Galton Machine

Implementazione

Titolo del progetto: Galton Machine
Alunno/a: Cristiano Colangelo
Classe: Info I4AC
Anno scolastico: 2017/2018
Docente responsabile: Ugo Bernasconi

Indice

Tabella delle figure

1 Implementazione

1.1 Griglia e coordinate

La simulazione è contenuta in una griglia definita come **array bidimensionale di tipo** `Ball`**.**

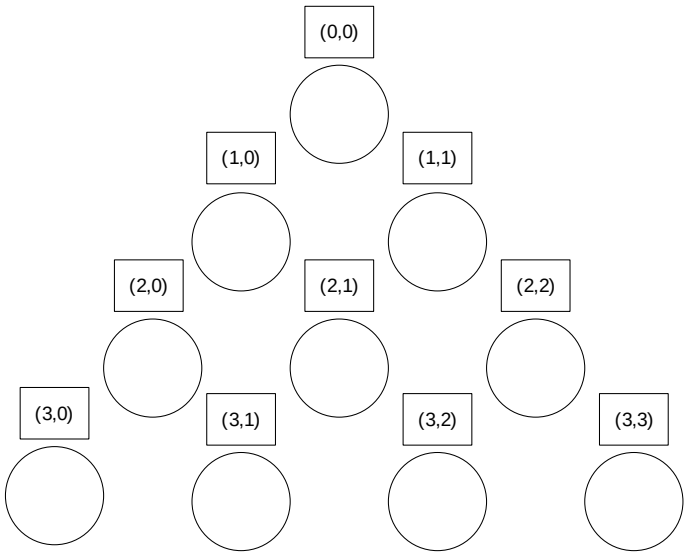


Figura 1 Rappresentazione grafica effettivata della griglia

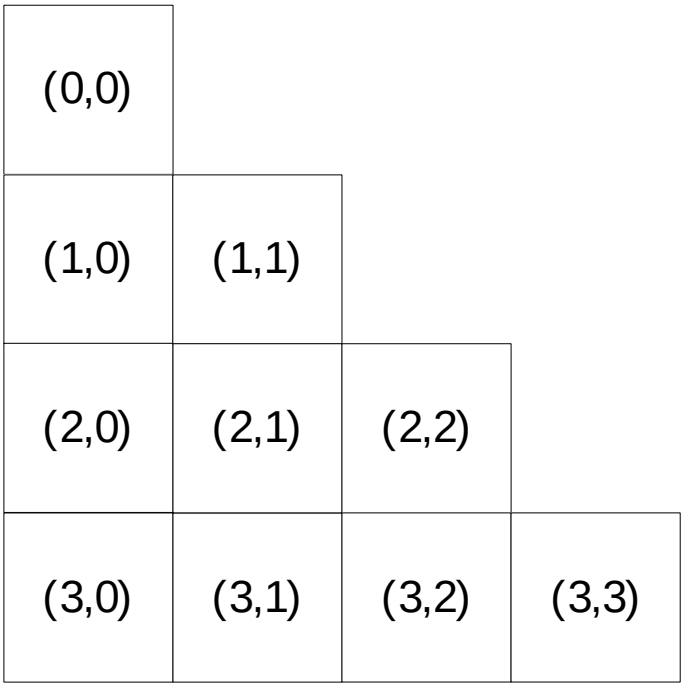


Figura 2 Rappresentazione della logica dell'array griglia

La prima immagine mostra una rappresentazione piuttosto fedele del modello grafico reale, ossia quello che viene mostrato all'utente nell'interfaccia grafica, la seconda rappresentazione invece ritrae la stessa struttura dati riadattata alla realtà, ovvero al modello di array bidimensionale.

Le coordinate mostrate sono le posizioni possibili nella quale la pallina potrà andare a posizionarsi. Ogni volta che la pallina scenderà di 1, la coordinata y incrementerà di 1, mentre la coordinata x incrementerà in modo casuale in un intervallo [0; 1]. Si può quindi affermare che la pallina andrà *a sinistra* quando la coordinata x non incrementerà, mentre andrà verso *destra* quando incrementerà.

Alla fine di un ciclo, quando la pallina avrà raggiunto la fine del suo percorso, verrà incrementato il conteggio della simulazione e riposizionata la pallina in cima alle stecche.

Di seguito è mostrato un diagramma di flusso che descrive il codice inerente la simulazione della pallina (solo lo spostamento/animazione della pallina).

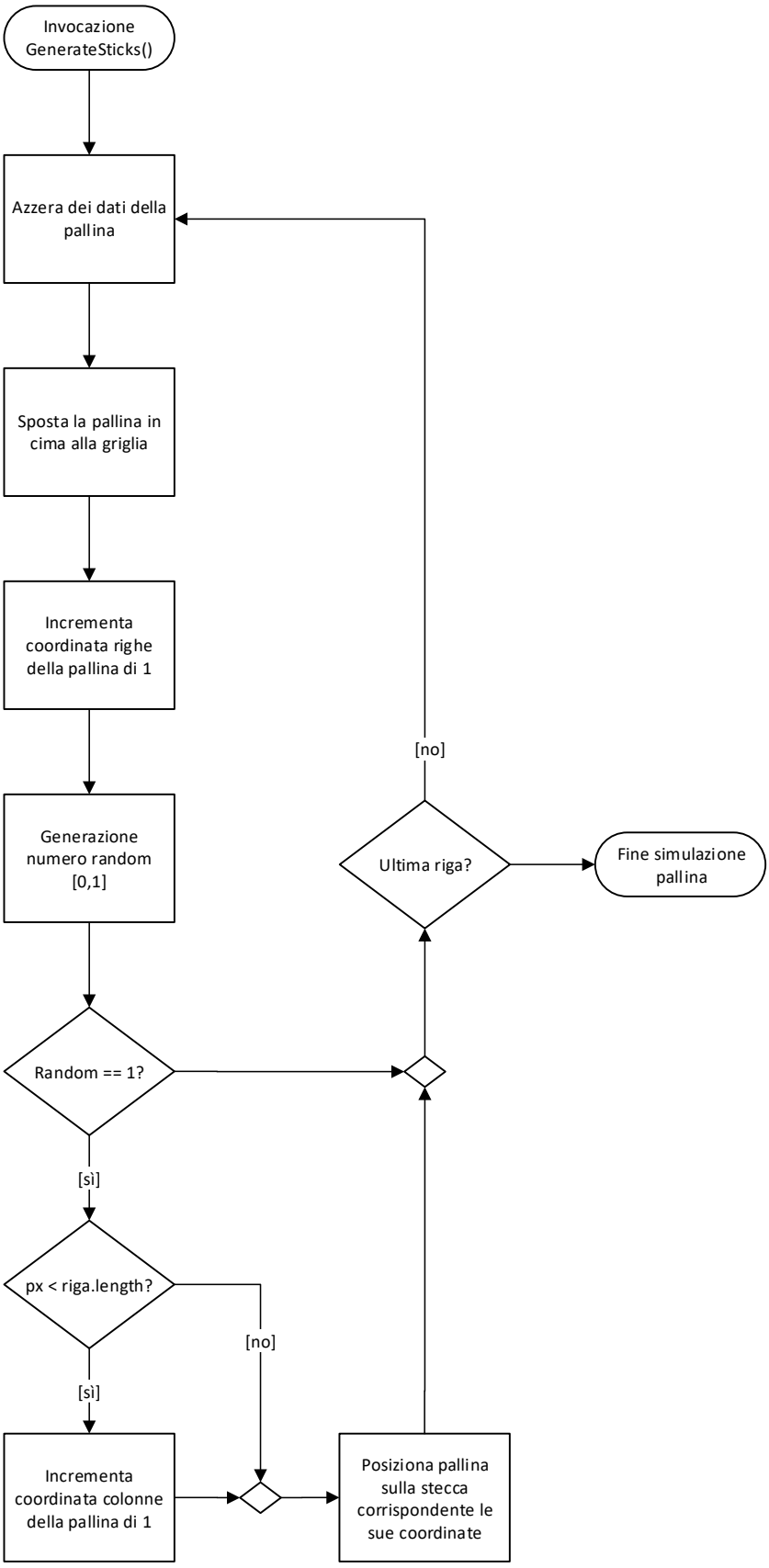


Figura 3 Diagramma di flusso movimento pallina

1.2 Prototipo console

Come primo prototipo ho realizzato un progetto Windows Console prendendo come base il diagramma di flusso (vedere *Figura 3*) che ho disegnato. Ho creato 3 semplici classi `QuincunxGrid`, `Ball` e `GaltonMachine` che mi permettessero di simulare la macchina di Galton, successivamente ho provveduto a stampare dei caratteri che rappresentassero la pallina e le stecche, in più ho fatto in modo che ogni volta che la pallina "rimbalzasse" su una stecca, incrementasse un numero in modo da indicare quali stecche siano state più sollecitate.

