OOP

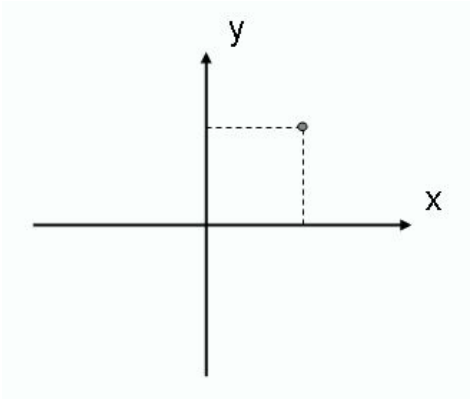
Ereditarietà
Parte 3

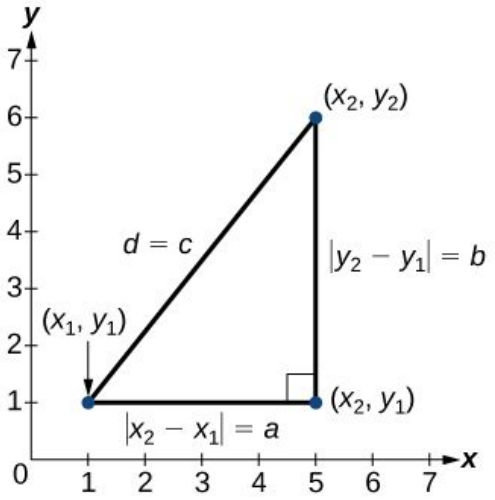
Esercizi

1. Creare la classe **Point2D** caratterizzata dagli attributi X e Y quali coordinate cartesiane del punto. La classe deve possedere il metodo double CalculateDistance(Point2D p) che calcola la distanza tra lo stesso punto e il punto p in input.
2. Impostare una gerarchia di classi per rappresentare le figure geometriche triangolo, quadrato, rettangolo, cerchio. Ogni figura si crea fornendo la lunghezza dei lati. Per ogni tipologia di figura deve essere possibile far calcolare perimetro e area.
   Per i triangoli si vuole anche sapere la tipologia (isoscele, equilatero, scaleno) e se è rettangolo oppure no.
   Per i quadrati e i rettangoli si vuole calcolare la diagonale.
3. I quadrati e i rettangoli possono essere visti come la congiunzione di due triangoli rettangoli lungo le rispettive ipotenuse. Implementare sui quadrati e rettangoli il metodo Riduci() che restituisce tale triangolo.
4. Per il triangolo implementare il metodo Componi() che restituisce la figura risultante dal raddoppio del triangolo stesso.
5. Per i quadrati e rettangoli creare l'overload del costruttore che compone la figura a partire da un triangolo rettangolo.
6. Per i triangoli implementare l'overload del costruttore che compone la figura a partire da un quadrato o da un rettangolo.
7.

Soluzione esercizio Point2D

```
public class Point2D
{
    public double X { get; private set; }
    public double Y { get; private set; }
    public Point2D(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double CalculateDistance(Point2D p)
    {
        double dx = Math.Abs(X - p.X);
        double dy = Math.Abs(Y - p.Y);
        return Math.Sqrt(dx * dx + dy * dy);
    }

    public override bool Equals(object obj)
    {
        if (base.Equals(obj)) return true;
        Point2D p = obj as Point2D;
        if (p == null) return false;

        return p.X == X && p.Y == Y;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(X, Y);
    }

    public override string ToString()
    {
        return string.Format("({0};{1})", X, Y);
    }
}
```





Soluzione esercizio Figure

Static

Abbiamo già visto la parola **static** utilizzata nei progetti console.

Cosa vuol dire static?

In informatica, static individua un **membro statico**, cioè **fisso** e quindi **condiviso** all'interno del software.

In ottica di programmazione ad oggetti, tutto ciò che è static è **condiviso tra tutti gli oggetti** (a meno di limitazioni di visibilità) e non partecipa alla vita degli oggetti.

