Föreläsning 5
När skall implementationsarv användas?

När skall man använda implementationsarv?

Implementationsarv är en konstruktion som kan användas för att få elegant kod

- lättläst
- tydlig
- utbyggbar
- återanvändbar

Men implementationsarv är också en konstruktion som ofta missbrukas, vilket leder till kod som blir

- svårbegriplig
- oflexibel
- svår att underhålla.

Motiveringar för att använda implementationsarv

Motiveringar till att använda implementationsarv:

- Återanvändning av kod

  Klass B har viss kod (data och metoder) som är identisk med kod i klass A.

- Is-a relation

  Varje objekt av klassen B ”är” också ett objekt av klassen A. Vilket betyder att mängden av alla objekt av klassen B är en delmängd av alla objekt i klassen A.

- Det publika gränssnittet

  Det publika gränssnittet av klass B inkluderar det publika gränssnittet av klass A.

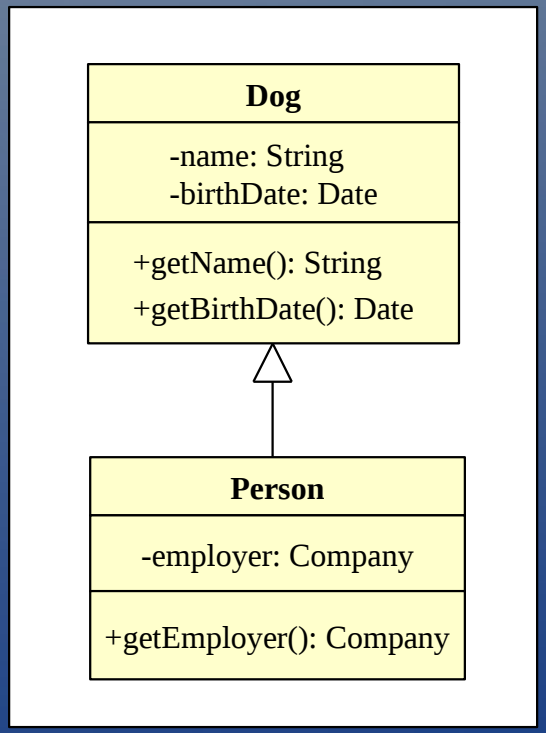
- Polymorfism

  Det finns ett behov av att kunna byta ut basklassen mot underklasser, dvs kunna tilldela ett objekt av klassen B till en variabel som deklareras av klassen A.

Implementationsarv kontra återanvändning av kod

En av de verkliga fördelarna med implementationssarv är återanvändning av kod. Men är återanvändning av kod i sig en tillräcklig anledning att använda implementationsarv?

Betrakta nedanstående UML-diagram:



Möjligheten att ärva kod som annars skulle behöva dupliceras är en användbar egenskap hos objektorienterade språk, och därför skall man alltid överväga möjligheten att använda implementationsarv.

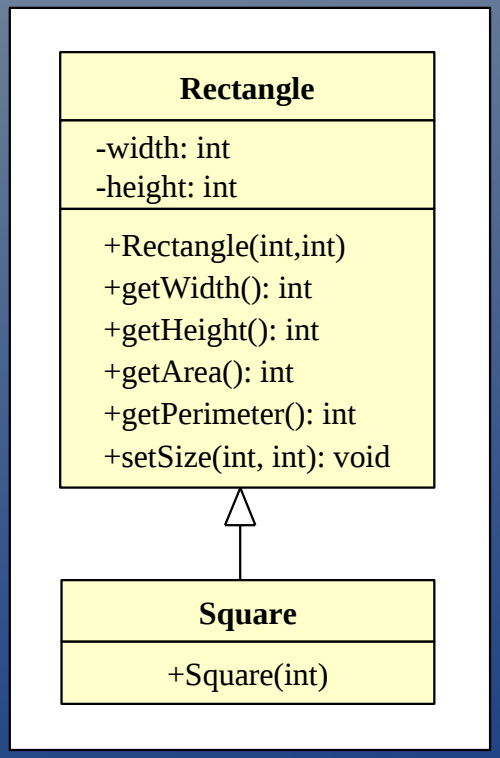
Men enbart återanvändning av kod rättfärdigar inte implementationsarv.

En person är ingen hund!!

Implemationsarv kontra ”is-a”-relation

Antag att vi skall utveckla ett system där vi behöver modellera geometriska rektanglar och kvadrater.

Från matematiken vet vi att en kvadrat är en speciell typ av rektangel. Således verkar det självklart att använda implementationsarv.



All kod i Rectangle återanvänds. Subklassen Square behöver endast implementera konstruktorn!

Implemationsarv kontra ”is-a”-relation

```
public class Rectangle {
    private int width, height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }//constructor

    public int getWidth() {
        return width;
    }//getWidth

    public int getHeight() {
        return height;
    }//getHeight

    public int getArea() {
        return width * height;
    }//getArea

    public int getPerimeter() {
        return 2 * (width + height);
    }//getPerimeter

    public void setSize(int width, int height) {
        this.width = width;
        this.height = height;
    }//setSize
}//Rectangle
```

```
public class Square extends Rectangle {
    public Square(int size) {
        super(size, size);
    }//constructor
}//Square
```

Implementationsarv kontra ”is-a”-relation

Det finns dock ett allvarligt problem med denna design. Eftersom klassen Square ärver alla publika metoder i superklassen Rectangle, kommer klassen Square att ha metoden

```
public void setSize(int w, int h)
```

som har två parametrar.

Anropet

```
square.setSize(5, 10);
```

gör att höjden och bredden på kvadraten square blir olika!!

En setSize-metod på en kvadrat skall endast ha en parameter.

Vad kan vi göra åt detta?

Implementationsarv kontra ”is-a”-relation

Det är möjligt att överskugga setSize-metoden i superklassen, t.ex kan vi i klassen Square ge metoden

```
public void setSize(int width, int height) {
    this.width = width;
    this.height = width;
}//setSize
```

Denna ändring garanterar att höjden och bredden av en kvadrat är lika stora, men lösningen är dålig:

- åtkomstmodifierarna för instansvariablerna width och height i klassen Rectangle måste ändras från **private** till **protected**
- det är förvirrande att ange storleken på en kvadrat med två parametrar
- om en klient tror att en variabel r refererar till ett objekt av typen Rectangle, men i verkligheten refererar till ett objekt av typen Square ändrar inte anropet

  r.setSize(5, 10);

  höjden av objektet till det värde användaren förväntar sig.

The Open-Closed Principle (OCP)

Att vi behöver ändra åtkomstmodifierarna för instansvariablerna i klassen Rectangle, när vi utvecklar subklassen Square innebär att vi bryter mot *The Open-Closed Principle*:.

> ***Software modules should be both open for extension and closed for modification. (Bertrand Meyer)***

Denna princip säger att en modul skall vara öppen för utvidgningar, men sluten för förändringar.

Superklassen Rectangle är inte sluten för förändringar när vi utvidgar med subklassen Square.