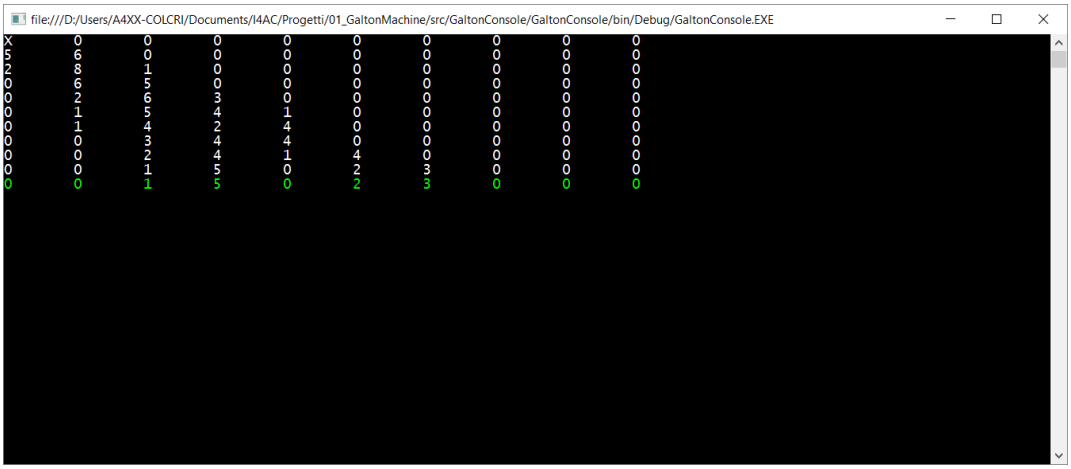


Figura 3 Prototipo console

1.3 Simulazione pallina/stecche

La rappresentazione delle palline e delle stecche è implementata grazie alla classe `Ball`. Inizialmente avevo previsto due classi separate, `Ball` e `Stick`, per indicare la pallina cadente e le stecche. Successivamente, notando che Ball era la stessa cosa di `Stick` ma contenente le coordiante della griglia, ho fatto ereditare `Ball`→`Stick`. Andando avanti con il refactoring verso la fine del progetto, ho avuto bisogno di avere le coordiante anche per `Stick`, in modo da poter implementare il metodo pubblico `void GetStick(int row, int column)`, quindi non mi è rimasto altro che eliminare `Stick` e utilizzare `Ball` per tutte le occorrenze. Ciò ha senza dubbio alleggerito il codice dato che mi ha permesso di eliminare una classe intera e non necessaria alla finalità del prodotto.

1.3.1 Generazione stecche

La generazione delle stecche è messa in funzione grazie al metodo privato della classe GaltonSimulation `void GenerateSticks()`. Il metodo viene chiamato a ogni istanza della simulazione e a ogni cambiamento della grandezza della base. Il metodo contiene due `for` imbricati che generano man mano le coordinate delle stecche. Una volta generata la coppia di coordinate, viene istanziato un oggetto di tipo `Ball` che viene inserito in una `ObservableCollection` dopo l'assegnazione delle coordinate `X` e `Y` alle sue omonime proprietà. Inizialmente avevo messo questo metodo all'interno del ViewModel, ma in seguito a un refactoring generale verso la fine del progetto, ho notato che il ViewModel era diventato una "god class" (antipattern) quindi ho spostato il metodo nella classe `GaltonSimulation`.

Di seguito è illustrata la prima versione del codice di `GenerateSticks()`.

```
private void GenerateSticks()
{
     // Grandezza della base
     double n = model.Grid.GetSize();

     // Distanza fra le stecche
     double dx = (CanvasWidth - STICKS_DIAMETER) / (n - 1);
     double dy = (CanvasHeight - STICKS_DIAMETER) / (n - 1);

     // Coordinate x y delle stecche iniziali
     double x = 0;
     double y = CanvasHeight - STICKS_DIAMETER;

     for (int i = 0; i < n; i++)
     {
        for (int j = 0; j < n - i; j++)
        {
            Ball currentBall = new Ball(x, y, STICKS_DIAMETER);
            SticksList.Add(currentBall);
            x += dx;
        }
        x = dx / 2 * (i + 1);
        y -= dy;

    }
}
```

Una volta implementato questo codice, mi sono reso conto che sarebbe stato necessario riordinare `SticksList` in ordine inverso per poter copiare gli elementi nell'ordine corretto (→ vedi Figura 1). L'ordinamento precedente delle stecche era infatti dal basso verso l'alto. Il riordinamento sarebbe stato un passaggio inutile, ho quindi deciso di riscrivere l'algoritmo (illustrato in seguito) in modo da generare le stecche partendo direttamente dall'alto.

Siccome è necessario avere i nodi contenuti in una `ObservableCollection` per il binding con `ItemsControl`, è stato necessario traslare le stecche dalla lista (lineare) alla griglia (array bidimensionale). Questo è stato un passaggio inutile, corretto in seguito.

In basso è illustrato cosa si intende per traslazione.

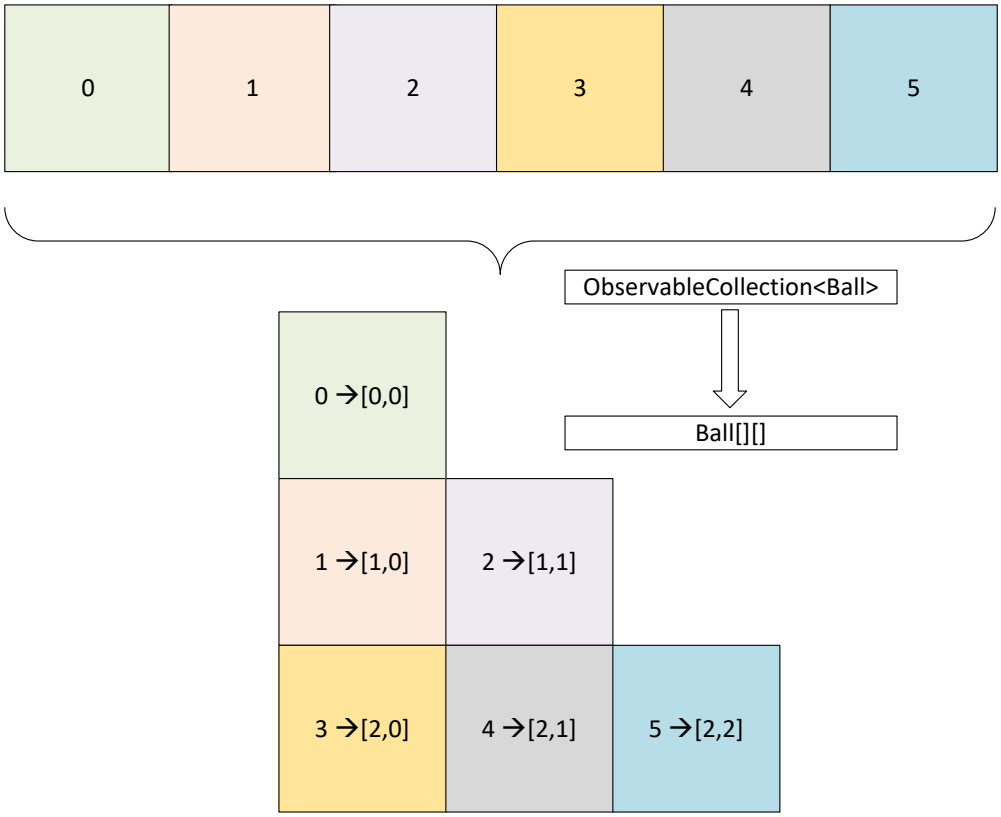


Figura 4 Traslazione da lista lineare a array bidimensionale con indici

Di seguito è illustrato il secondo codice di `GenerateSticks()`.

```
private void GenerateSticks()
{
    // Offset orizzontale e verticale
    double hoffset = 50;
    double voffset = 50;

    // Larghezza e altezza canvas
    double cw = CanvasWidth - hoffset;
    double ch = CanvasHeight - voffset;

    // Grandezza della base
    double n = model.Grid.GetSize();

    // Distanza fra le stecche
    double dx = (cw - STICKS_DIAMETER) / (n - 1) / 2;
    double dy = (ch - STICKS_DIAMETER) / (n - 1);

    // Coordinate x y delle stecche iniziali
    double x = CanvasWidth / 2 - STICKS_DIAMETER;
    double y = 0;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i + 1; j++)
        {
            // Se è la prima stecca
            if (j == 0)
            {
                x = dx * (n - i - 1);
            }
            else
            {
                x += dx * 2;
            }

            SticksList.Add(new Ball(x + (hoffset / 2), y + (voffset / 2), STICKS_DIAMETER));

        }
        y += dy;
    }
}

// Inserisce gli elementi di SticksList nel model.Grid
int index = 0;
for (int i = 0; i < model.Grid.Size; i++)
{
    for (int j = 0; j < model.Grid.GetRowSize(i); j++)
    {
        model.Grid.SetCell(i, j, SticksList.ElementAt(index));
        index++;
    }
}
```