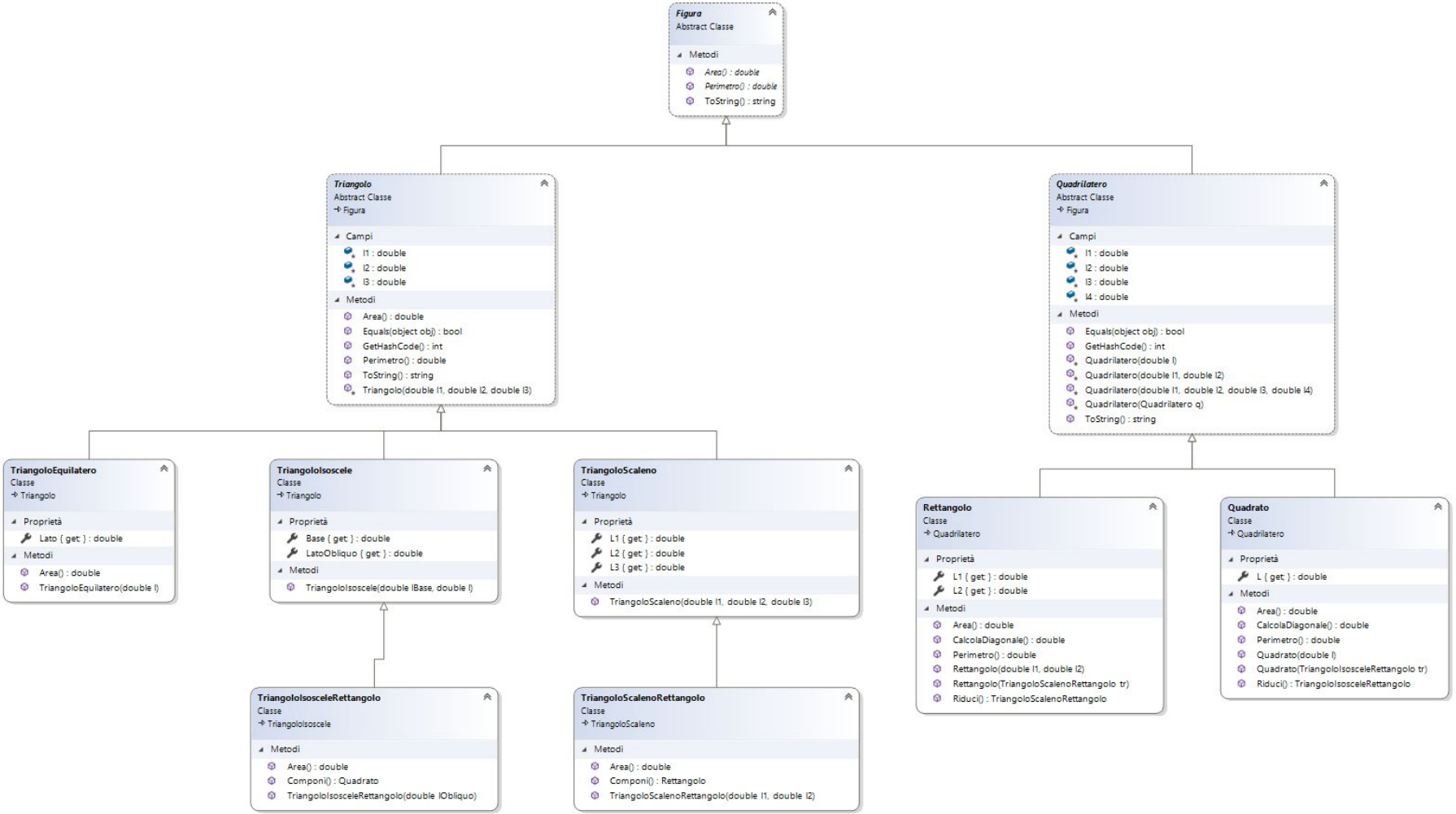
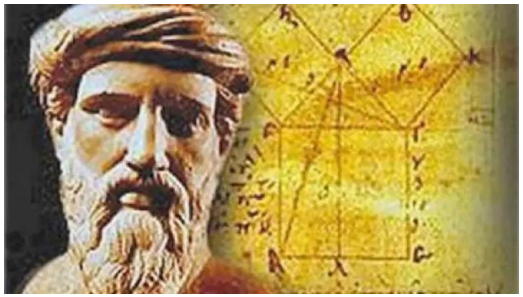