Implementationsarv kontra ”is-a”-relation

Att slippa ange storleken på en kvadrat med både bredden och höjden kan vi lösa genom att överlagra setSize-metoden i Square:

```
public void setSize(int size) {
    this.width = size;
    this.height = size;
}//setSize
```

Men detta åtgärdar inte grundproblemet med vår design. Nämligen att det fortfarande är möjligt att skriva

```
Rectangle r = new Square(8);
. . .
r.setSize(5, 10);
```

Den tidigare problematiken kvarstår alltså.

The Principle of Least Astonishment

Om en klient tror att den har en referens till ett objekt av typen A, men i verkligheten har en referens till ett objekt som är av subtypen B, skall det inte bli några överraskningar när klienten sänder meddelanden till objektet.

Vår design av klasserna Rectangle och Square bryter mot denna princip. Betrakta följande kodavsnitt:

```
Rectangle r;
. . .
r = new Square(8);
. . .
r.setSize(5, 10);
. . .
```

Om användaren tror att r är av typen Rectangle vid anropet av

```
r.setSize(5, 10);
```

förväntar sig användaren att r har höjden 10 efter anropet.

Inte att höjden är 5!!

Liskov Substitution Principle (LSP)

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T. (Barbara Liskov)

LSP säger att det är acceptabelt att göra klassen S till en subklass av klassen T, *om och endast om* det för varje publik metod som finns både i T och S gäller att S's metod som indata godtar alla värden som T's metod godtar samt att S gör alla bearbetningar på denna indata som T gör.

(S får däremot göra fler bearbetningar än T.)

Om en subtyp S till typen T uppfyller Liskov Substitution Principle säger vi att S är en *äkta subtyp* till T.

Liskov Substitution Principle (LSP)

Metoden `setSize` i klassen `Rectangle` har beteendet att ändra storleken på höjden oberoende av storleken på bredden. Metoden `setSize` i klassen `Square` kan omöjligt ha detta beteende och fortfarande ha kvar egenskapen att vara en kvadrat.

Från LSP följer att klassen `Square` inte kan implementeras som en subklass till klassen `Rectangle`, vilket innebär att *en kvadrat är INTE en rektangel!*

Har då matematikerna fel som säger att *en kvadrat ÄR en rektangel*?

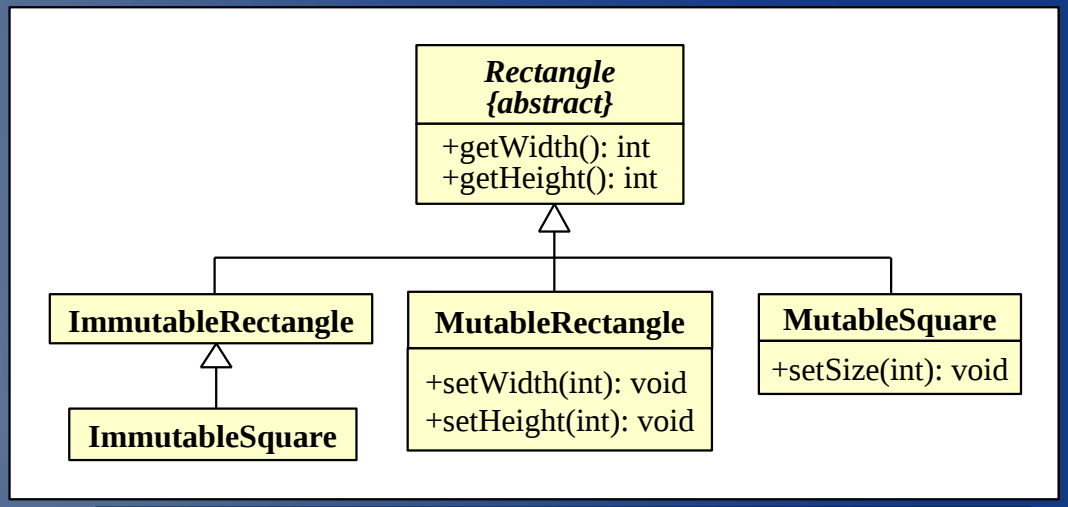
- Nej, matematiskt betraktas alla geometriska figurer som *icke-muterbara*, d.v.s. när man matematiskt ändrar storleken på en rektangel skapas en helt ny rektangel.
- Våra klasser `Rectangle` och `Square` är muterbara, så vad LSP egentligen säger är att *en muterbar kvadrat är INTE en muterbar rektangel*.

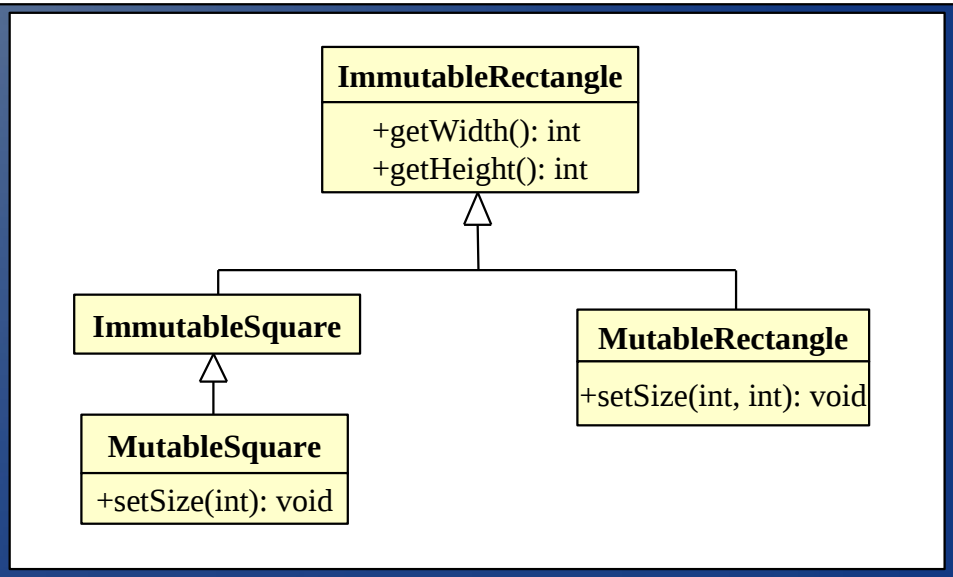
Implementationsarv kontra is-a-relation

Hur är då en lämplig relation mellan en muterbar kvadrat och en muterbar rektangel?

Ett sätt är att införa *en tredje klass*, som en gemensam superklass, och låta denna klass innehålla de gemensamma delarna.

Här visas två olika sätt på vilket detta kan göras.





Behövs klassen Square?

Innan man inför klassen Square skall man naturligtvis fråga sig om denna klass verkligen behövs.

- Vad tillför klassen Square som inte klassen Rectangle tillhandahåller?
- Är det nödvändigt att skilja mellan kvadrater och rektanglar?
  - Räcker det att klassen Rectangle tillhandahåller en metod isSquare() för att avgöra om höjden och bredden är lika?