Traslazione da array 2d a collection lineare

Per ovviare a questo “problema” ho eliminato completamente la classe `QuincunxGrid`. Essa creava inutili complicazioni, perciò ho semplicemente incluso la `ObservableCollection` contenente le stecche nella classe `GaltonSimulation` (il nostro model). Avrei anche potuto separare la logica dalla grafica, ma basandomi anche sui desideri del committente ho preferito tenere semplice il tutto, anche perchè avrei dovuto reimplementare il pattern observable nella mia griglia per poterla usare nel codice di formattazione della GUI (XAML), e avrebbe comportato una spesa di tempo infine non necessaria.

Di seguito illustrato il codice finale, dove anche `GenerateSticks()` è stato spostato nella classe `GaltonSimulation`.

```
private void GenerateSticks()
{
    Sticks.Clear();

    // Larghezza e altezza canvas
    double cw = GDeviceSize.Width - HorizontalOffset;
    double ch = GDeviceSize.Height - VerticalOffset;

    // Distanza fra le stecche
    double dx = (cw - SticksDiameter) / (SimulationSize - 1) / 2;
    double dy = (ch - SticksDiameter) / (SimulationSize - 1);

    // Coordinate x y delle stecche iniziali
    double x = GDeviceSize.Width / 2 - SticksDiameter;
    double y = 0;

    for (int i = 0; i < SimulationSize; i++)
    {
        for (int j = 0; j < i + 1; j++)
        {
            // Se non è la prima stecca
            if (j != 0)
            {
                x += dx * 2;

            }
            else
            {
                x = dx * (SimulationSize - i - 1);
            }

            Sticks.Add(new Ball(i, j,
              x + (HorizontalOffset / 2), y + (VerticalOffset / 2),
              SticksDiameter, SticksColor));
        }
        y += dy;
    }
}
```

1.3.2 Disegno delle stecche

`SimulationItemsCollection` è una `ObservableCollection` tipizzata `Ball` connessa con un binding al codice XAML tramite il tag `ItemsControl`, che permette di aggiungere dinamicamente degli elementi alla view semplicemente aggiungendoli alla lista. Essa è passata grazie a una `StaticResource` dichiarata all'inizio del codice XAML, questo perchè essendo che `SimulationItemsCollection` è una `CompositeCollection`, XAML necessita di una risorsa statica. Lo scopo delle `CompositeCollection` è quello di poter riunire sotto un unico oggetto più `ObservableCollection`. Inizialmente ho creato due `Canvas` separate per disegnare sia le stecche che la pallina cadente, una che disegnava le stecche e una che disegnava la pallina.

Questo si è dimostrato un workaround e non una vera e propria soluzione, per questo motivo sono riuscito a trovare la seguente soluzione.

```
<UserControl.Resources>
    <CollectionViewSource x:Key="SimulationItemsCollection"
       Source="{Binding Path=SimulationItemsCollection, Mode=OneTime}"/>
    <CollectionViewSource x:Key="ChartItemsCollection"
       Source="{Binding Path=ChartItemsCollection, Mode=OneTime}"/>
</UserControl.Resources>

[...]

<!-- Stecche -->
<ItemsControl Panel.ZIndex="1">
    <ItemsControl.ItemsSource>
        <CompositeCollection>
            <CollectionContainer
              Collection="{Binding Source={StaticResource SimulationItemsCollection}}"/>
            <Ellipse Canvas.Top="{Binding Path=FallingBall.Y}"
              Canvas.Left="{Binding Path=FallingBall.X}"
              Width="{Binding Path=BALL_DIAMETER}" Height="{Binding Path=BALL_DIAMETER}"
              Fill="Black" />
        </CompositeCollection>
    </ItemsControl.ItemsSource>
<ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
        <Canvas Focusable="False" Width="{Binding Path=CanvasWidth}"
          Height="{Binding Path=CanvasHeight}"/>
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
    <ItemsControl.ItemContainerStyle>
        <Style>
            <Setter Property="Canvas.Left" Value="{Binding Path=X}"/>
            <Setter Property="Canvas.Top" Value="{Binding Path=Y}"/>
        </Style>
    </ItemsControl.ItemContainerStyle>
    <ItemsControl.Resources>
        <DataTemplate DataType="{x:Type model:Ball}">
            <Ellipse Width="{Binding Path=Diameter}" Height="{Binding Path=Diameter}"
              Fill="{Binding Path=Color}"/>
        </DataTemplate>
    </ItemsControl.Resources>
</ItemsControl>
```

Utilizzando `Canvas` come contenitore dell'area di disegno, è stato necessario anche definire un `ItemsPanel`. Essendo che gli oggetti aggiunti al canvas ereditano alcune proprietà, come `Canvas.Top` e `Canvas.Left`, è stato necessario definire un `ItemContainerStyle` e applicare uno stile attraverso alcuni `Setter` che aggiungono, sempre grazie al binding, gli attributi `Canvas.Left` e `Canvas.Top` agli `Ellipse` generati dinamicamente. Più in particolare, i Setter generano un binding fra gli attributi `X` e `Y` e le proprietà `Canvas.Left` e `Canvas.Top`.

Per seguire appieno la filosofia del pattern MVVM che prevede la separazione fra la cosiddetta business logic e interfaccia utente, si utilizza un `DataTemplate` che ha l'utilità di sfruttare l'astrazione di una classe per collegarla con binding all'oggetto vero e proprio. Si noti infatti che `SticksList` contiene degli oggetti di tipo `Ball` (vedi UML delle classi) e non `Ellipse`. In pratica, le proprietà `Width` e `Height` di `Ellipse` sono collegate alla proprietà `Radius` di `Ball` (si utilizza `Radius` in quanto la rappresentazione grafica è di un cerchio che avrà sempre larghezza e altezza uguali) in modo concettualmente analogo a come spiegato nel paragrafo precedente per quanto concerne la posizione delle stecche

In basso è rappresentato uno schema che spiega visivamente la disposizione dei binding/setter in relazione ai vari attributi.

