XamlKonverter – Implementierungsdetails

1 Möglichkeiten des XamlKonverter's

Beim XamlKonverter haben wir uns zunächst vor allem auf die Umsetzung möglichst vieler grundlegender Konzepte konzentriert, die Xaml bietet und die u.a. in der Xaml-Dokumentation unter http://msdn2.microsoft.com/en-us/library/ms752059.aspx aufgeführt werden. Mit der bestehenden Struktur sollte es leicht möglich sein neue GUI-Elemente mit ihren Properties zu integrieren und nutzbar zu machen.

Nachfolgend ist eine kleine Liste unterstützter Xaml-Konzepte anhand von Beispielen aufgeführt:

Objekte

Einzelne Knoten im Xaml-Baum stellen mit ihren Tag-Namen im C#-Code Objekte dar, die eine Variable besitzen und deren Properties Werte zugewiesen werden.

Beispiel:
Xaml:
```
<Grid />
```
C#:
```
Grid grid1 = new Grid();
```

Unqualifizierte Werte-Attribute

Wert-Attribute können ohne weitere Qualifizierung direkt innerhalb einer Xaml-Tag-Definition stehen.

Beispiel:
Xaml:
```
<Grid Background="Red" />
```
C#:
```
Grid grid1 = new Grid();
grid1.Background = Brushes.Red;
```

Qualifizierte Werte-Attribute

Wert-Attribute lassen sich mit „`klasse.attributname`" qualifizieren, wobei auch direkt die Klassen Verwendung finden können:

Beispiel:
Xaml:
```
<Grid Grid.Width="100" Panel.Background="Red" />
```
C#:
```
Grid grid1 = new Grid();
```

```
grid1.Width = 100;
grid1.Background = Brushes.Red;
```

Qualifizierte Werte-Attribute als einzelne Tags

Durch die Qualifizierung von Wert-Attributen können sie auch als gesonderte Knoten unter dem Xaml-Knoten stehen, zu dem sie gehören. Die Behandlung findet bei Wert-Attributen statt, als ob sie direkt im zugehörigen Knoten angegeben worden wären.

Beispiel:
Xaml:
```
<Grid Width="100">
  <Grid.Height>100</Grid.Height>
  <Grid.Background>Red</Grid.Background>
</Grid>
```
C#:
```
Grid grid1 = new Grid();
grid1.Width = 100;
grid1.Height = 100;
grid1.Background = Brushes.Red;
```

Qualifizierte Element-Attribute

Properties eines Objekts in C# können nicht nur einen Wert annehmen, sondern auch eine weitere Variable zugewiesen bekommen. Dies ist auch durch die Qualifizierung von Attributen in Xaml möglich. Ein qualifiziertes Attribut als eigenständiger Xaml-Knoten kann weitere Knoten enthalten, sodass sich unter ihm eine ganze Baumstruktur auf tut.

Beispiel:
Xaml:
```
<Grid>
  <Grid.Width>100</Grid.Width>
  <Grid.Background>
    <SolidColorBrush>Red</SolidColorBrush>
  </Grid.Background>
</Grid>
```
C#:
```
Grid grid1 = new Grid();
grid1.Width = 100;
SolidColorBrush brush1 = new SolidColorBrush();
brush1.Color = Colors.Red;
grid1.Background = brush1;
```

Zu beachten ist, dass `Grid.Width` nur einen Wert enthält und somit dafür keine neue Variable angelegt wird. Sobald ein qualifizierter Attribut-Knoten jedoch mind. einen weiteren Kindknoten besitzt (wie `Grid.Background` im Beispiel), werden dafür neue Objekte angelegt.

Element-Attribute als Collection

Qualifizierte Attribute, die als eigenständige Xaml-Knoten stehen und weitere Knoten enthalten (also „Element-Attribute“ und keine „Werte-Attribute“ sind), repräsentieren eine Property eines Objekts in C#. Diese Property kann allerdings auch eine Collection sein,

sodass ihr mehrere Elemente zugewiesen werden können. Ein Element-Attribut-Knoten in Xaml kann dann mehrere Kind-Knoten beinhalten, die dem übergeordneten Objekt mit der Collection-Methode „`Add()`“ hinzugefügt werden.

