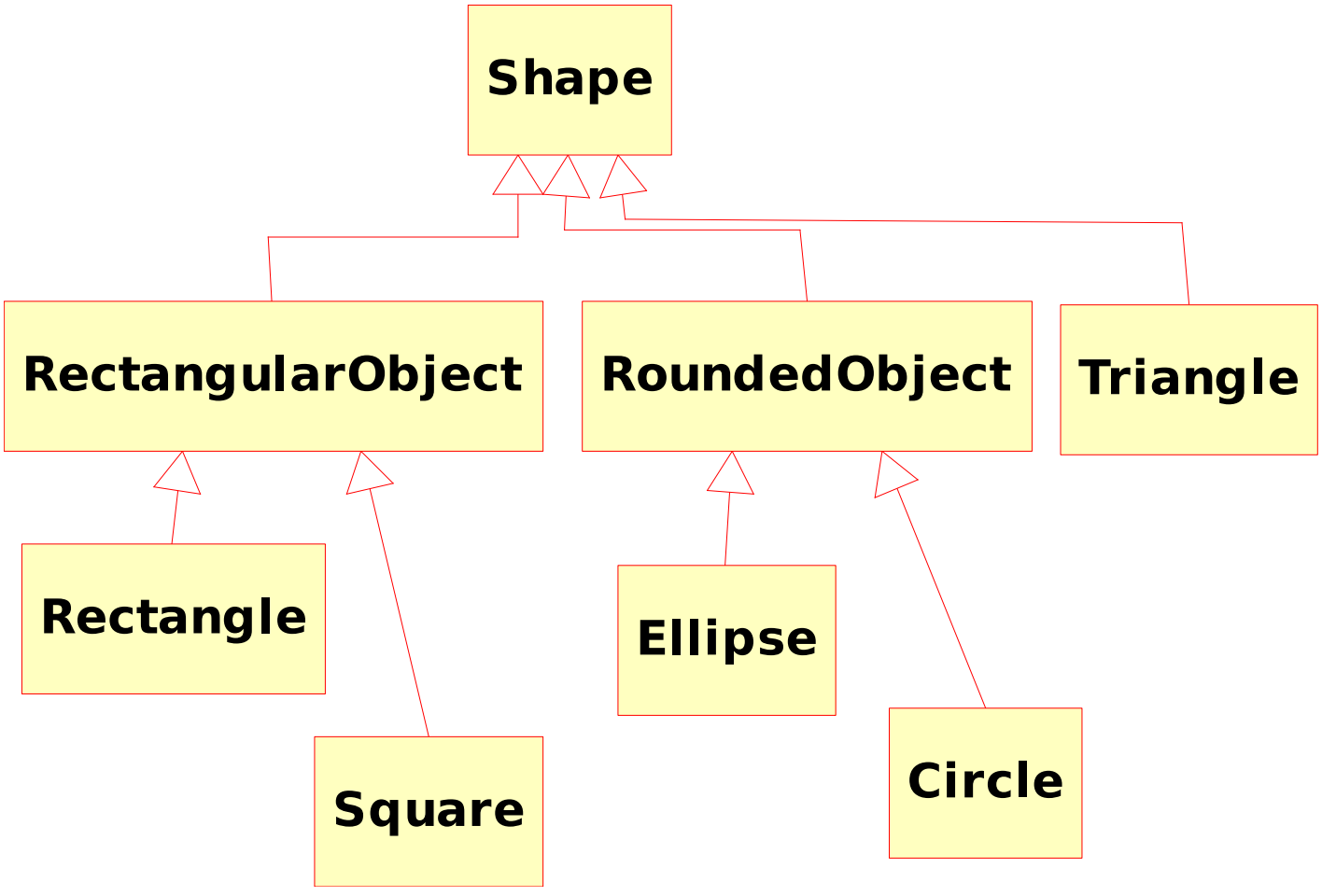
- "Subtypes must be substitutable for their base types"
- Ett objekt av en subklass ska kunna användas där objekt av basklassen/superklassen används

Square & Rectangle bryter mot LSP eftersom
Square inte kan ersätta Rectangle
i exemplet.

Vad betyder relationen "är en" ?

- En Square är en Rectangle
- Men en Square *uppför sig inte* som en Rectangle !!!
- En Rectangle har egenskapen att width och height kan ges värden oberoende av varandra.
- Bryt mot LSP endast vid signifikant vinst! Exempel?


Shape
Rectangle
Circle
Triangle
Square
Ellipse


Shape
RectangularObject
RoundedObject
Triangle
Rectangle
Square
Ellipse
Circle

RectangularObject

```
abstract class RectangularObject extends Shape{
    abstract int area ();
    abstract int width();
    abstract int height();
}
```

Rectangle

```
class Rectangle extends RectangularObject{
    int width, height;

    void setWidth (int w) { width = w; }
    void setHeight (int h) { height = h; }

    int width() { return width; }
    int height() { return height; }

    int area () {
        return width*height;
    }
}
```

Square

```
class Square extends RectangularObject {
    int size;
    void setSize (int s) {
        size = s;
    }
    int width() {
        return size;
    }
    int height() {
        return size;
    }
    int area () {
        return size*size;
    }
}
```