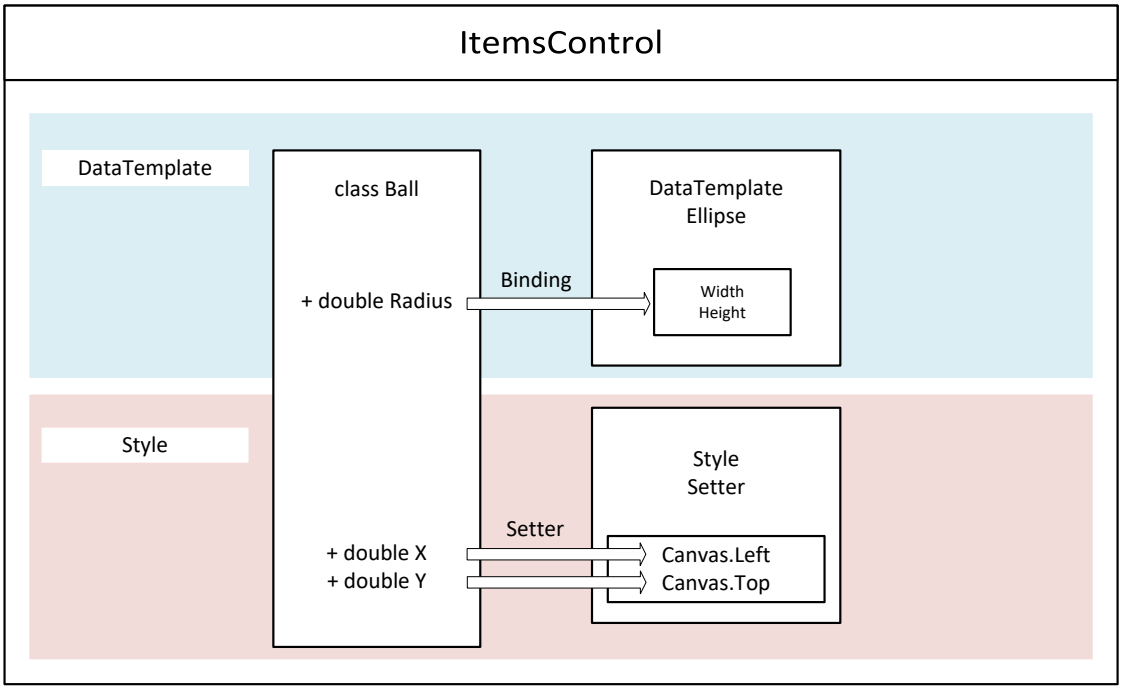


Figura 5 Schema ItemsControl

1.3.3 Animazione pallina

La pallina è animata dal metodo privato `AnimateFallingBall()`. Ho testato anche l'utilizzo della classe `DispatcherTimer` ma essa non si è rivelata adatta al lavoro in quanto il metodo `Timer_Tick` viene chiamato ricorrentemente ogni *n* millisecondi indipendentemente dal codice contenuto nel metodo. Per questo motivo ho optato per l'utilizzo di un semplice thread aggiuntivo interno al model-view.

Il thread viene istanziato e avviato nel costruttore del model-view.

```
Thread ballAnimationThread = new Thread(new ThreadStart(AnimateFallingBall));
ballAnimationThread.Start();
```

Inizialmente ho creato un piccolo algoritmo per animare la pallina che prendeva i riferimenti delle stecche da `SticksList` e disegnava su quel riferimento, di seguito è illustrato come funzionava.

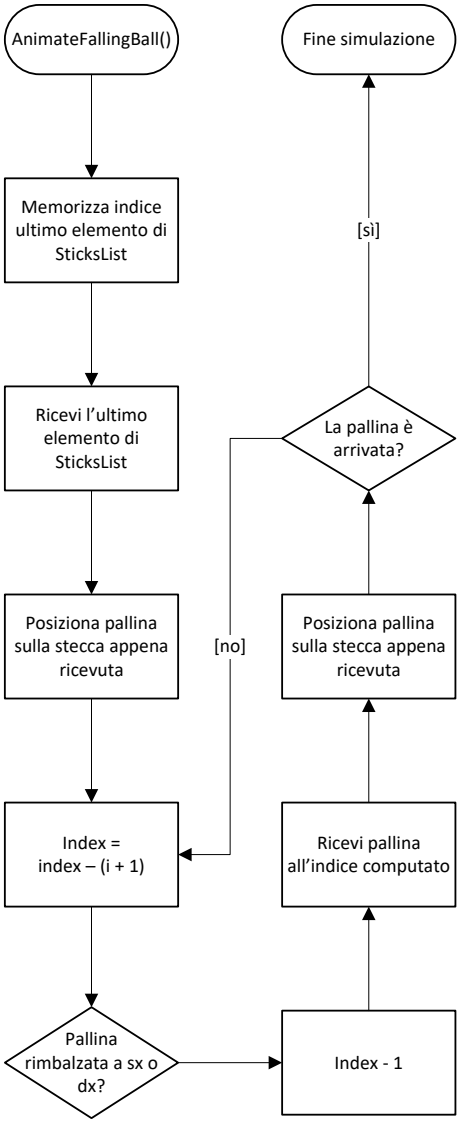


Figura 6 Diagramma di flusso animazione pallina prototipo iniziale

```
private void AnimateFallingBall()
{
    int index = SticksList.Count - 1;
    Ball currentBall = SticksList.ElementAt(index);

    PlaceBallOnStick(currentBall);

    for (int i = 0; i < model.Grid.GetSize() - 1; i++)
    {
        Thread.Sleep(ANIMATION_SPEED);
        index = index - i - 1;

        if (!model.Ball.Bounce())
        {
            index--;
        }
        currentBall = SticksList.ElementAt(index);

        PlaceBallOnStick(currentBall);
    }
}
```

Andando avanti con il design ho cambiato metodo utilizzando la soluzione originariamente impiegata per il prototipo GaltonConsole, lavorando sulla griglia (vedi Figura 3).

Le proprietà `BallRow` e `BallColumn` del model contengono la posizione nella griglia della pallina, grazie al metodo `PlaceBallOnStick()` la pallina viene posizionata sulla stecca corrispondente.

Di seguito è illustrato il codice relativo all'animazione della pallina e allo svolgimento della simulazione.

```
private void AnimateFallingBall()
{
    for (int i = 0; i < SIMULATION_LENGTH; i++) {
        // Piazza la pallina sulla prima stecca
        PlaceBallOnTopStick();

        for (int j = 0; j < model.Grid.GetSize() - 1; j++)
        {
            Thread.Sleep(ANIMATION_SPEED);

            model.BallRow++;

            // Fa rimbalzare la pallina se la pallina non cade fuori dalla riga
            if (model.Ball.Bounce() && model.BallColumn < model.Grid.GetRowSize(j))
            {
                model.BallColumn++;
            }

            // Riceve la stecca su cui posizionare la pallina
            PlaceBallOnStick(model.Grid.GetCell(model.BallRow, model.BallColumn));
        }
        Thread.Sleep(ANIMATION_SPEED);
    }
}
```

Anche qui ci sono state complicazioni inutili, in particolare la griglia (che come detto in precedenza, ho eliminato in favore dell'utilizzo diretto della `ObservableCollection` contenente le palline), e l'appartenenza del metodo `PlaceBallOnTopStick()` al ViewModel, il quale doveva appartenere al model `GaltonSimulation`.

Di seguito è illustrato il codice definitivo dell'animazione della pallina. Notare che `FallingBall` appartiene comunque alla classe `GaltonSimulation` (→ `GaltonSim` nel codice), ma ho creato una proprietà che ritorna la referenza dalla suddetta classe per accorciare il codice siccome si accede spesso alla pallina. `FallingBall` appartiene dalla classe `GaltonSimulation`.

Si noti anche come il metodo `Ball.Bounce()` sia stato eliminato in quanto il rimbalzo non genera nessun cambiamento interno della classe `Ball`, dunque si noti come sia stato un metodo effettivamente inutile e forse fuorviante per chi avrebbe potuto leggere il codice. Al suo posto è stato istanziato un oggetto `Random`.

```
private void AnimateFallingBall()
{
    try
    {
        for (int i = 0; i < SimulationLength - 1; i++)
        {
            // Piazza la pallina in cima
            FallingBall.Reset();
            GaltonSim.PlaceBallOnTopStick();

            // Simulazione caduta pallina
            for (int j = 0; j < SimulationSize - 1; j++)
            {
                Thread.Sleep(SimulationSpeed);

                FallingBall.Row++;

                Random rnd = new Random();
                if (rnd.Next(0, 2) == 1 && FallingBall.Column < j + 1)
                {
                    FallingBall.Column++;
                }

                GaltonSim.PlaceBallOnStick(GaltonSim.GetStick(FallingBall.Row,
                 FallingBall.Column));
            }
            [...]
        }
        [...]
    }
    catch (ThreadInterruptedException)
    {
        return;
    }
}
```

1.4 Grafico

Il grafico è formato da 3 elementi, gli istogrammi, la curva e le etichette. Questi 3 elementi sono rappresentati dalle classi `Histogram`, `BellCurve` e `ChartLabel`. Questi elementi sono gestiti centralmente grazie alla classe `DistributionChart`. Inizialmente non utilizzavo una classe centrale, e ciò era molto sconveniente in quando dovevo manualmente aggiornare ogni cosa singolarmente. Grazie a `DistributionChart`, ho potuto avvalermi di un'interfaccia centrale che mi permettesse di agire soltanto su pochi membri pubblici e lasciar propagare i cambiamenti alla suddetta classe.