Beispiel:
Xaml:
```
<Grid>
  <Grid.Width>100</Grid.Width>
  <Grid.Background>
    <SolidColorBrush>Red</SolidColorBrush>
  </Grid.Background>
  <Grid.Children>
    <Button Content=”OK” />
    <StackPanel Name=”myPanel” />
  </Grid.Children>
</Grid>
```
C#:
```
Grid grid1 = new Grid();
grid1.Width = 100;
SolidColorBrush brush1 = new SolidColorBrush();
brush1.Color = Colors.Red;
grid1.Background = brush1;
Button button1 = new Button();
button1.Content = “OK”;
StackPanel myPanel = new StackPanel();
myPanel.Name = “myPanel”;
grid1.Children.Add(button1);
grid1.Children.Add(myPanel);
```

Unqualifizierte Collection-Kind-Attribute

Element-Attribute, welche eine Collection darstellen, können statt in einem qualifizierten Attribut-Knoten auch direkt unter den Vaterknoten geschrieben werden. Das letzte Beispiel von oben würde dann wie folgt aussehen. Man beachte, dass die erzeugten C#-Codes äquivalent sind.

Beispiel:
Xaml:
```
<Grid>
  <Grid.Width>100</Grid.Width>
  <Grid.Background>
    <SolidColorBrush>Red</SolidColorBrush>
  </Grid.Background>
  <Button Content=”OK” />
  <StackPanel Name=”myPanel” />
</Grid>
```
C#:
```
Grid grid1 = new Grid();
grid1.Width = 100;
SolidColorBrush brush1 = new SolidColorBrush();
brush1.Color = Colors.Red;
grid1.Background = brush1;
Button button1 = new Button();
button1.Content = “OK”;
StackPanel myPanel = new StackPanel();
myPanel.Name = “myPanel”;
grid1.Children.Add(button1);
grid1.Children.Add(myPanel);
```

XmlText als Attribut

Ist innerhalb einer Xaml-Tag-Spezifikation reiner Text enthalten, so kann im XamlKonverter spezifiziert werden, wenn dieser zu einem Attribut gehört. Im erzeugten Code wird er dann dem entsprechenden Property zugeordnet:

Beispiel:
Xaml:
```
<Button Width="100">Ich bin ein Button</Button>
```
C#:
```
Button button1 = new Button();
button1.Width = 100;
button1.Content = "Ich bin ein Button";
```

Der Xaml-Code ist dabei semantisch äquivalent zu:
```
<Button Width="100" Content="Ich bin ein Button" />
```

XmlText als Kindknoten mit Attribut

Ist innerhalb einer Xaml-Tag-Spezifikation reiner Text enthalten, so kann im XamlKonverter spezifiziert werden, wenn dieser zu einem neuen Kindknoten als Wert eines Attributs gehört. Im erzeugten Code wird dann das Objekt des Kindknotens erzeugt und der XmlText dem entsprechenden Property dieses Knotens zugeordnet:

Beispiel:
Xaml:
```
<Menu Width="100">Menu-Eintrag</Menu>
```
C#:
```
Menu menu1 = new Menu();
menu1.Width = 100;
MenuItem menuItem1 = new MenuItem();
menuItem1.Header = "Menu-Eintrag";
menu.Items.Add(menuItem1);
```

Der Xaml-Code ist dabei semantisch äquivalent zu:
```
<Menu Width="100">
  <MenuItem Header="Menu-Eintrag" />
</Menu>
```

Zugreifbare GUI-Objekte

Xaml-Knoten, die das Attribut „Name“ definiert haben, sind im C#-Code in der gesamten erstellten Klasse zugreifbar, werden dort also als Member-Variablen definiert.

Beispiel:
Xaml:
```
<Grid Name="myGrid" Background="Red">
  <Button />
</Grid>
```
C#:
```
class GUI{
```

```
  Grid myGrid = new Grid();

  public GUI(){
    InitializeComponent();
  }

  void InitializeComponent(){
    Button button1 = new Button();
    myGrid.Background = Brushes.Red;
    myGrid.Add(button1);
  }
}
```

HINWEIS: Zur Zeit ist es noch nicht ausgeschlossen, dass ein mit „Name“ definiertes Objekt im erzeugten C#-Code einen anderen Variablennamen besitzt. Der Name muss genau dann umbenannt werden, wenn bereits eine automatisch erzeugte Variable mit diesem Namen existiert. Dies sollte allerdings fast nie der Fall sein und stellt eine absolute Ausnahme dar.

Flexible Wert-Formatierungen

Werte der verschiedenen Typen können in Xaml auf unterschiedliche Arten angegeben werden. Durch gesondert programmierte Attributwert-Konverter-Klassen werden sie in Code übersetzt, der als rechte Seite einer Anweisung in C# stehen kann.

Beispiel:
Xaml:
```
<Button Background=“#00FF99“ />
```
C#:
```
Button button1 = new Button();
button1.Background = new SolidColorBrush(Color.FromArgb(255,0,255,153));
```

Angehängte Properties

Dabei handelt es sich um Properties, die sich nicht direkt auf den aktuellen Knoten beziehen, sondern auf einen übergeordneten Knoten. Diese werden in der aktuellen Version des XamlKonverters nicht unterstützt. Ein Beispiel für angehängte Properties wäre:

Beispiel:
Xaml:
```
<DockPanel>
  <Button DockPanel.Dock="Left" Width="100" Height="20">Left</Button>
  <Button DockPanel.Dock="Right" Width="100" Height="20">Right</Button>
</DockPanel>
```

Events, Namespace-Erweiterungen, Ressourcen

Für diese existiert vorerst ebenfalls keine Behandlung im XamlKonverter.