Figura
Abstract Classe
Metodi
Area() : double
Perimetro() : double
ToString() : string
Triangolo
Abstract Classe
Figura
Campi
l1 : double
l2 : double
l3 : double
Metodi
Area() : double
Equals(object obj) : bool
GetHashCode() : int
Perimetro() : double
ToString() : string
Triangolo(double l1, double l2, double l3)
Quadrilatero
Abstract Classe
Figura
Campi
l1 : double
l2 : double
l3 : double
l4 : double
Metodi
Equals(object obj) : bool
GetHashCode() : int
Quadrilatero(double l)
Quadrilatero(double l1, double l2)
Quadrilatero(double l1, double l2, double l3, double l4)
Quadrilatero(Quadrilatero q)
ToString() : string
TriangoloEquilatero
Classe
Triangolo
Proprietà
Lato { get } : double
Metodi
Area() : double
TriangoloEquilatero(double l)
TriangoloIsoscele
Classe
Triangolo
Proprietà
Base { get } : double
LatoObliquo { get } : double
Metodi
TriangoloIsoscele(double lBase, double l)
TriangoloScaleno
Classe
Triangolo
Proprietà
L1 { get } : double
L2 { get } : double
L3 { get } : double
Metodi
TriangoloScaleno(double l1, double l2, double l3)
Rettangolo
Classe
Quadrilatero
Proprietà
L1 { get } : double
L2 { get } : double
Metodi
Area() : double
CalcolaDiagonale() : double
Perimetro() : double
Rettangolo(double l1, double l2)
Rettangolo(TriangoloScaleneoRettangolo tr)
Riduci() : TriangoloScaleneoRettangolo
Quadrato
Classe
Quadrilatero
Proprietà
L { get } : double
Metodi
Area() : double
CalcolaDiagonale() : double
Perimetro() : double
Quadrato(double l)
Quadrato(TriangoloIsosceleRettangolo tr)
Riduci() : TriangoloIsosceleRettangolo
TriangoloIsosceleRettangolo
Classe
TriangoloIsoscele
Metodi
Area() : double
Componi() : Quadrato
TriangoloIsosceleRettangolo(double lObliquo)
TriangoloScalenoRettangolo
Classe
TriangoloScaleno
Metodi
Area() : double
Componi() : Rettangolo
TriangoloScalenoRettangolo(double l1, double l2)

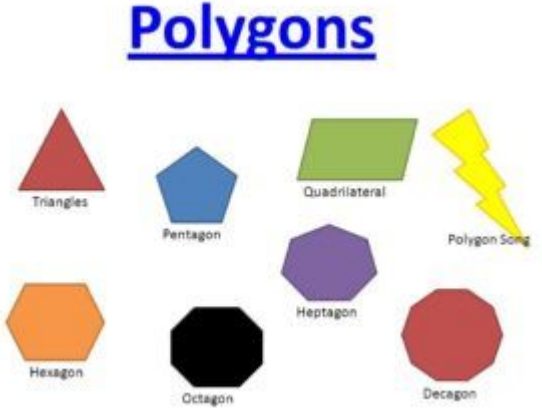
Libreria per triangoli rettangoli

```
public static class Pitagora
{
    public static readonly double SQRT2 = Math.Sqrt(2);

    public static double Ip(double c1, double c2)
    {
        return Math.Sqrt(c1 * c1 + c2 * c2);
    }
}
```

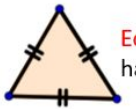


Figura

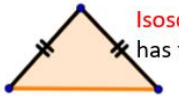
```
public abstract class Figura
{
    public abstract double Area();
    public abstract double Perimetro();
    public override string ToString()
    {
        return this.GetType().Name;
    }
}
```



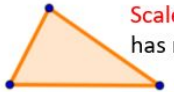
Triangolo



Equilateral Triangle
has three equal sides



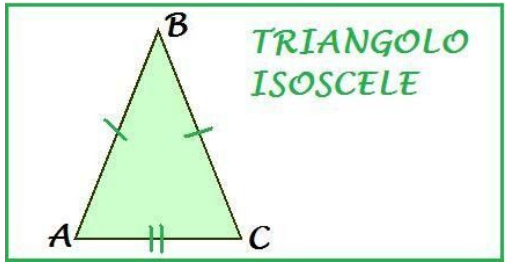
Isosceles Triangle
has two equal sides



Scalene Triangle
has no equal sides

```
public abstract class Triangolo : Figura
{
        protected readonly double l1, l2, l3;

        protected Triangolo(double l1, double l2, double l3)
        {
            this.l1 = l1;
            this.l2 = l2;
            this.l3 = l3;
        }

        public override double Perimetro()
        {
            return l1 + l2 + l3;
        }

        public override double Area()
        {
            double p = Perimetro() / 2;
            return Math.Sqrt(p * (p - l1) * (p - l2) * (p - l3));
        }

        public override string ToString()
        {
            return base.ToString() + string.Format(" ({0} | {1} | {2})", l1, l2, l3);
        }

        public override bool Equals(object obj)
        {
            if (base.Equals(obj)) return true; // uguaglianza referenziale
            Triangolo t = obj as Triangolo;
            if (t == null) return false;

            return t.l1 == l1 && t.l2 == l2 && t.l3 == l3;
        }

        public override int GetHashCode()
        {
            return HashCode.Combine(l1, l2, l3);
        }
}
```