1.4.1 Generazione grafico

Gli istogrammi, la curva e le labels vengono generate grazie al metodo privato `void GenerateChart()`, che viene invocato a ogni istanza di `DistributionChart` o al cambiamento della grandezza della base. All'istanza di `DistributionChart` è necessario specificare alcuni parametri grafici, in particolare si notino i parametri specificanti le dimensioni del device grafico e l'offset verticale (definito come `VerticalAnchor`) e la grandezza del device grafico (in poche parole, il nostro `Canvas`).

Di seguito è illustrato il codice del metodo sopracitato.

```
if (Size > 0)
{
    Histograms.Clear();
    Labels.Clear();
    normalCurve = null;

    // Distanza fra gli istogrammi
    double dx = (GDeviceSize.Width - (HistogramWidth * Size)) / (Size + 1);

    // Coordinate iniziali x y degli istogrammi
    double x = dx;

    for (int i = 0; i < Size; i++)
    {
        Histograms.Add(new Histogram(x, VerticalAnchor, HistogramWidth, 0));
        Labels.Add(new ChartLabel(x, VerticalAnchor, HistogramWidth, "0"));

        x += dx + HistogramWidth;
    }
    normalCurve = new BellCurve(Size, GDeviceSize);

}
```

La posizione in cui cade la pallina alla fine di ogni ciclo di simulazione viene passata al grafico grazie al metodo pubblico `void IncrementValue(int index)`, esso si occupa di aggiornare ogni elemento grafico (istogrammi, curva e etichette) in modo appropriato.

1.4.2 Generazione istogrammi

Gli istogrammi vengono rappresentati in scala a dipendenza del peso che hanno nel grafico (riferirsi alla progettazione → 3.4 Istogrammi per i dettagli).

Di seguito è illustrato il codice che aggiorna l'altezza degli istogrammi. Questo spezzone è contenuto nel metodo `IncrementValue(int index)` di `DistributionChart.`

```
// Aggiorna altezza degli istogrammi
Histogram maxHistogram = Histograms.Aggregate((i1, i2) => i1.Value > i2.Value ? i1 : i2);

for (int i = 0; i < Histograms.Count; i++)
{
    Histogram h = Histograms[i];
    int value = h.Value;
    float perc = value / (float)maxHistogram.Value;
    double barHeight = Math.Round(perc * (GDeviceSize.Height));
    if (barHeight < 0) barHeight = 1;
    h.Height = barHeight;
    h.Y = VerticalAnchor - barHeight;
}
```

1.4.3 Generazione curva

La curva viene generata all'interno della classe `BellCurve`, che esporta il metodo pubblico `void UpdateCurve(int index)`, il quale aggiunge il valore specificato come parametro a una lista privata contenente il dataset da cui l'algoritmo di disegno va a prendere le informazioni. La classe esporta anche gli attributi `Mean`, `Variance` e `StdDev` (standard deviation), che usa internamente per tenere in memoria i dati necessari all'algoritmo di disegno. Il metodo privato `void DrawCurve()` assegna alla proprietà pubblica `Image` una `BitmapImage` contenente il disegno della curva. Il ViewModel accede alla proprietà `Image` e la riassegna a una sua proprietà che viene passata alla View con un binding.

Per rendere il disegno più facile, DrawCurve() crea un sistema di coordiante che gli è naturale, in modo che il codice definisca costanti per rappresentare l'area di disegno. Per esempio, per questo esempio ho usato $-5.1 \leq x \leq 5.1$, $-0.2 \leq y \leq 1.1$. Questi valori funnzionano bene per distribuzioni vicine allo 0 o per deviazioni standard relativamente piccole.

Successivamente, il codice crea un RectangleF che definisce l'area di disegno e un array di PointF che indicano dove il sia il punto più in alto a sinistra, in alto a destra e gli angoli bassi del mondo. Il tutto sarà quindi mappato per calzare l'oggetto Graphics, usato per il disegno. RectangleF e PointF[] sono usati per creare la matrice rapprentante la trasformazione dalle coordinate dell'area di disegno all'oggetto Graphics.

Di seguito è illustrato il codice di disegno.
Fonte: http://csharphelper.com/blog/2015/09/draw-a-normal-distribution-curve-in-c/

```
private void DrawCurve()
{
    Bitmap bm = new Bitmap(GDeviceSize.Width, GDeviceSize.Height);
    using (System.Drawing.Graphics gr = System.Drawing.Graphics.FromImage(bm))
    {
        gr.SmoothingMode = SmoothingMode.AntiAlias;

        // Definisce la mappatura di world (la matrice rappresentante l'area di disegno)
        // Ampiezza orizzontale
        const float wxmin = -5.1f;
        // Traslazione verticale
        const float wymin = -0.2f;
        const float wxmax = -wxmin;
        // Ampiezza verticale
        const float wymax = 1.1f;
        const float wwid = wxmax - wxmin;
        const float whgt = wymax - wymin;

        // Crea il rettangolo rappresentate l'area di disegno
        RectangleF world = new RectangleF(wxmin, wymin, wwid, whgt);

        // Crea i punti degli angoli dell'area di disegno
        PointF[] device_points =
        {
            new PointF(0, GDeviceSize.Height),
            new PointF(GDeviceSize.Width, GDeviceSize.Height),
            new PointF(0, 0),
        };

        // Crea la matrice sulla base del rettangolo e dei punti
        Matrix transform = new Matrix(world, device_points);

        // Crea una penna sottile da poter usare
        using (Pen pen = new Pen(Color.Red, 0))
        {
            using (Font font = new Font("Arial", 8))
            {
                // Disegna la curva
                gr.Transform = transform;
                List<PointF> points = new List<PointF>();

                float dx = (wxmax - wxmin) / GDeviceSize.Width;
                for (float x = wxmin; x <= wxmax; x += dx)
                {
                    double y = F(x, Mean, StdDev, Variance);
                    if (double.IsNaN(y)) y = 0;
                    points.Add(new PointF(x, (float)y));
                }

                pen.Color = Color.Red;
                gr.DrawLines(pen, points.ToArray());

            } // Font
        } // Pen
        Image = BitmapToImageSource(bm);
    }
}
```

1.4.4 Disegno grafico

Il grafico viene disegnato analogamente a come spiegato per il disegno delle stecche. Una risorsa statica che contiene il riferimento a una `CompositeCollection` viene passata a un `CollectionContainer`. E' specificata una `Image` che viene disegnata in sovraimpressione al resto per mostrare la curva.

```
<!-- Grafico -->
<Grid Grid.Column="1">
    <ItemsControl Panel.ZIndex="1000">
        <ItemsControl.ItemsSource>
            <CompositeCollection>
                <CollectionContainer Collection="{Binding Source={StaticResource
                  ChartItemsCollection}}"/>
                <Image Stretch="None">
                    <Image.Source>
                        <DrawingImage PresentationOptions:Freeze="True">
                            <DrawingImage.Drawing>
                                <DrawingGroup>
                                    <ImageDrawing Rect="{Binding Path=CurveDimensions}"
                                      ImageSource="{Binding Path=Curve}"/>

                                </DrawingGroup>
                            </DrawingImage.Drawing>
                        </DrawingImage>
                    </Image.Source>
                </Image>
            </CompositeCollection>
        </ItemsControl.ItemsSource>
        <ItemsControl.ItemsPanel>
            <ItemsPanelTemplate>
                <Canvas Focusable="False" Width="{Binding Path=CanvasWidth}"
                  Height="{Binding Path=CanvasHeight}"/>
            </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
        <ItemsControl.ItemContainerStyle>
            <Style>
                <Setter Property="Canvas.Left" Value="{Binding Path=X}"/>
                <Setter Property="Canvas.Top" Value="{Binding Path=Y}"/>
            </Style>
        </ItemsControl.ItemContainerStyle>
        <ItemsControl.Resources>
            <DataTemplate DataType="{x:Type model:Histogram}">
                <Rectangle Width="{Binding Path=Width}" Height="{Binding Path=Height}"
                  Fill="Blue" />
            </DataTemplate>
            <DataTemplate DataType="{x:Type model:ChartLabel}">
                <TextBlock Text="{Binding Path=Text}" Width="{Binding Path=Width}"
                  TextAlignment="Center" Background="Yellow"/>
            </DataTemplate>
        </ItemsControl.Resources>
    </ItemsControl>
</Grid>
```

1.4.5 Etichette

Le etichette rappresentano il conteggio delle volte che la pallina ha raggiunto un certo “foro”. Dal lato del code-behind la classe `ChartLabel` contiene alcuni attributi che vengono passati a dei `TextBlock` nello XAML.