2 Allgemeine Vorgehensweise

Indem der zu übersetzende Xaml-Code zunächst als `XmlReader` eingelesen wird, um dann durch einen `XamlReader` in ein `UIElement` übersetzt zu werden, werden bereits vor der Abarbeitung Xml- und Xaml-Syntax direkt durch die C#-Mechanismen geprüft. Dadurch musste hier keine Entwicklungsarbeit aufgebracht werden. Einzig die Klasse `XamlFilter` war notwendig, um Objekte aus dem Code zu entfernen, welche das dynamische Erstellen eines `UIElement` verhindern.

Die Erstellung des C#-Codes aus dem Xaml-Code geschieht in 2 Schritten.

- In **Schritt 1** wird über die Klasse `CSTree` ein Baum erstellt, der die in Xaml definierten Elemente als vorverarbeitete Baumknoten enthält. Ein Hauptgrund zur Erstellung eines Baumes vor der Code-Erzeugung war die gleichwertige Behandlung von Fällen, die im Xaml-Code unterschiedlich definiert werden können, sich in C# aber gleich ausdrücken. Ein zweiter Grund war u.a. die Erstellung eindeutiger Variablennamen für den C#-Code, was durch Speicherung dieser in den Baumknoten erreicht werden konnte.
- In **Schritt 2** wird aus dem Baum durch Traversierung dessen der eigentliche Code erzeugt. Dabei wird gekapselter Code derart zurück gegeben, dass er a) in verschiedene Logik-Teile gegliedert ist (Element-Code, Hinzufüge-Code, Using-Deklarationen, Member-Variablen-Deklarationen) und b) nicht als `string` gespeichert wird, sondern in konkreten Realisierungen des Datentyps Statement (für eine Anweisung).

3 Baumbeschreibung

Die Klasse `CSTree` beschreibt die Baumstruktur, welche bei der Verarbeitung des Xaml-Codes aufgebaut wird. Wurde der Baum erst einmal erstellt, so muss der Xaml-Code nicht wieder betrachtet werden, da sich alle Informationen im Baum wiederfinden, der dann die beste Grundlage zur Erstellung des C#-Codes bildet.

Die grundlegende Idee des Baumes ist es, für jedes Element im Xaml-Code einen eigenen Baumknoten zu halten. Daher gibt es auch verschiedene Arten von Baumknoten, die in einem der kommenden Unterabschnitte beschrieben werden. D.h. einzelne Xaml-Knoten werden durch Baumknoten repräsentiert, aber auch für die einzelnen Attribute eines Xaml-Knotens werden extra Baumknoten bereitgestellt.

Aufbau und Codeerzeugung

Der Aufbau des gesamten Baumes findet durch die Konstruktion der Baumwurzel statt. Dazu wird die Wurzel des Xaml-Codes übergeben. Die Erstellung des untergeordneten Baumes erfolgt über Verarbeitung der Baumwurzel und rekursive Konstruktoraufrufe für die Kindelemente.

Code in gekapselter Form über die Klasse `CSCodeStorage` erhält man, indem für die Baumwurzel die Methode `GetCode()` aufgerufen wird. Rekursiv wird der Baum traversiert und die enthaltenen Informationen werden in gekapselten C#-Code umgewandelt, der dann zurück gegeben wird.

Arten von Baumknoten

Wie bereits erwähnt wurde, gibt es mehrere Arten von Baumknoten, um unterschiedliche Arten von Knoten des Xaml-Codes behandeln und speichern zu können. Allgemein kann zunächst unterschieden werden zwischen Xaml-Knoten, die für eigenständige Variablen im C#-Code stehen und Attributen von Xaml-Knoten, die lediglich zur Speicherung eines Wertes dienen. Die dritte Art von Baumknoten ist als Mischling der ersten beiden anzusehen. Diese Knoten stehen für ein Attribut eines Xaml-Knotens, enthalten aber eine Liste weiterer Kindknoten, die eine eigene Baumhierarchie bilden und über Variablen im C#-Code zu der entsprechenden Property des Vaterknotens hinzugefügt werden.

ChildElementNode

Dieser Baumknoten-Typ speichert einen Xaml-Knoten, der im C#-Code durch ein eigenes Objekt repräsentiert wird. Dazu werden der Typ des Objektes (entspricht dem Tag-Namen des Xaml-Knotens) und der eindeutig generierte Variablenname gespeichert. Weiterhin ist eine Liste von `PropertyValueNode`'s enthalten, durch die die zu setzenden Properties des Objekts gespeichert werden. Außerdem ist eine Liste von `ElementNode`'s enthalten, die jeder für sich die Baumstruktur nach unten erweitern.