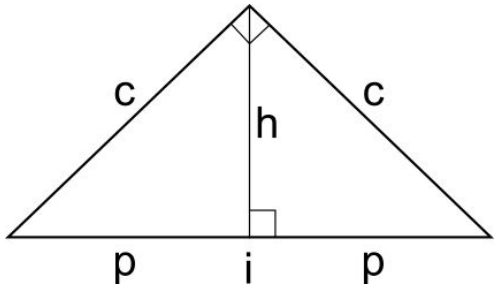
Triangolo isoscele



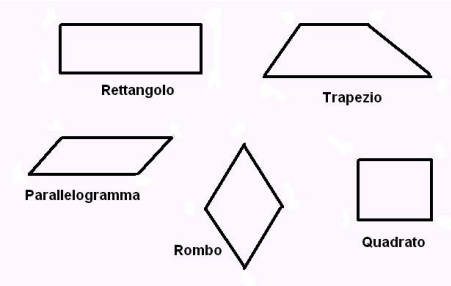

```
public class TriangoloIsoscele : Triangolo
{
    public double LatoObliquo
    {
        get { return l1; }
    }

    public double Base
    {
        get { return l3; }
    }

    public TriangoloIsoscele(double lBase, double l) : base(l, l, lBase)
    {
    }
}
```

Triangolo isoscele rettangolo

```
public class TriangoloIsosceleRettangolo : TriangoloIsoscele
{
    public TriangoloIsosceleRettangolo(double lObliquo) : base(lObliquo * Pitagora.SQRT2, lObliquo)
    {
    }

    public Quadrato Componi()
    {
        return new Quadrato(LatoObliquo);
    }

    public override double Area()
    {
        return Math.Pow(LatoObliquo, 2) / 2;
    }
}
```



Quadrilatero




```
public abstract class Quadrilatero : Figura
{
    protected readonly double l1, l2, l3, l4;

    protected Quadrilatero(double l1, double l2, double l3, double l4)
    {
        this.l1 = l1;
        this.l2 = l2;
        this.l3 = l3;
        this.l4 = l4;
    }

    /// <summary>
    /// Costruttore per rettangoli
    /// </summary>
    protected Quadrilatero(double l1, double l2)
    {
        this.l1 = l1;
        this.l2 = l2;
        this.l3 = l1;
        this.l4 = l2;
    }

    /// <summary>
    /// Costruttore per quadrati
    /// </summary>
    protected Quadrilatero(double l)
    {
        this.l1 = l;
        this.l2 = l;
        this.l3 = l;
        this.l4 = l;
    }

    protected Quadrilatero(Quadrilatero q)
    {
        l1 = q.l1;
        l2 = q.l2;
        l3 = q.l3;
        l4 = q.l4;
    }

    public override string ToString()
    {
        return base.ToString() +
        string.Format(" ({0} | {1} | {2} | {3})", l1, l2, l3, l4);
    }

    public override bool Equals(object obj)
    {
        if (base.Equals(obj)) return true; // uguaglianza referenziale
        Quadrilatero t = obj as Quadrilatero;
        if (t == null) return false;

        return t.l1 == l1 && t.l2 == l2 && t.l3 == l3 && t.l4 == l4;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(l1, l2, l3, l4);
    }
}
```

Rettangolo e quadrato

```
public class Quadrato : Quadrilatero
{
    public double L { get { return l1; } }

    public Quadrato(double l) : base(l)
    {
    }

    public Quadrato(TriangoloIsosceleRettangolo tr) : base(tr.Componi())
    {
    }

    public TriangoloIsosceleRettangolo Riduci()
    {
        return new TriangoloIsosceleRettangolo(L);
    }

    public override double Area()
    {
        return L * L;
    }

    public override double Perimetro()
    {
        return L * 4;
    }

    public double CalcolaDiagonale()
    {
        return L * Pitagora.SQRT2;
    }
}
```