1.5 Interfaccia grafica

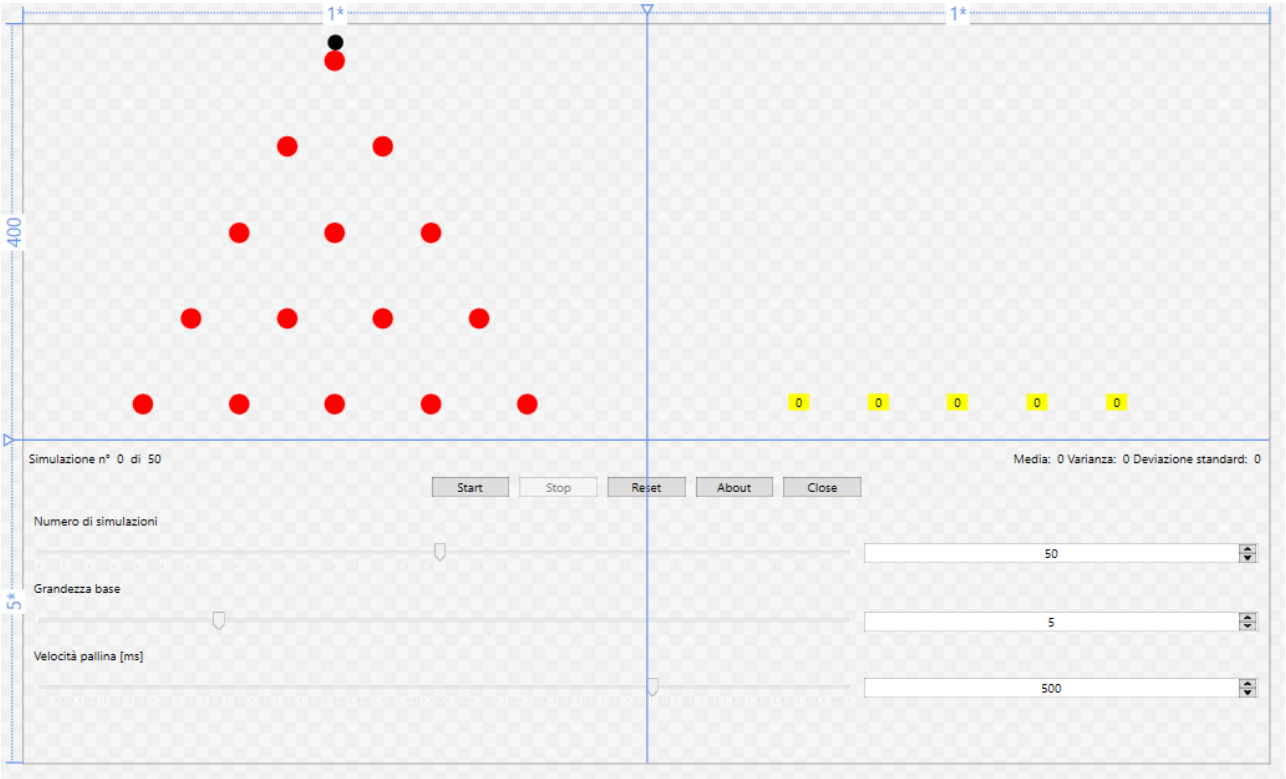


Figura 7 Layout grafico GUI

L'interfaccia grafica è stata suddivisa visivamente in 3 grosse aree. L'area superiore sinistry è dove viene mostrata l'animazione della simulazione, l'area superiore destra invece è dove vengono mostrati istogrammi e curva normale. La parte sottostante è dedicata ai controlli e ad alcune `TextBlock` che mostrano informazioni interessanti sulla simulazione in corso. Tutti i controlli sono contenuti in una `Grid`. La parte sottostante contiene a sua volta una `Grid` che contiene tutti i controlli. I bottoni sono messi una `StackPanel`.

Gli `IntegerUpDown` provengono da una libreria di controlli grafici esterna chiamata Xceed WPF Toolkit.

I controlli si abilitano/disabilitano dinamicamente a seconda dello stato dell'animazione. Grazie alla classe `DelegateCommand` che si appoggia all'interfaccia `IDelegateCommand`, il programma gestisce per mezzo di alcuni delegati le funzionalità dei bottoni e si appoggia sull'attributo globale booleano `IsSimulationRunning` per abilitare/disabilitare i controlli. In particolare:

- Il pulsante "Start" viene disabilitato durante l'esecuzione della simulazione e riabilitato al termine della stessa. Di default è abilitato.
- Il pulsante "Stop" viene abilitato durante l'esecuzione della simulazione e disabilitato al termine della stessa. Di default è disabilitato.
- Il pulsante "Reset" è sempre abilitato. Se lo si preme, esegua la funzione di Stop e in più resetta il programma ai valori di default.
- Il pulsante "About" fa apparire un `MessageBox` con scritto il nome dell'autore del programma.
- Il pulsante "Close" chiude l'applicazione
- Gli Slider e i correlati IntegerUpDown del numero di simulazioni e della grandezza base vengono disabilitati durante la simulazione.
- Lo Slider e correlato IntegerUpDown della velocità pallina rimangono sempre abilitati.

I controlli aggiornano i parametri del programma dinamicamente, ciò significa che se per esempio si modifica la grandezza della base, i cambiamenti si vedranno in tempo reale.

2 Test

2.1 Protocolli di test

Test Case **Riferimento**	TC-001 REQ-001/003	**Nome**	Slider “Numero di simulazioni”
Descrizione	Modificare i valori dei controlli “Numero di simulazioni” e vederne i cambiamenti in tempo reale		
Prerequisiti	Nessuno		
Procedura	1. Spostare il controllo dello slider a destra e a sinistra 2. Modificare il numero dell’IntegerUpDown		
Risultati attesi	1. Il valore del correlato IntegerUpDown deve modificarsi 2. Il valore della label “Simulazione n° x di y” deve modificarsi 3. Il valore del correlato Slider deve modificarsi		

Test Case **Riferimento**	TC-002 REQ-001/003	**Nome**	Slider “Grandezza base”
Descrizione	Modificare i valori dello slider “Grandezza base” e vederne i cambiamenti in tempo reale		
Prerequisiti	Nessuno		
Procedura	1. Spostare il controllo dello slider a destra e a sinistra 2. Modificare il numero dell’IntegerUpDown		
Risultati attesi	1. Il valore del correlato IntegerUpDown deve modificarsi 2. La base delle stecche e il numero di labels devono modificarsi conseguentemente 3. Il valore del correlato Slider deve modificarsi		

Test Case **Riferimento**	TC-003 REQ-001/001	**Nome**	Funzionamento della simulazione
Descrizione	Far partire una simulazione e verificare il funzionamento della caduta della pallina		
Prerequisiti	Nessuno		
Procedura	1. Cliccare su “Start” 2. Osservare la pallina cadere 3. Attendere che la simulazione finisca		
Risultati attesi	1. Al momento del click, la pallina deve iniziare a cadere 2. La pallina deve cadere in modo casuale a sinistra o destra 3. Arrivata in fondo alle stecche, la pallina deve ritornare nella stecca più in alto 4. Il valore della label “Simulazione n° x di y” deve modificarsi 5. La simulazione deve terminare al raggiungimento di “Numero simulazioni”		

Test Case **Riferimento**	TC-004 REQ-001/003	**Nome**	Slider “Velocità pallina [ms]”
Descrizione	Modificare i valori dello slider “Velocità pallina [ms]” e vederne i cambiamenti in tempo reale		
Prerequisiti	Nessuno		
Procedura:	1. Spostare il controllo dello slider a destra e a sinistra 2. Modificare il numero dell’IntegerUpDown 3. Far partire una simulazione 4. Ripetere punti 1 e 2		
Risultati attesi	1. Il valore del correlato IntegerUpDown deve modificarsi 2. Il valore del correlato Slider deve modificarsi 3. La velocità della pallina deve modificarsi conseguentemente durante la simulazione		