Beispiel:

```
<Grid Width="100" Height="100">
  <Button />
</Grid>
```

Hieraus würde ein `ChildElementNode` „`Grid`" erzeugt werden. Als Kind-Elementknoten bekommt dieser Knoten einen `ChildElementNode` „`Button`", zudem sind in der Liste der Wert-Attributknoten (`PropertyValueNode`'s) die Knoten für „`Width`" und „`Height`" gespeichert.

PropertyValueNode

In diesem Baumknoten-Typ wird ein Xaml-Attribut gespeichert, welches als Property eines Objekts im C#-Code direkt gesetzt werden kann. Dazu werden Name und Wert der Property gespeichert, wobei der Wert ein `string` ist, der so direkt auf der rechten Seite der Property-Zuweisung in C# stehen kann. Dazu ist i. Allg. eine Konvertierung des Wertes aus dem Xaml-Code notwendig, wofür die `AttribValueConverter`-Klassen gedacht sind.

Beispiel:

```
<Grid Background="Red" />
```

Hier würde ein `ChildElementNode` „`Grid`" generiert werden, welcher in der Liste von Wert-attribut-Knoten einen `PropertyValueNode` enthalten würde mit „`Background`" als Name und „`Brushes.Red`" als konvertierter Wert.

`PropertyValueNode`'s werden durch die `NodeGenerator`-Hierarchie des zugrunde liegenden `ChildElementNode` gebildet, indem im Konstruktor des `ChildElementNode` eine Liste der Xaml-Attribute erstellt wird und dann der Methode `GenerateAttribNodes()` des `NodeGenerators` dieses Knotens übergeben wird, wodurch man eine Liste von `PropertyValueNode`'s erhält.

PropertyElementNode

Dieser Typ eines `CSTree`-Baumknotens wird erzeugt, wenn ein Xaml-Attribut als eigenständiger Knoten unter seinem Xaml-Element erscheint und dieser Knoten weitere Unterknoten besitzt. Dem Xaml-Attribut, welches ja für ein Property des im Oberknoten gespeicherten Objekts steht, wird in diesem Fall die Variable zugewiesen, welche im Kopf des Unterbaumes gespeichert ist. `PropertyElementNode` ist daher eine Mischform von `PropertyValueNode` und `ChildElementNode`. Die eigentlichen Werte, die bei `PropertyValueNode` direkt im Knoten stehen, sind hier wie bei `ChildElementNode` auf den Unterbaum verteilt.

Beispiel:

```
<Grid Width="100">
  <Grid.Height>100</Grid.Height>
  <Grid.Background>
    <SolidColorBrush>Red</SolidColorBrush>
  </Grid.Background>
</Grid>
```

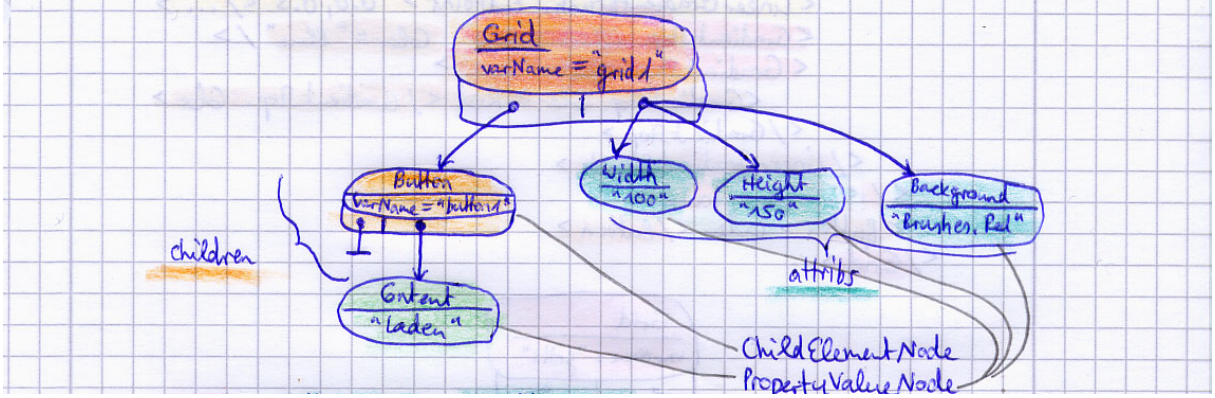
In diesem Beispiel würde ein `ChildElementNode` für „`Grid`“ erzeugt werden. Die Wertattribute hier sind „`Width`“ und „`Height`“, für diese werden `PropertyValueNode`’s erzeugt. Zu beachten ist hier, dass „`Grid.Height`“ zwar als eigenständiger Unterknoten von `Grid` erscheint, allerdings keine weiteren Unterknoten, sondern nur Text enthält. Damit wird `Grid.Height` als Werteattribut behandelt. „`Grid.Background`“ hingegen besitzt weitere Unterknoten. Daher wird hierfür ein „`PropertyElementNode`“ erzeugt, der als Unterknoten einen `ChildElementNode` „`SolidColorBrush`“ enthält.