”is-a”-relation och gemensamt gränssnitt

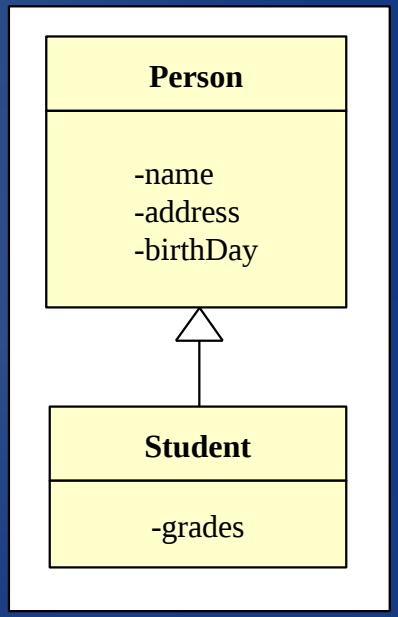
Ett klassiskt exempel för att introducera begreppet arv är klasserna Person och Student.

Uppenbarligen är en student en person, dvs ett objekt av klassen Student har alla egenskaper och alla beteenden som ett objekt av klassen Person har, såsom namn, födelsedag och adress.

Klassen Student har dessutom ytterligare egenskaper såsom utbildningsnivå, linje, inskrivningsår och avklarade poäng.

Man kan alltså här argumentera för att klassen Student skall vara en subklass till klassen Person.

Men är detta en bra lösning?



Implementation av klasserna Person och Student

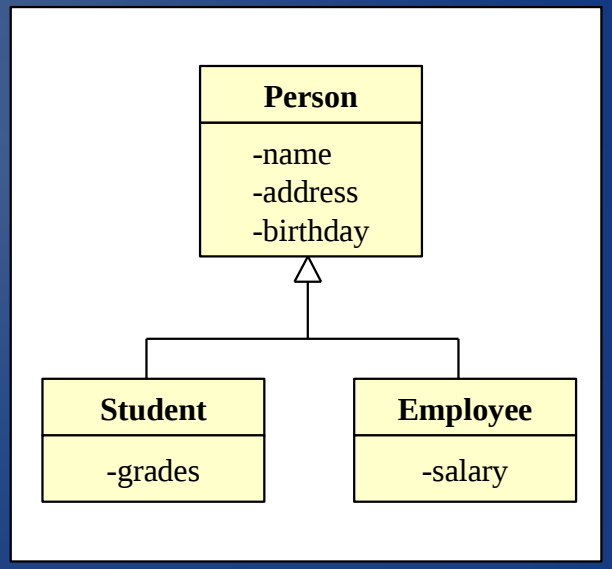
```
import java.util.Date;
public class Person {
   private String name;
   private String address;
   private Date birthdate;
   public Person(String name, String address, Date birthdate) {
      this.name = name;
      this.address = address;
      this.birthdate = birthdate;
   }//constructor
   public String getName() {
      return name;
   }//getName
   public String getAddress() {
      return address;
   }//getAddress
   public Date getBirthdate() {
      return birthdate;
   }//getBirthdate
   //more methods
}//Person
```

```
import java.util.Date;
public class Student extends Person {
   private Grade grades;
   public Student(Name name, Address address,
                  Date birthdate, Grade grades) {
      super(name, address, birthdate);
      this.grades = grades;
   }//constructor
   public Grade getGrades() {
      return grades;
   }//getGrades
}//Student
```

”is-a”-relation och gemensamt gränssnitt

Anta att klasserna Person och Student skall användas i ett system för att lagra uppgifter om studenter på ett universitet. Antag också att man i samma system vill lagra uppgifter om de anställda på universitetet. Således behövs ytterligare en klass Employee.

Av samma skäl som gällde för en student kan man argumentera för att Employee skall vara en subklass till klassen Person.



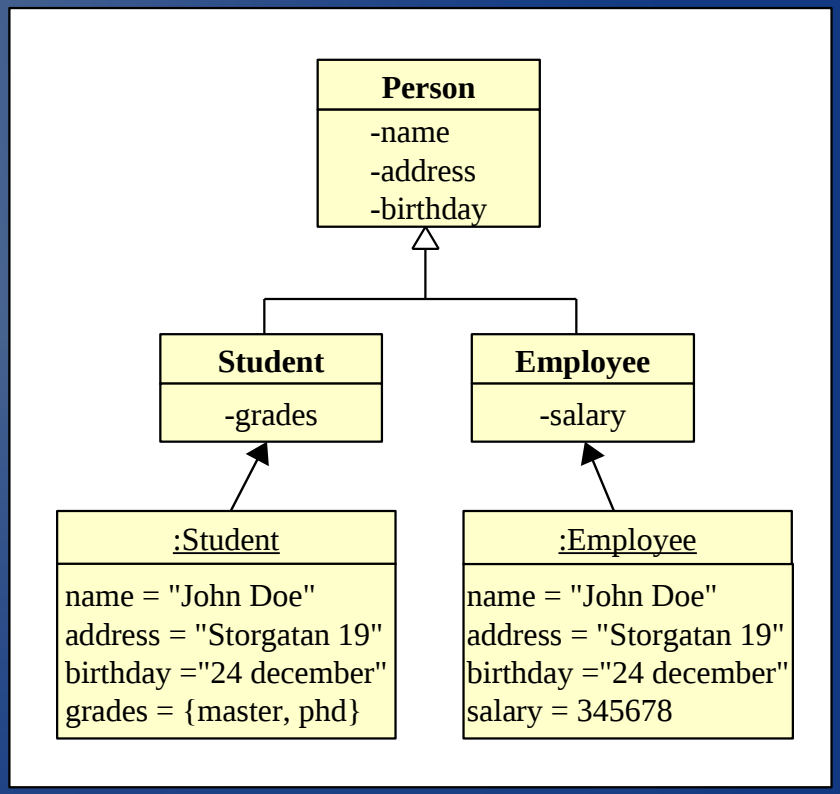
”is-a”-relation och gemensamt gränssnitt

Vad händer om en student blir klar med sin examen och får en anställning på universitetet eller om en student under sin studietid får en deltidsanställning som övningsledare?

Personen kommer att vara representerad av två objekt, ett Student-objekt och ett Employee-objekt.

Det gemensamma datat som finns i Person-klassen dupliceras!