Test Case **Riferimento**	TC-005 REQ-001/003	**Nome**	Pulsante “Start”
Descrizione	Cliccare sul pulsante “Start” e osservare la disattivazione di determinati controlli		
Prerequisiti	Nessuno		
Procedura:	1. Cliccare sul pulsante “Start” 2. Osservare il comportamento del pulsante “Start” e “Stop” 3. Osservare il comportamento degli slider “Numero di simulazioni”, “Grandezza base” e dei relativi IntegerUpDown		
Risultati attesi	1. Il pulsante “Start” deve disabilitarsi 2. Il pulsante “Stop” deve abilitarsi 3. “Numero di simulazioni”, “Grandezza base” e i relativi IntegerUpDown devono disabilitarsi		

Test Case **Riferimento**	TC-006 REQ-001/003	**Nome**	Pulsante “Stop”
Descrizione	Cliccare sul pulsante “Stop” e osservare la disattivazione di determinati controlli		
Prerequisiti	Nessuno		
Procedura:	1. Cliccare sul pulsante “Stop” 2. Osservare il comportamento del pulsante “Start” e “Stop” 3. Osservare il comportamento degli slider “Numero di simulazioni”, “Grandezza base” e dei relativi IntegerUpDown		
Risultati attesi	1. Il pulsante “Stop” deve disabilitarsi 2. Il pulsante “Start” deve abilitarsi 3. “Numero di simulazioni”, “Grandezza base” e i relativi IntegerUpDown devono abilitarsi		

Test Case **Riferimento**	TC-007 REQ-001/003	**Nome**	Pulsante “Reset”
Descrizione	Cliccare sul pulsante “Reset” e osservare lo stop dell’applicazione e il settaggio dei valori di default		
Prerequisiti	Nessuno		
Procedura:	1. Modificare il valore degli slider “Numero di simulazioni”, “Grandezza base” e “Velocità pallina [ms]” 2. Cliccare sul pulsante “Reset” 3. Osservare il comportamento dei controlli e dei relativi IntegerUpDown 4. Cliccare sul pulsante “Start” 5. Ripetere i punti 1, 2 e 3		
Risultati attesi	1. I valori dei controlli devono tornare a quelli di default 2. La pallina deve tornare in cima alle stecche 3. Gli istogrammi devono cancellarsi 4. Il pulsante “Stop” deve abilitarsi 5. “Numero di simulazioni”, “Grandezza base” e i relativi IntegerUpDown devono disabilitarsi 6. Gli istogrammi devono cancellarsi 7. La curva deve cancellarsi 8. Le etichette devono azzerarsi		

Test Case **Riferimento**	TC-008 REQ-001/003	**Nome**	Pulsante “About”
Descrizione	Cliccare sul pulsante “About” e osservare lo stop dell’applicazione e il settaggio dei valori di default		
Prerequisiti	Nessuno		
Procedura:	1. Premere il pulsante “About”		
Risultati attesi	1. Alla pressione del pulsante “About” deve comparire una MessageBox MessageBox contenente una nota sul creatore del progetto		

Test Case **Riferimento**	TC-009 REQ-001/003	**Nome**	Pulsante “Close”
Descrizione	Cliccare sul pulsante “Close” e osservare		
Prerequisiti	Nessuno		
Procedura:	1. Premere il pulsante “Close”		
Risultati attesi	1. Alla pressione del pulsante “About” l’applicazione deve chiudersi		

Test Case **Riferimento**	TC-010 REQ-002/003	**Nome**	Disegno della curva normale
Descrizione	Avviare la simulazione e visualizzare la curva normale		
Prerequisiti	Nessuno		
Procedura:	1. Premere il pulsante “Start” 2. Attendere che la pallina raggiunga 3 punti di arrivo diversi 3. Attendere la fine della simulazione		
Risultati attesi	1. Raggiunti 3 punti di arrivo diversi, la curva deve disegnarsi 2. La curva deve adattarsi a dipendenza di dove la pallina va a cadere. Se cade più verso la destra, la curva dovrà spostarsi verso destra e viceversa. Se la pallina cade su un dato punto, la curva deve restringersi di più in quella posizione. Se la pallina cade in modo più distribuito, la curva dovrà appiattirsi con la progressione della distribuzione 3. Le label “Media”, “Varianza” e “Deviazione standard” devono aggiornarsi mostrando i valori corrispondenti alla curva disegnata		

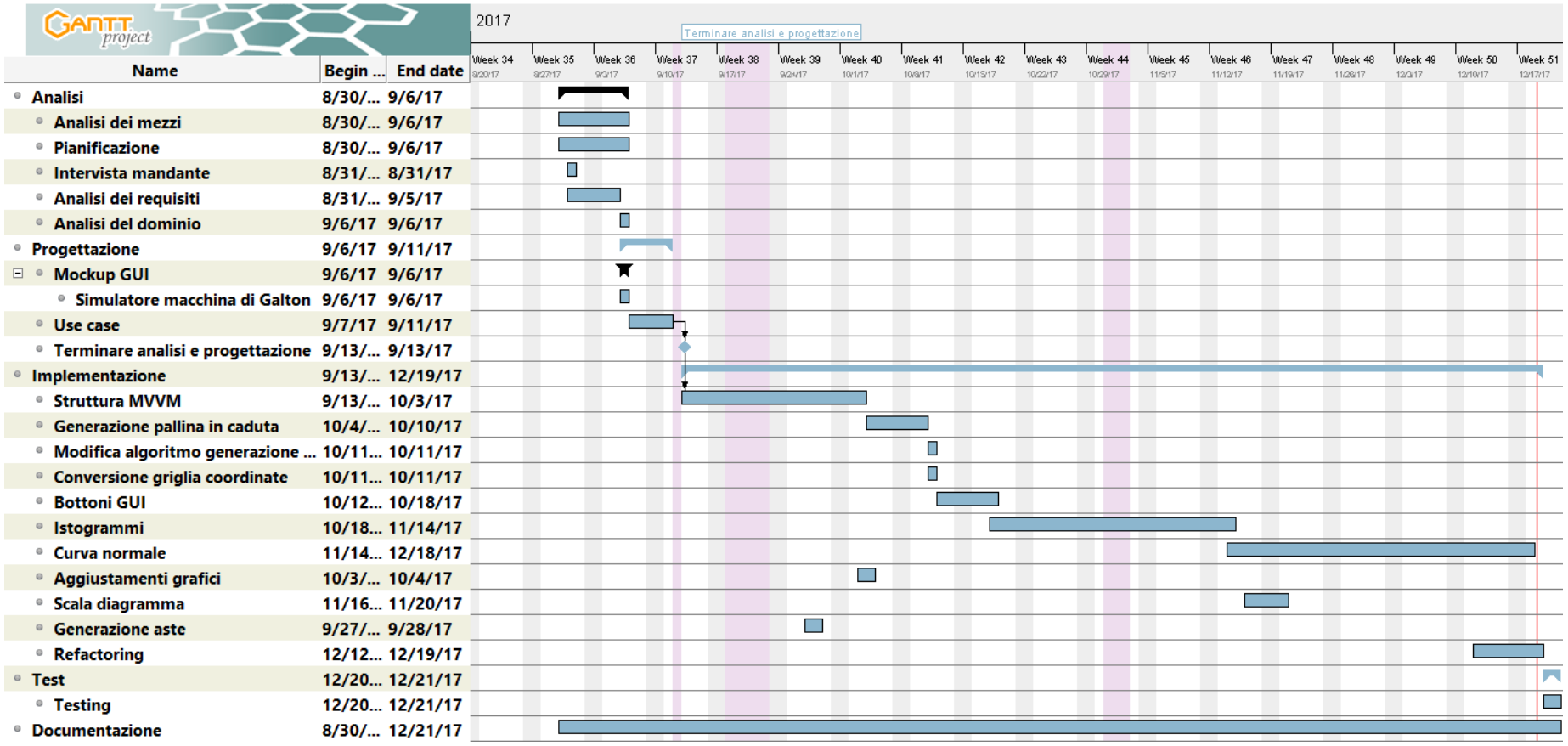
Test Case **Riferimento**	TC-011 REQ-002/002	**Nome**	Disegno degli istogrammi
Descrizione	Avviare la simulazione e visualizzare gli istogrammi		
Prerequisiti	Nessuno		
Procedura:	1. Premere il pulsante “Start” 2. Attendere che la pallina raggiunga la base delle stecche 3. Attendere la fine della simulazione		
Risultati attesi	1. Quando la pallina raggiunge la base delle stecche, deve generarsi un istogramma grande il 100% dell’altezza corrispondente a quella posizione 2. Quando le successive palline raggiungono la base delle stecche, l’istogramma corrispondente deve aggiornarsi. L’altezza dell’istogramma corrisponde in percentuale al peso dello stesso 3. La label corrispondente deve incrementarsi 4. La configurazione della distribuzione deve essere simile a quella di una curva a campana nella maggior parte delle simulazioni		