Eine wichtige interne Unterscheidung bei `PropertyElementNode` besteht noch darin, ob es sich bei dem repräsentierten Property um ein normales Werte-Element handelt oder um eine Collection, der mehrere Objekte hinzugefügt werden können. Stellt die Property einen Wert dar, so besitzt die Kindliste des `PropertyElementNode` (wie auch in Xaml) genau einen Knoten. Im Falle einer Collection ist die Liste mit mind. 1 Sohn beliebig lang. Im C#-Code werden Zuweisungen an die Collection per `collection.Add()` geregelt.
Im XamlKonverter ist die Unterscheidung zwischen Werte-Property und Collection-Property so geregelt, dass in jedem `NodeGenerator` gespeichert wird, welche Attribute Collections darstellen.

Beispiele für erzeugte Bäume

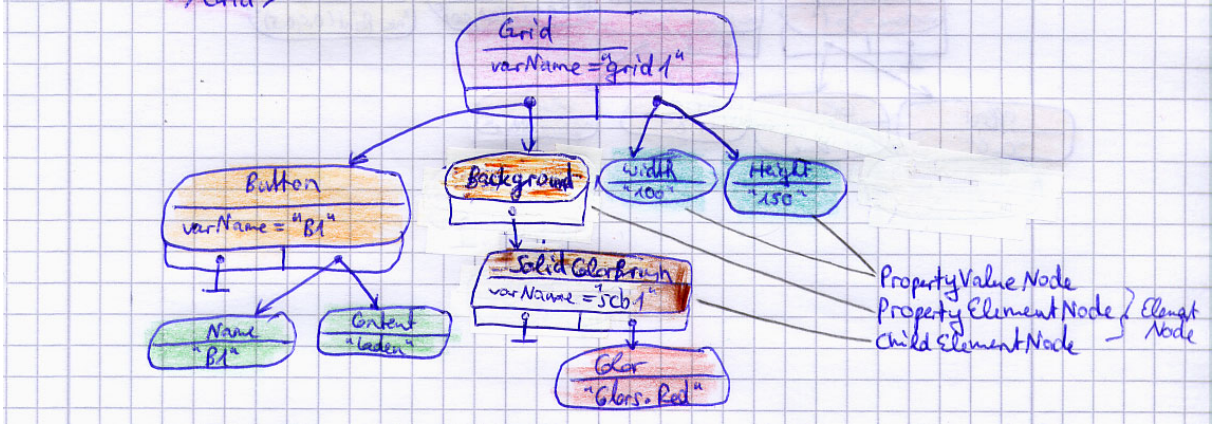
Bsp. 1)

```
<Grid Width="100" Height="150">
    <Grid.Background>Red</Grid.Background>
    <Button>laden</Button>
</Grid>
```



Bsp. 2)