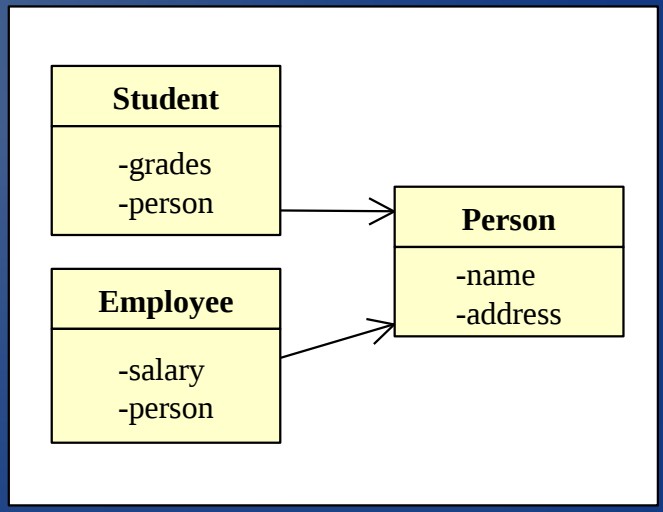
Ändras denna data måste detta åtgärdas i båda objekten, annars leder det till att datat blir osynkroniserad!



Delegering alternativ till implementationsarv

En bättre lösning än implementationsarv är i detta fall att se student och anställd som *en roll som en person spelar*. En sådan roll är ofta temporär, en person är inte alltid student eller anställd. På grund av den temporära karaktären hos rollerna (i vårt fall student och anställd) och den mera permanenta karaktären hos personen, är det *inte* lämpligt att låta klasserna `Student` och `Employee` vara subklasser till klassen `Person`.

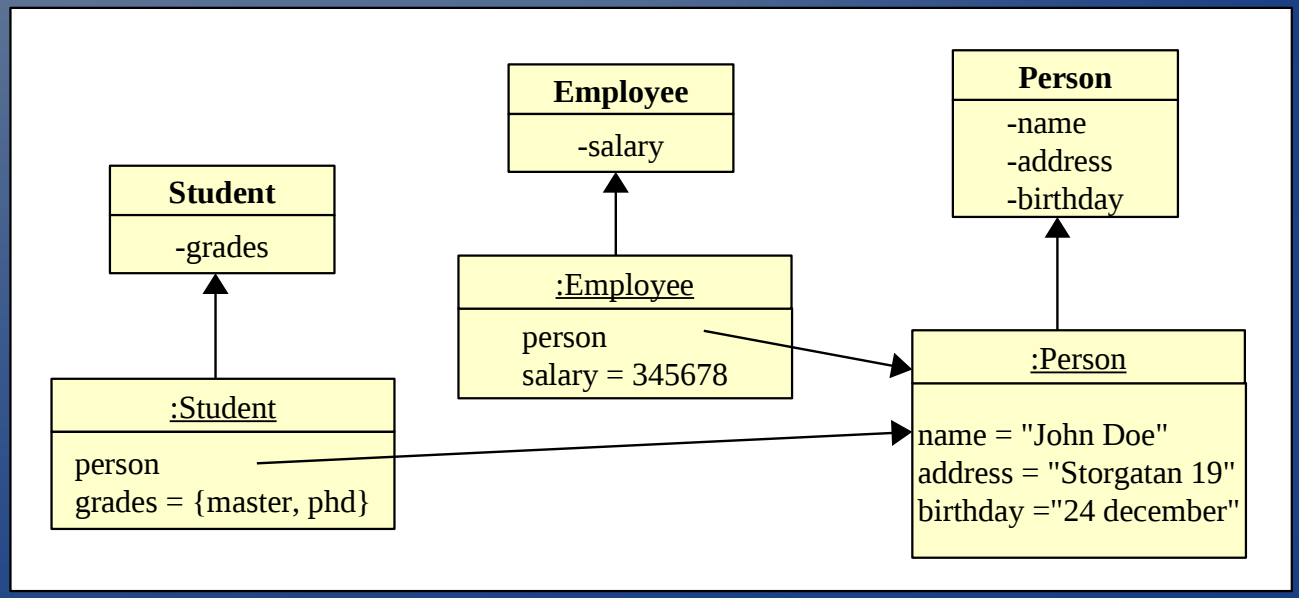
En bättre design är att använda delegering och låta klasserna `Student` och `Employee` ha ett objekt av klassen `Person`, till vilket de refererar.



Delegering innebär att ett objekt utåt erbjuder en viss tjänst, men internt överlåter ansvaret för att utföra tjänsten till ett annat objekt.

Delegering alternativ till implementationsarv

I vårt exempel innebär delegering att vi får exakt ett objekt av klassen Person för varje verklig person, som är oberoende av alla roller som spelas av denna person.



Den nya implementationen av klassen Student

```
import java.util.Date;
public class Student {
    private Person person;
    private Grade grades;
    public Student(Person person, Grade grades) {
        this.person = person;
        this.grades = grades;
    }//constructor
    public String getName() {
        return person.getName();
    }//getName
```

```
    public String getAddress() {
        return person.getAddress();
    }//getAddress
    public Date getBirthdate() {
        return person.getBirthdate();
    }//getBirthdate
    public Grade getGrades() {
        return grades;
    }//getGrades
}//Student
```

Slutsatser

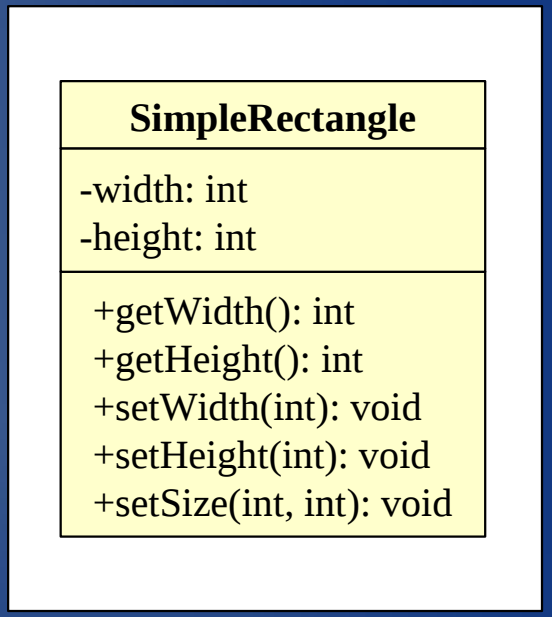
- Ett likadant gränssnitt är inte tillräckligt för ett elegant arvsförhållande, ett konsistent beteende är också nödvändigt.
- Subklasser skall *lägga till nya beteenden*, inte ändra redan befintliga beteenden.
- En klass A som är identisk med en klass B, förutom att den har *extra restriktioner* på sitt tillstånd, skall inte vara en subklass till B såvida klasserna inte är icke-muterbara.
- Gör så många klasser som möjligt icke-muterbara.
- Överväg alltid om en subklass som inte tillför något eget unikt beteende verkligen behövs.
- Använd aldrig arv utan att beakta *Liskov Substitution Principle*.

Implementationsarv = stark koppling

Ett problem med implementationsarv är den hårda kopplingen som subklasser har med sina superklasser.

Antag att vi har en klass SimpleRectangle enligt bredvidstående klassdiagrammet.

Säg nu att vi har behov av en ny klass som har samma beteenden som klassen SimpleRectangle förutom att klassen måste hålla reda på hur många gånger bredden av rektangeln förändras.



Här verkar implementationsarv lämpligt.