2.2 Risultati test

Test case	Risultato	Osservazioni
TC-001	**OK**	
TC-002	**OK**	
TC-003	**OK**	
TC-004	**OK**	
TC-005	**OK**	
TC-006	**OK**	
TC-007	**OK**	
TC-008	**OK**	
TC-009	**OK**	
TC-010	**NON OK (2/3)**	Il risultato atteso n° 2 ha dato esito negativo. La curva continuna ad appiattirsi progressivamente e non si sposta lateralmente.
TC-011	**OK**	

3 Consuntivo e conclusioni

3.1 Diagramma di Gantt consuntivo



Name	Begin ...	End date
Analisi	8/30/...	9/6/17
Analisi dei mezzi	8/30/...	9/6/17
Pianificazione	8/30/...	9/6/17
Intervista mandante	8/31/...	8/31/17
Analisi dei requisiti	8/31/...	9/5/17
Analisi del dominio	9/6/17	9/6/17
Progettazione	9/6/17	9/11/17
Mockup GUI	9/6/17	9/6/17
Simulatore macchina di Galton	9/6/17	9/6/17
Use case	9/7/17	9/11/17
Terminare analisi e progettazione	9/13/...	9/13/17
Implementazione	9/13/...	12/19/17
Struttura MVVM	9/13/...	10/3/17
Generazione pallina in caduta	10/4/...	10/10/17
Modifica algoritmo generazione ...	10/11...	10/11/17
Conversione griglia coordinate	10/11...	10/11/17
Bottoni GUI	10/12...	10/18/17
Istogrammi	10/18...	11/14/17
Curva normale	11/14...	12/18/17
Aggiustamenti grafici	10/3/...	10/4/17
Scala diagramma	11/16...	11/20/17
Generazione aste	9/27/...	9/28/17
Refactoring	12/12...	12/19/17
Test	12/20...	12/21/17
Testing	12/20...	12/21/17
Documentazione	8/30/...	12/21/17

Figura 8 Diagramma di Gantt consuntivo

3.2 Costi

Nome risorsa	Costo unitario	Quantità	Totale
Cristiano Colangelo (risorsa umana)	35 [fr/h]	156 ore	5'600 [fr]

Totale spesa: **5'600 CHF**

3.3 Mancanze/limitazioni conosciute

- La curva normale non viene disegnata correttamente. La curva continua ad appiattirsi e manca la scalatura della stessa.
- Manca il completamento del REQ-003 inerente al sistema di report (comunque un requisito opzionale).
- Gli `IntegerUpDown` non si sono riadattati correttamente secondo i pesi della `Grid`, ho quindi provveduto a ridimensionarli manualmente applicando dei `Margin` sia agli `Slider` che agli `IntegerUpDown`. Sono al corrente che non è una soluzione ottimale ma sospetto fortemente (anche alla luce di un controllo effettuato dal docente responsabile) che il problema sia legato agli `IntegerUpDown` in sè.

3.4 Sviluppi futuri

Sarebbe interessante aggiungere un'altra rappresentazione grafica chiamata Box Plot (in italiano diagramma a scatola e baffi) che sarebbe interessante da integrare. Purtroppo ne sono venuto a conoscenza quando ero già concentrato ad implementare la curva, ma con il senno di poi avrei implementato il box plot invece della curva normale: sarebbe stato meno problematico.

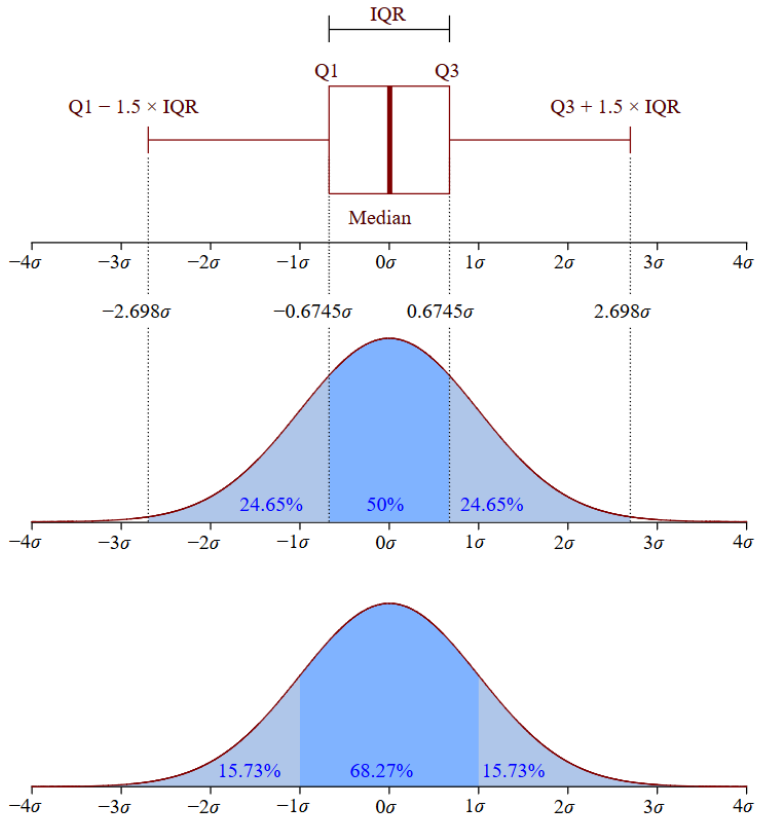


Figura 9 Schema Box Plot

In statistica il diagramma a scatola e baffi (o diagramma degli estremi e dei quartili o box and whiskers plot o box plot) è una rappresentazione grafica utilizzata per descrivere la distribuzione di un campione tramite semplici indici di dispersione e di posizione. Viene rappresentato (orientato orizzontalmente o verticalmente) tramite un rettangolo diviso in due parti, da cui escono due segmenti. Il rettangolo (la "scatola") è delimitato dal primo e dal terzo quartile, q1/4 e q3/4, e diviso al suo interno dalla mediana, q1/2. I segmenti (i "baffi") sono delimitati dal minimo e dal massimo dei valori.
In questo modo vengono rappresentati graficamente i quattro intervalli ugualmente popolati delimitati dai quartili. **[fonte: wikipedia]**

Inoltre sarebbe stato interessante aggiungere più palline contemporaneamente che scendevano una dopo l'altra, cosa che ho provato a fare ma che si è rivelata essere un task troppo lungo da fare e che ho quindi abbandonato. Un'altra idea che mi è venuta in mente è stata quella di aggiungere dei pulsanti per navigare indietro e avanti nella simulazione e per mettere in pausa la simulazione. Il commitente aveva proposto anche di aggiungere un'animazione alle palline per abbellire la simulazione quando scendono, ma in ultima analisi avendo un'unica pallina questa animazione non è stata strettamente necessaria ai fini della comprensione per l'utente.

3.5 Considerazioni personali

Ho trovato questo progetto interessante, anche dal punto di vista della statistica. Purtroppo ho avuto non poca frustrazione per quanto riguarda la generazione della curva. Ho ricercato informazioni sulla natura del problema in lungo e in largo (l'appiattimento progressivo della curva), cercando sul web e chiedendo al docente responsabile nonchè a un docente di programmazione e una docente di matematica, ma nessuno ha saputo darmi indicazioni precise. Ritengo di conseguenza di essermi sforzato al massimo delle mie conoscenze di programmazione e matematica/statistica. Inoltre ho trovato alquanto "tediosi" i meccanismi di binding di C#, un linguaggio molto potente e ricco di funzionalità, ma forse un po' sovradimensionato per un progetto così semplice. Al contrario ho trovato molto comodi i delegati e la classe `DelegateCommand` per gestire i controlli.

4 Sitografia

https://geniincompresse.files.wordpress.com/2013/03/220px-galton_box-svg.png
https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/MonoGame_Logo.svg/2000px-MonoGame_Logo.svg.png
https://webtooling.visualstudio.com/themes/standard/favicon/mstile-310x310.png
https://www.mathsisfun.com/data/standard-normal-distribution.html
https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1

5 Allegati

- Diari di lavoro
- Codice sorgente
- Diagramma delle classi
- Quaderno dei compiti
- File eseguibile del prodotto