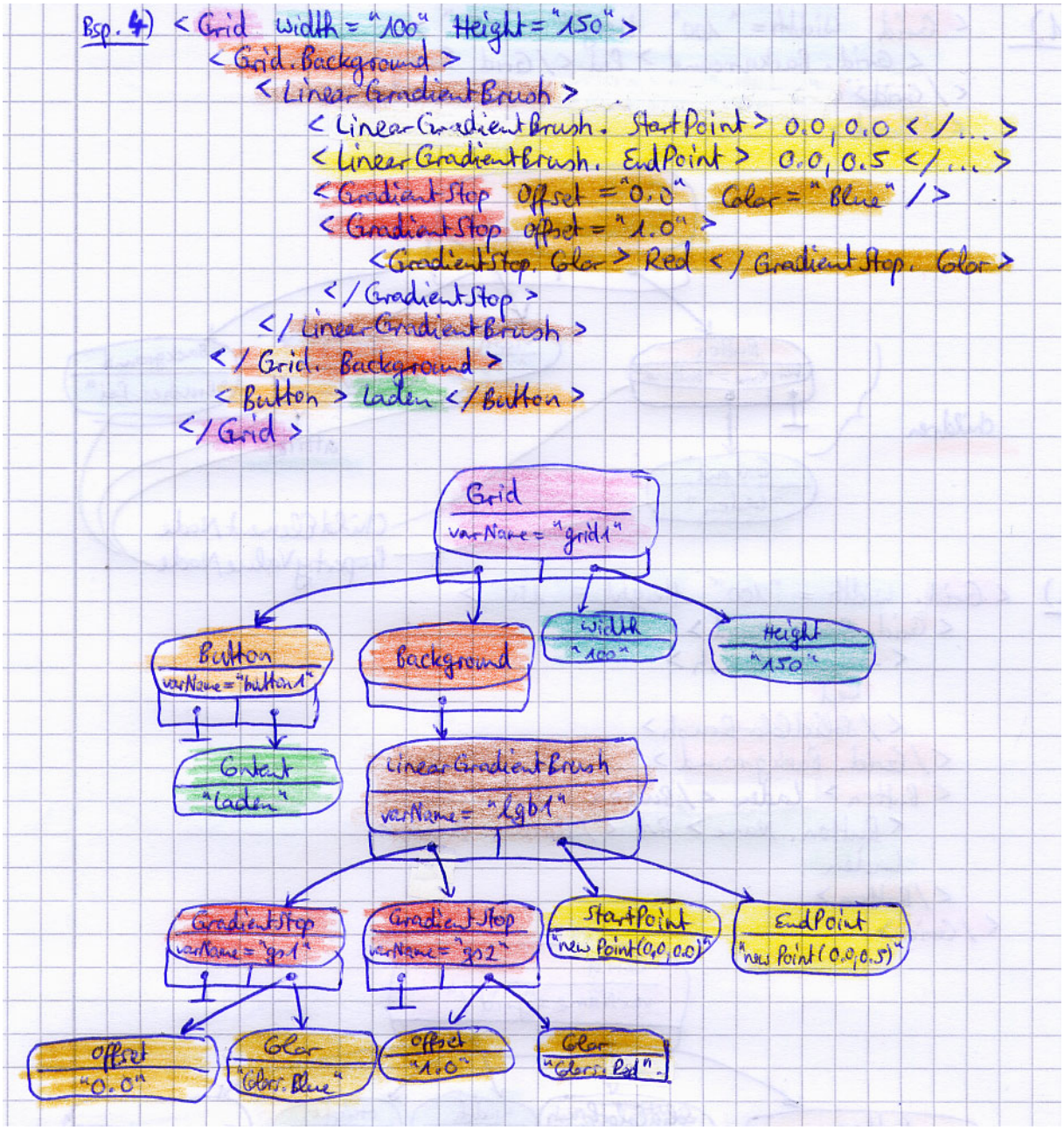
```
<Grid Width="100" Height="150">
    <Grid.Background>
        <SolidColorBrush>
            Red
        </SolidColorBrush>
    </Grid.Background>
    <Button>
        <Button.Name>B1</Button.Name>
        laden
    </Button>
</Grid>
```



Bsp. 3)

```
<Grid Width="100" Height="150" Background="Red">
    <Button>laden</Button>
</Grid>
```