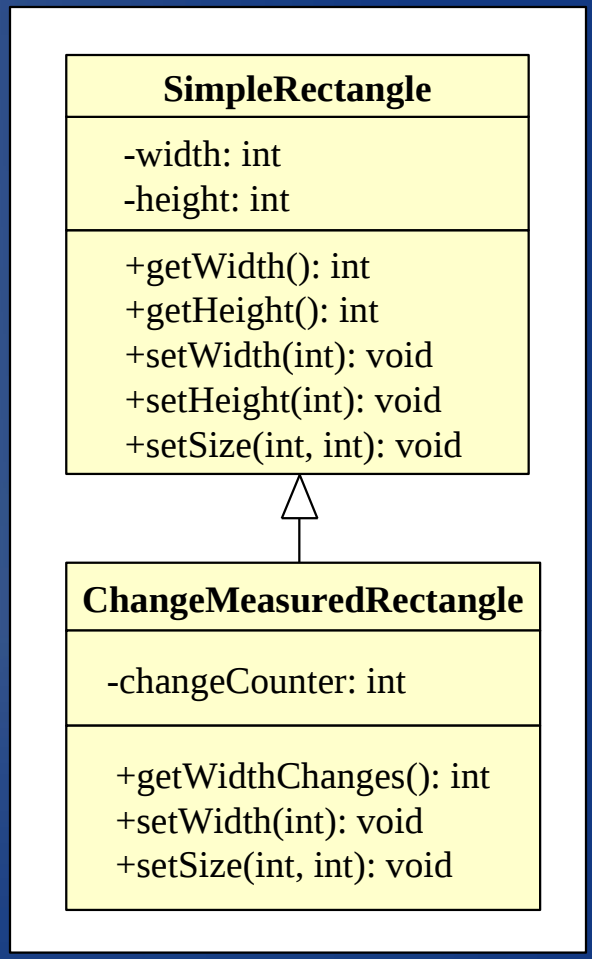
Implementationsarv = stark koppling

Vi skapar en klass, ChangeMeasuredRectangle, som är en subklass till SimpleRectangle.

Vi inför en instansvariabel changeCounter i ChangeMeasuredRectangle för att hålla reda på hur många gånger bredden av rektangeln förändras.

Vidare överskuggar vi metoderna setWidth och setSize, på så sätt att dessa metoder kontrollerar om bredden på rektangeln förändras och i så fall räknar upp instansvariabeln changeCounter.

Metoderna setWidth och setSize i klassen ChangeMeasuredRectangle utökar alltså beteendet hos motsvarande metoder i klassen SimpleRectangle.



Koden för ChangeMeasuredRectangle

```
public class ChangeMeasuredRectangle extends SimpleRectangle {
    private int changeCounter = 0;
    public ChangeMeasuredRectangle (int width, int height) {
        super(width, height);
    }//konstruktor
    public int getWidthChanges() {
        return changeCounter;
    }//getWidthChanges
    public void setWidth(int width) {
        if (width != getWidth()) {
            changeCounter++;
            super.setWidth(width);
        }
    }//setWidth
    public void setSize(int width, int height) {
        if (width != getWidth())
            changeCounter++;
        super.setSize(width, height);
    }//setSize
}//ChangeMeasuredRectangle
```

En snygg implementation!

Men är den korrekt?

Ett litet testprogram

Vi skriver ett litet testprogram och ser vad resultatet blir.

```
public static void main(String[] args) {
    ChangeMeasuredRectangle r = new ChangeMeasuredRectangle(5, 2);
    r.setWidth(4);
    r.setSize(6, 7);
    r.setWidth(8);
    r.setSize(12, 9);
    System.out.println("Antal ändringar: " + r.getWidthChanges());
}//main
```

Testprogrammet ger utskriften:

Antal ändringar: 6

Men vi har ju bara ändrat vidden på rektangeln 4 gånger!!

Vad är galet? För att ta reda på detta måste vi veta hur klassen SimpleRectangle är implementerad.

Koden för SimpleRectangle

```
public class SimpleRectangle {
    private int width, height;
    public SimpleRectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public int getWidth() {
        return width;
    }//getWidth
    public int  getHeight() {
        return height;
    }//getHeight
    public void setWidth(int width) {
        this.width = width;
    }//setWidth
    public void setHeight(int height) {
        this.height = height;
    }//setHeight
    public void setSize(int width, int height) {
        setWidth(width);
        setHeight(height);
    }//setSize
}//SimpleRectangle
```

Förklaring till felet

Metoden setSize i klassen SimpleRectangle har följande utseende:

```
public void setSize(int width, int height)
{
    setWidth(width);
    setHeight(height);
}//setSize
```

Detta innebär att vår implementation av metoden setSize i klassen ChangeMeasuredRectangle är felaktig!

```
public void setSize(int width, int height)
{
    if (width != getWidth())
        changeCounter++;
    super.setSize(width, height);
}//setSize
```

Varför är metoden felaktig??

Förklaring till felet

Antag att metoden setSize i ChangeMeasuredRectangle anropas med ett värde på parametern width som innebär att bredden på rektangeln förändrats. Då inträffar följande:

Metoden setSize i ChangeMeasuredRectangle ökar instansvariabeln changeCounter med 1 för att markera att bredden har förändrats.

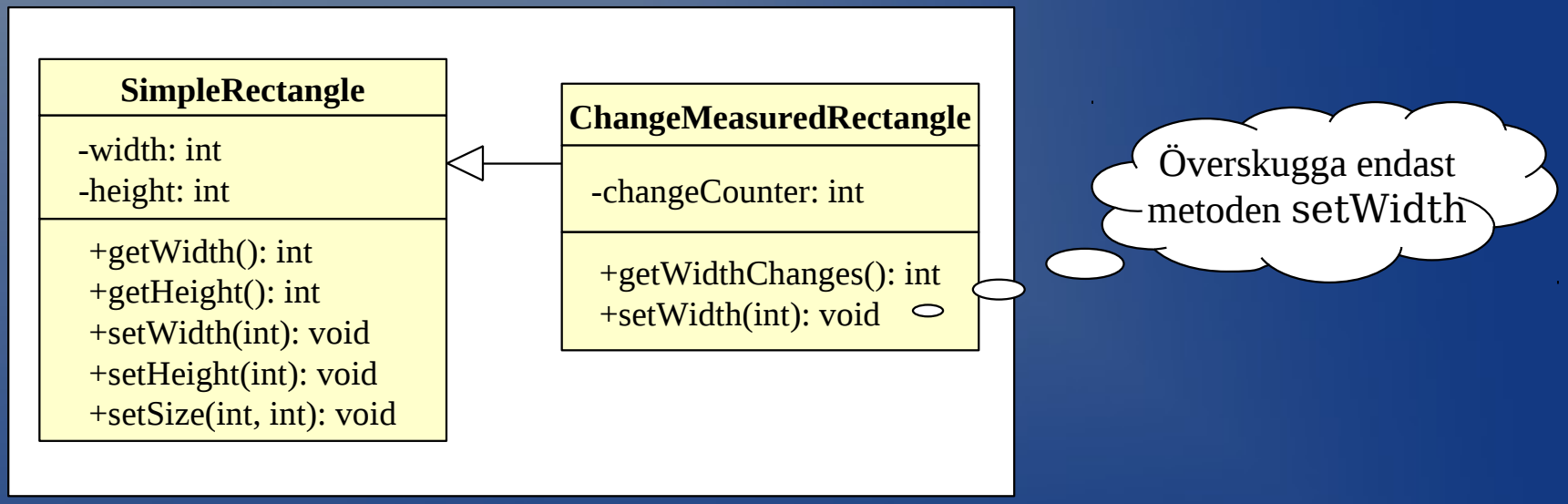
Sedan anropas metoden setSize i SimpleRectangle.