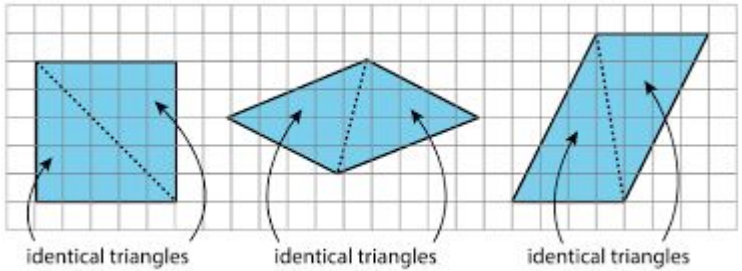
```
public class Rettangolo : Quadrilatero
{
    public double L1 { get { return l1; } }
    public double L2 { get { return l2; } }

    public Rettangolo(double l1, double l2) : base(l1, l2)
    {
    }

    public Rettangolo(TriangoloScalenoRettangolo tr) : base(tr.Componi())
    {

    }

    public override double Area()
    {
        return L1 * L2;
    }

    public override double Perimetro()
    {
        return L1 * 2 + L2 * 2;
    }

    public TriangoloScalenoRettangolo Riduci()
    {
        return new TriangoloScalenoRettangolo(L1, L2);
    }

    public double CalcolaDiagonale()
    {
        return Pitagora.Ip(L1, L2);
    }
}
```

Spunti di evoluzione del software

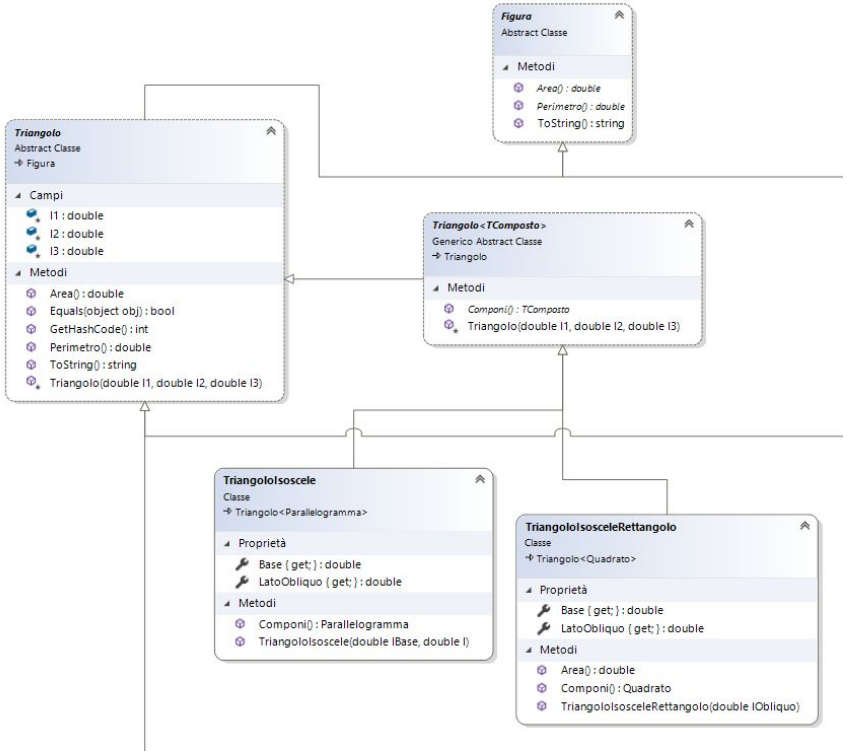
E se tutti i triangoli potessero comporre un quadrilatero?

E se tutti i quadrilateri potessero ridursi ad un triangolo?

Come modifichiamo il software?



Una soluzione: i generics



Si aggiunge una variante della classe Triangolo che utilizza un tipo generico per rappresentare il tipo di ritorno del metodo Componi.

Triangolo con generic

```
public abstract class Triangolo<TComposto> : Triangolo
        where TComposto : Quadrilatero
{
    protected Triangolo(double l1, double l2, double l3) : base(l1, l2, l3)
    {

    }

    public abstract TComposto Componi();
}
```

Con la direttiva **where** si può vincolare un generic a rispettare un tipo minimo di gerarchia.

In questo esempio il TComposto deve essere almeno un Quadrilatero: quindi potrà essere un Quadrilatero, un Quadrato, un Rettangolo, e così via.