↳ gleicher Baum und Code wie in Bsp. 1



4 Node-Generatoren und Hierarchiebildung

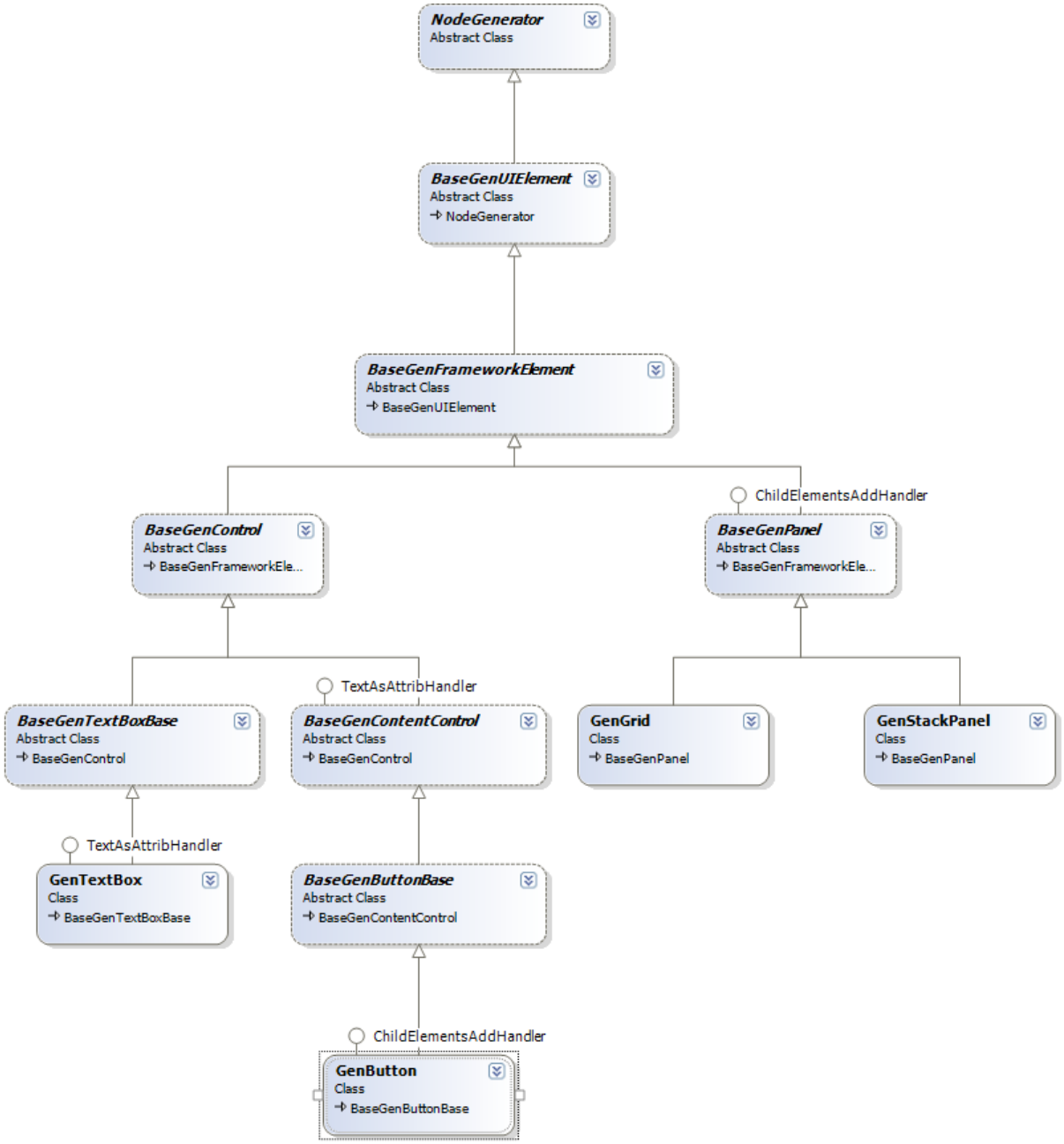
Ein Basis-Konzept der Programmierung des XamlKonverters bilden die Node-Generatoren und die darauf basierende Abbildung der Hierarchie von GUI-Elementen in C#.

Die Klasse `NodeGenerator` bzw. ihre Vererbungen beinhalten die folgende Funktionalität:

1. Aus einer Liste von übergebenen Attributen mit `(Name,Wert)` eines Xaml-Knotens eine Liste von Wert-Attributknoten (`PropertyValueNode`'s) machen, wobei die dort eingetragenen Werte bereits in C#-Notation konvertiert sein sollen.
2. Generierung eindeutiger Variablennamen über einen fortlaufenden Index.
3. Führen einer Liste von Attributen, die in C# Collections darstellen.

`NodeGenerator` ist eine abstrakte Klasse, welche durch konkrete Implementierungen vererbt werden muss. Die Idee hinter diesem Konzept ist, für jedes durch den XamlKonverter zu behandelnde GUI-Element (`Button`, `Grid`, etc.) eine eigene konkrete `NodeGenerator`-

Vererbung zu erzeugen. Für die Basisklassen der GUI-Elemente (z.B. `ButtonBase`, `ContentControl`, `Control` etc. bei `Button`) werden weiterhin abstrakte `NodeGenerator`-Klassen erstellt, wobei diese entsprechend ihrer Hierarchie in C# auch im XamlKonverter angeordnet werden. Folgendes Klassendiagramm zeigt ein Beispiel dieser Hierarchieabbildung:



Vergleicht man die C#-Hierarchie der GUI-Elemente mit dem hier abgebildeten Klassendiagramm, so erkennt man direkt die 1:1-Abbildung. Nimmt man z.B. `Grid`, so lassen sich davon in C# konkrete Objekte erstellen. Die Basisklassen von `Grid` sind in dieser Reihenfolge `Panel`, `FrameworkElement` und `UIElement`. Diese Basisklassen sind in C# abstrakt ebenso wie hier im XamlKonverter.