Metoden setSize i SimpleRectangle anropar i sin tur metoden setWidth. Eftersom objektet som anropas har typen ChangeMeasuredRectangel kommer, på grund av *dynamisk bindning*, metoden setWidth i klassen ChangeMeasuredRectangle att anropas.

Metoden setWidth i klassen ChangeMeasuredRectangle konstaterar att bredden har förändrats och ökar instansvariabeln changeCounter med 1.

Korrekt implementering

För den givna implementationen av klassen SimpleRectangle, skall en korrekt implementation av ChangeMeasuredRectangle inte överskugga metoden setSize.



Implementationsarv = stark koppling

Om metoden setSize i klassen SimpleRectangle istället haft följande implementation:

```
public void setSize(int width, int height) {
    this.width = width;
    this.height = height;
}//setSize
```

hade dock vår första implementation av klassen ChangeMeasuredRectangle varit korrekt!

The yo-yo problem:

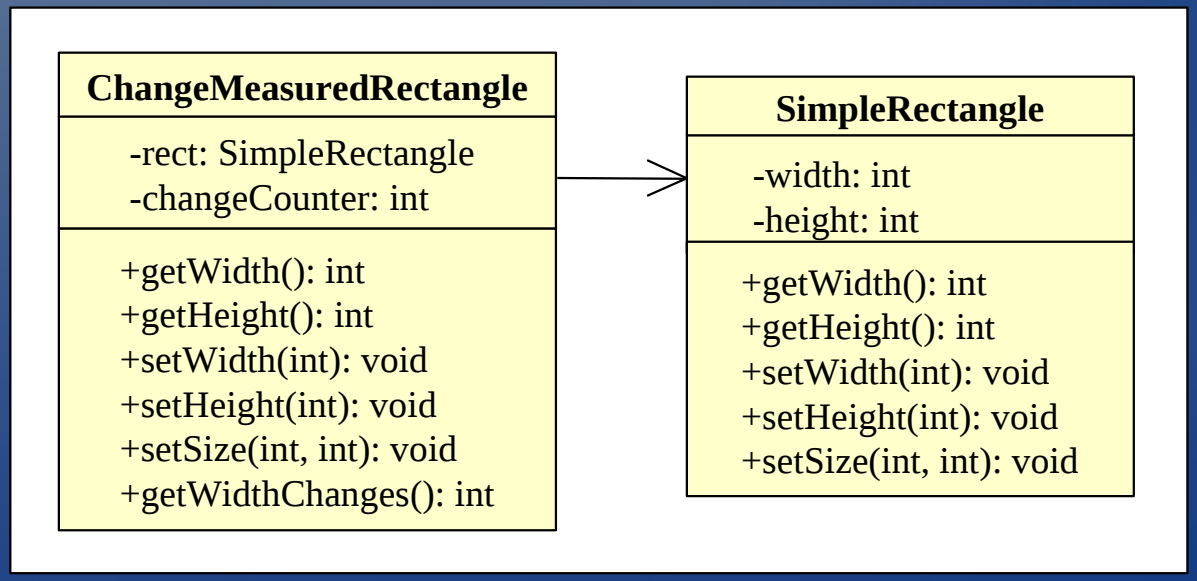
När programutvecklarna måste hoppa fram och tillbaka bland klasserna i arvshierarkin och läsa koden för att förstå hur en förändring av implementationen i en superklass påverkar tillhörande subklasser.

Delegering ett alternativ till implementationsarv

För att kunna göra ett korrekt implementationsarv måste utvecklaren som skriver subklassen känna till implementationsdetaljer i superklassen.

Ofta är delegering en bättre lösning än implementationsarv:

- ger en svagare koppling
- måste användas då implementationen är okänd.



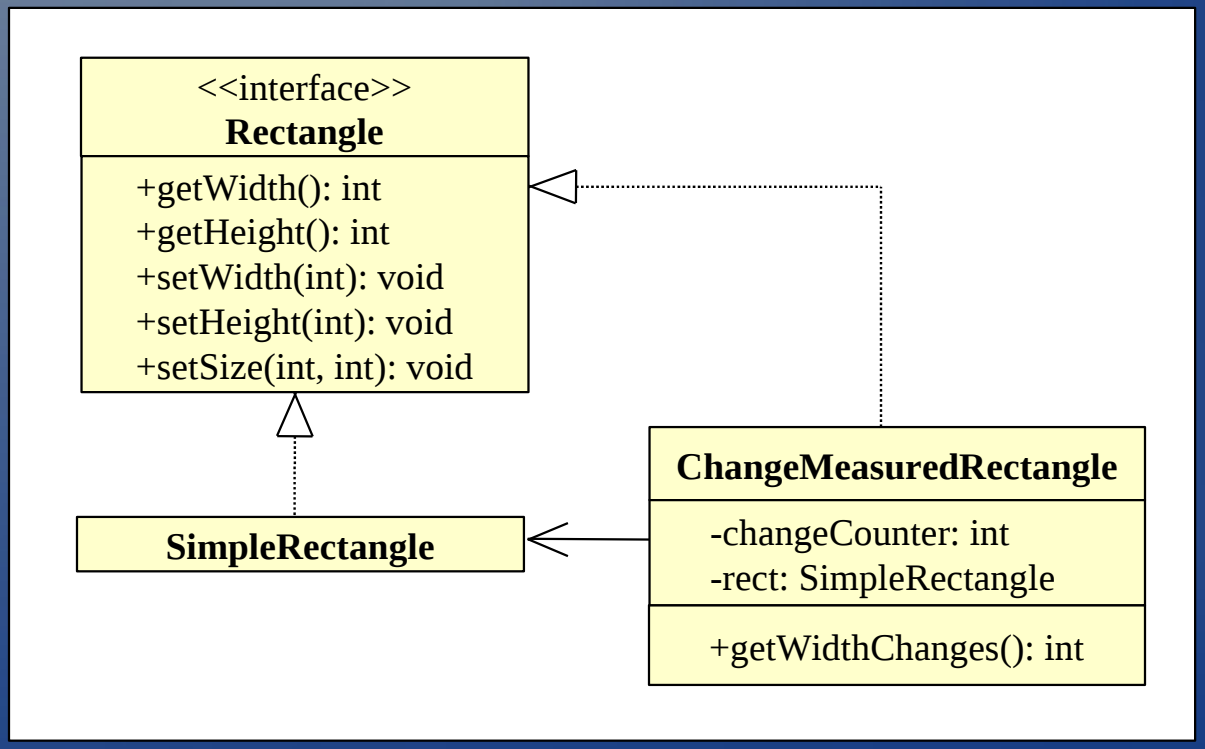
Delegering ett alternativ till implementationsarv

```
public class ChangeMeasuredRectangle {
    private SimpleRectangle rect;
    private int changeCounter = 0;
    public ChangeMeasuredRectangle(int width, int height) {
        rect = new SimpleRectangle(width, height);
    }//constructor
    public int getWidth() {
        return rect.getWidth();
    }//getWidth
    public int  getHeight() {
        return rect.getHeight();
    }//getHeight
    public int getWidthChanges() {
        return changeCounter;
    }//getWidthChanges
```

```
    public void setWidth(int width) {
        if  (width != getWidth())
            ChangeCounter++;
        rect.setWidth(width);
    }//setWidth
    public void setHeight(int height) {
        rect.setHeight(height);
    }//setHeight
    public void setSize(int width, int height) {
        if  (width != getWidth())
            changeCounter++;
        rect.setSize(width, height);
    }//setSize
}//ChangeMeasuredRectangle
```