Triangolo isoscele con generic

```
public class TriangoloIsoscele : Triangolo<Parallelogramma>
{
    public double LatoObliquo
    {
        get { return l1; }
    }

    public double Base
    {
        get { return l3; }
    }

    public TriangoloIsoscele(double lBase, double l) : base(l, l, lBase)
    {
    }

    public override Parallelogramma Componi()
    {
        return new Parallelogramma(LatoObliquo, Base);
    }
}
```

Il triangolo isoscele eredita da **Triangolo<Parallelogramma>**, in questo modo acquisisce il metodo **Componi** applicato ai parallelogrammi.

È la classe derivata (utilizzatore) che decide il tipo generic, come voi quando create una List<T> decidete il tipo T della lista.

Triangolo isoscele rettangolo con generic

```
public class TriangoloIsosceleRettangolo : Triangolo<Quadrato>
{
    public double LatoObliquo
    {
        get { return l1; }
    }

    public double Base
    {
        get { return l3; }
    }

    public TriangoloIsosceleRettangolo(double lObliquo)
                : base(lObliquo, lObliquo, lObliquo * Pitagora.SQRT2)
    {

    }

    public override Quadrato Componi()
    {
        return new Quadrato(LatoObliquo);
    }

    public override double Area()
    {
        return Math.Pow(LatoObliquo, 2) / 2;
    }
}
```

Ora il TriangoloIsosceleRettangolo deve ereditare da Triangolo specificando il generic Quadrato per avere il metodo Componi.

Vantaggi e svantaggi di questa modifica

VANTAGGI

- Il metodo Componi è ora acquisito da tutti i triangoli in modo formale avendolo inserito nella classe base.
- Questo meccanismo rende il codice più robusto a possibili errori di programmazione.

SVANTAGGI

- Abbiamo dovuto "rompere" parte della catena di ereditarietà creata in precedenza.
- Il triangolo rettangolo isoscele non eredita più dal triangolo rettangolo: questo è brutto dal punto di vista logico e introduce duplicazione di codice per le proprietà LatoObliquio e Base.

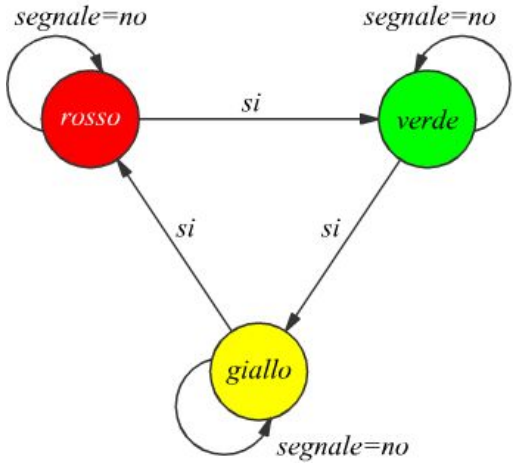
Si può fare di meglio? Certo!

Oggetto e stato interno

Gli oggetti posseggono uno stato interno, una configurazione. Anche noi possediamo uno stato.

Un oggetto può essere utilizzato come un automa a stati finiti (ASF*).

Ad esempio, modelliamo un semaforo.



*https://it.wikipedia.org/wiki/Automa_a_stati_finiti

Esercizi

1. Utilizzando il Timer e la programmazione ad oggetti, creare una WPF che simula un semaforo. Ogni 5 secondi il semaforo cambia stato (rosso->verde->giallo).
2. Vostra madre vi chiede un programma per gestire e memorizzare la lista della spesa.
   Per ogni elemento della lista la mamma vuole vedere il nome e la quantità (es. banane:3). La mamma vuole anche che la lista sia ordinata per priorità (bassa, media, alta). Ovviamente la mamma vuole poter chiudere il programma e riaprirlo il giorno dopo trovando la sua lista salvata!
3. Vostro nonno si dimentica le cose: vi chiede di fargli un programma dove può appuntarsi le cose che deve ricordare. Le note da memorizzare devono essere ordinate per data di inserimento, dalla più nuova alla più vecchia. Le note si devono poter modificare e cancellare. All'apertura il programma deve mostrare le note salvate.
4. Successivamente il nonno chiede di poter salvare, oltre alle note, anche degli avvisi (es. fai l'autolettura del contatore dell'acqua). Questi avvisi devono mostrare una data di scadenza. All'apertura del programma, gli avvisi scaduti devono mostrare un alert grafico.