Designa för förändringar

I en mer förutseende design hade klassen SimpleRectangle implementerat ett gränssnitt Rectangle.



Implementationsarv kontra delegering

Implementationsarv och delegering är olika två tekniker för att åstadkomma återanvändning av kod i enlighet med *The Open-Closed Principle*.

Båda teknikerna har sitt berättingande och det finns situationer där den ena tekniken är att föredra framför den andra.

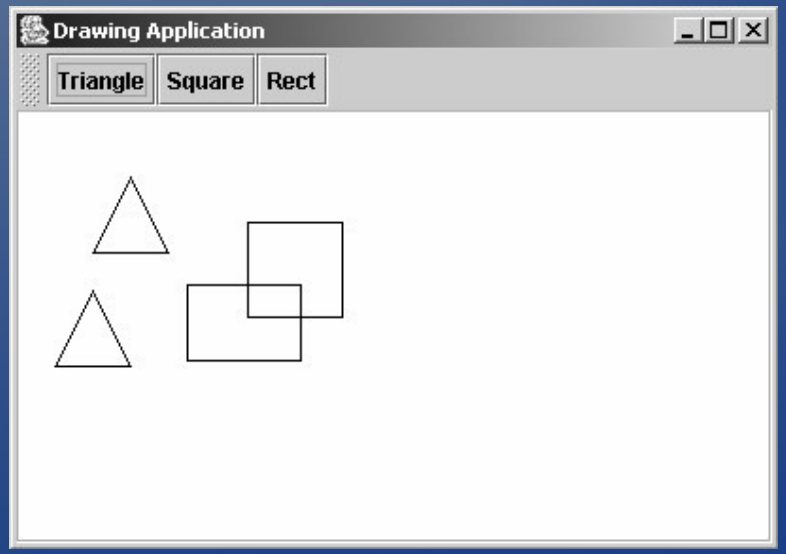
Finns tvekan om vilken teknik som skall användas, skall man använda delegering!

You can choose your friends, but you are stuck with your family.

Exempel: Rita polygon

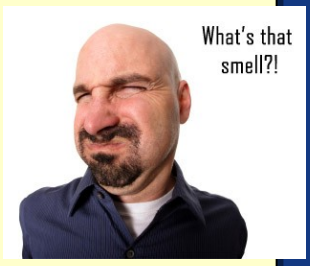
Vi skall i detta exempel utgå från ett program med mycket bristfällig design och stegvis förbättra designen.

Programet är ett enkelt ritprogram för att rita ut polygoner som antingen är kvadrater, retanglar eller trianglar enligt figuren nedan. Vi kommer endast att beakta den del i programmet som gör själva ritandet av polygonen.



Rita polygon: Design 1

```
private ArrayList<String> polygonName;
private ArrayList<Point> centerPoints;
. . .
public void paint(Graphics g) {
   for (int i = 0; i < polygonNames.size(); i++) {
       String currentPolygon = polygonNames.get(i);
       Point currentCenter = centerPoint.get(i);
       if (currentPolygon.equals("square"))
          g.drawRect(currentCenter.x -10, currentCenter.y -10, 20, 20);
       else if (currentPolygon.equals("triangle")) {
          g.drawLine(currentCenter.x, currentCenter.y-10, currentCenter.x-10, currentCenter.y+10);
          g.drawLine(currentCenter.x-10, currentCenter.y+10,
                              currentCenter.x+10, currentCenter.y+10);
          g.drawLine(currentCenter.x+10, currentCenter.y+10, currentCenter.x, currentCenter.y-10);
       }
       else if (currentPolygon.equals("rectangle"))
          g.drawRect(currentCenter.x -20, currentCenter.y -10, 40, 20);
   }
}//paint
. . .
```



Rita polygon: Design 1

- Datat som beskriver en polygon finns på två ställen:
  - typen av polygonen finns i listan `polygonName` i form av en `String`
  - centrumpunkten för polygonen finns i listan `centerPoint` som en `Point`

  Detta innebär att listorna lätt blir osynkroniserade. Tas ett element bort ur en lista måste tillhörande element tas bort i den andra listan.
- Metoden `paint` innehåller alla kod för att rita ut de olika polygonerna.
- Om man vill införa nya typer av polygoner (ovaler, cirklar eller hexagoner) måste implementationen av `paint` förändras.

Designen följer definitivt inte Open-Closed principen!

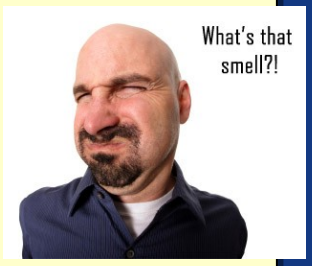
Drawing polygon: Design 2

Sammanför data som beskriver en polygon till en egen klass.

```
import java.awt.Point;
public class PolygonData {
    private String name;
    private Point center;
    public PolygonData(String name, Point center) {
        this.name = name;
        this.center = center;
    }//constructor
    public String getName() {
        return name;
    }//getName
    public Point getCenter() {
        return center;
    }//getPoint
}//PolygonData
```

Drawing polygon: Design 2

```
private ArrayList<PolygonData> polygons;
. . .
public void paint(Graphics g) {
   for (int i = 0; i < polygons.size(); i++) {
       String currentPolygon = polygons.get(i).getName();
       Point currentCenter = polygons.get(i). getCenter();
       if (currentPolygon.equals("square"))
          g.drawRect(currentCenter.x -10, currentCenter.y -10, 20, 20);
       else if (currentPolygon.equals("triangle")) {
          g.drawLine(currentCenter.x, currentCenter.y-10, currentCenter.x-10, currentCenter.y+10);
          g.drawLine(currentCenter.x-10, currentCenter.y+10,
                              currentCenter.x+10, currentCenter.y+10);
          g.drawLine(currentCenter.x+10, currentCenter.y+10, currentCenter.x, currentCenter.y-10);
       }
       else if (currentPolygon.equals("rectangle"))
          g.drawRect(currentCenter.x -20, currentCenter.y -10, 40, 20);
   }
}//paint
. . .
```



Rita polygon: Design 2

Det vi har åstadkommit är att datat för en polygon är samlat på ett ställe och vi slipper därmed problemet med att synkronisera de två listorna.

I övrigt kvarstår bristerna från design 1.

Designen följer definitivt inte Open-Closed principen!

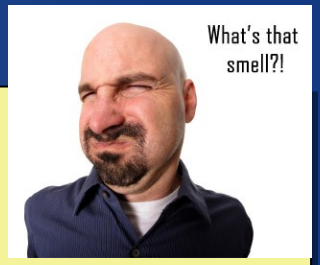
Men nu när vi har en klass `PolygonData` som har all data som berör en polygon borde väl denna klass vara mest lämpad för att rita ut polygonen.

Låt oss göra detta. Samtidigt döper vi om klassen `PolygonData` till `Polygon`, samt inför metoder för att rita ut varje kategori av polygon.

Rita polygon: Design 3

```
import java.awt.Point;
import java.awt.Graphics;
public class Polygon {
    private String name;
    private Point center;
    public Polygon(String name, Point center) {
        this.name = name;
        this.center = center;
    }//constructor
    public String getName() {
        return name;
    }//getName
    public Point getCenter() {
        return center;
    }//getPoint
```

```
    public void draw(Graphics g) {
        if (name.equals("square"))
            drawSquare(g);
        else if (name.equals("triangle"))
            drawTriangle(g);
        else if (name.equals("rectangle"))
            drawRectangle(g);
    }//draw
    private void drawSquare(Graphics g) {...}
    private void drawTriangle(Graphics g) {...}
    private void drawRectangle(Graphics g) {...}
}//Polygon
```



Rita polygon: Design 3

```
private ArrayList<Polygon> polygons = new ArrayList<Polygon>();
. . .
public void paint(Graphics g) {
   for (Polygon poly : polygons) {
       poly.draw(g);
   }
}//paint
. . .
```

Rita polygon: Design 3

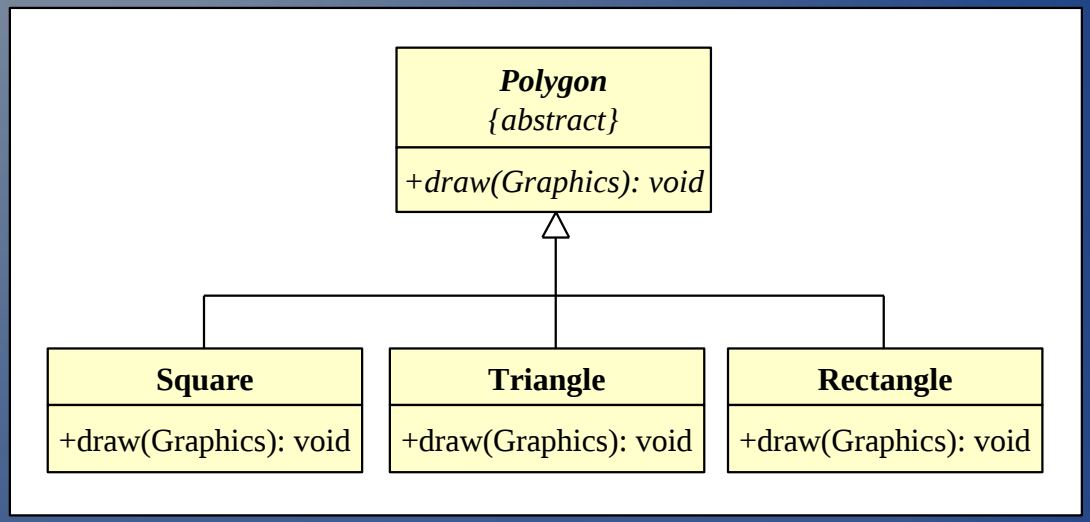
Vi har nu åstadkommit att metoden `paint` följer Open-Closed principen.

Men vi har flyttat problemet till metoden `draw` i klassen `Polygon`. Denna metod bryter definitivt mot OCP!

Om vi tänker efter en stund inser vi att ett objekt av klassen `Polygon` håller reda på vilken ”typ” av polygon det är för att `draw`-metoden skall kunna särskilja denna ”typ” från andra ”typer” av polygon. De olika “typerna” av polygon är naturligtvis polygon, men de är ju också olika varandra. Alla är specialiserade polygon.

De olika ”typerna” av polygon är subtyper till typen `Polygon`.

Rita polygon: Design 4



Rita polygon: Design 4

```
import java.awt.Point;
import java.awt.Graphics;
public abstract class Polygon {
   private Point center;
   public Polygon(Point center) {
      this.center = center;
   }//constructor
   public Point getCenter() {
      return center;
   }//getCenter
   public abstract void  draw(Graphics g);
}//Polygon
```

```
import java.awt.Point;
import java.awt.Graphics;
public class Square extends Polygon {
   public Square(Point center) {
      super(center);
   }//constructor
   public void  draw(Graphics g) {
      g.drawRect(getCenter().x -10, getCenter().y -10, 20, 20);
   }//draw
}//Square
```

Rita polygon: Design 4

```
import java.awt.Point;
import java.awt.Graphics;
public class Triangle extends Polygon {
    public Triangle(Point center) {
        super(center);
    }//constructor
    public void  draw(Graphics g) {
        g.drawLine(getCenter().x, getCenter().y-10, getCenter().x-10, getCenter().y+10);
        g.drawLine(getCenter().x-10, getCenter().y+10, getCenter().x +10, getCenter().y+10);
        g.drawLine(getCenter().x +10, getCenter().y+10, getCenter().x, getCenter().y-10);
    }//draw
}//Triangle
```

```
import java.awt.Point;
import java.awt.Graphics;
public class Rectangle extends Polygon {
    public Rectangle(Point center) {
        super(center);
    }//constructor
    public void  draw(Graphics g) {
        g.drawRect(getCenter().x -20, getCenter().y -10, 40, 20);
    }//draw
}//Rectangle
```

Rita polygon: Design 4

Vi har åstadkommet en design som uppfyller *Open-Closed principen*.

> *Den existerande koden behöver inte ändras när ny funktionalitet läggs till, t.ex. som att kunna rita ovaler, cirklar och hexagon*er.

Vi har gått från en rigid centraliserad design, där all data och funktionalitet var samlad på ett ställe, till en objektorienterad design där data och funktionalitet har distribuerats till ändamålsenliga klasser, och där arv och polymorfism används för att skapa en tydlig, flexibel och utbyggbar struktur.