Warum wird dieser Aufwand der Hierarchiebildung betrieben? Aus Gründen der größtmöglichen Vermeidung von Redundanzen! In C# sind viele Properties nicht direkt in den GUI-Elementen, sondern in ihren Basisklassen definiert. So ist es auch im XamlKonverter: die Behandlung der Attribute, für die eigene Baumknoten erstellt werden,

findet dort statt, wo diese auch in C# definiert sind. Beispielsweise besitzt `FrameworkElement` die Attribute „`Width`“ und „`Height`“. Sollen nun die Attribut-Baumknoten für ein `Grid` erstellt werden, welches „`Width`“ und „`Height`“ definiert hat, so findet der erste Aufruf der Attributknoten-Erzeugung direkt für `Grid` statt. Nur Attribute, welche direkt in `Grid` definiert sind werden hier behandelt. Aus dieser Methode wird dann für die restlichen Attribute die Basisklassen-Methode (in `BaseGenPanel`) aufgerufen, wo die Attribute behandelt werden, welche auch in C# in `Panel` als Properties definiert sind. So setzen sich die Aufrufe fort, bis keine Basisklasse mehr existiert, welche die Methode implementiert. Dann sollten alle Attribute abgearbeitet sein, ansonsten wird ein Fehler in das Log geschrieben.

Die Hierarchie verhilft also dazu, dass Attribute in Basisklassen von Objekten nicht mehrfach behandelt werden müssen. Die konkreten Node-Generatoren sind auch dafür zuständig, automatisch Variablennamen zu erzeugen und in `ChildElementNode`’s des Baums zu setzen. Jeder `ChildElementNode` besitzt einen `NodeGenerator`, mit welchem er erzeugt wurde und ist so anhand dessen eindeutig bestimmt.

5 Schnittstellen für NodeGenerator

Die folgenden Schnittstellen können von abgeleiteten `NodeGenerator`-Klassen implementiert werden, um Ihnen spezielle Funktionalität hinzuzufügen. Die Baumknoten greifen auf diese Funktionalität zu, falls es der Xaml-Code erfordert.

Die zentrale Definition innerhalb der `NodeGenerator`-Hierarchie erlaubt nicht redundante Funktionalität, was die Vorteile dieser Hierarchie weiter unterstreicht.

ChildElementsAddHandler

Implementiert ein `NodeGenerator` diese Schnittstelle, so kann der zugehörige Xaml-Knoten weitere Unterknoten enthalten, die im C#-Code dem Objekt auf bestimmte Art hinzugefügt werden. Wie dies geschieht wird durch die Methode „`GenerateAddCode()`“ der `ChildElementsAddHandler` Schnittstelle geregelt. Diese Methode nimmt den im Baum definierten Elternknoten und ihm hinzuzufügende Kindknoten entgegen und erzeugt die Anweisungen, welche den Hinzufüge-Code generieren. Dabei kann über die Node-Generatoren der Kinder unterschieden werden, wie welche Kindknoten hinzugefügt werden sollen.

Beispiel:

Xaml:

```
<Button>
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color = “Blue“ />
      <GradientStop Color = “Red” />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

C#:

```
Button button1 = new Button();
LinearGradientBrush brush1 = new LinearGradientBrush();
GradientStop gs1 = new GradientStop();
gs1.Color = Colors.Blue;
GradientStop gs2 = new GradientStop();
```

```
gs2.Color = Colors.Red;
brush1.GradientStops.Add(gs1);
brush1.GradientStops.Add(gs2);
button1.Background = brush1;
```

In der NodeGenerator-Implementierung für LinearGradientBrush:

```
List<Statement> GenerateAddCode(ChildElementNode p, List<ChildElementNode> ch)
{
  List<Statement> addCode = new List<Statement>();
  foreach (ChildElementNode c in ch)
  {
    if (c.NodeGenerator is GenGradientStop)
      addCode.Add(new MethodCall(p.VarName + ".GradientStops.Add", c.VarName));
  }
  return addCode;
}
```

Der `NodeGenerator` für `LinearGradientBrush` würde hier `ChildElementsAddHandler` implementieren. In der `GenerateAddCode()`-Methode wird gesagt, wie Kindelemente zum Vater (einem `LinearGradientBrush`) hinzugefügt werden sollen und der entsprechende Code wird zurück gegeben. In diesem Fall werden Knoten mit einem `NodeGenerator` für `GradientStop` behandelt, für sie wird der entsprechende Code erzeugt.

TextAsAttribHandler

Diese Schnittstelle definiert für einen `NodeGenerator`, dass XmlText, der unter diesem Element in Xaml angegeben ist, als Attribut des zugehörigen Knotens aufzufassen ist. Dazu gibt es die Methode „`GetTextAttribName()`“, welche zu dem `NodeGenerator` zurückgibt, als welches Attribut bzw. als welche Property definierter XmlText behandelt werden soll.

Beispiel:

Xaml:

```
<Button>Drück mich</Button>
```

C#:

```
Button button1 = new Button();
button1.Content = “Drück mich”;
```

In diesem Fall würde der `NodeGenerator` für Button bzw. für die Basisklasse `ContentControl` die Schnittstelle `TextAsAttribHandler` implementieren, wobei von `GetTextAttribName()` der `string` „`Content`“ zurück gegeben wird. Dadurch wird definiert, dass angegebener XmlText wie das Attribut „`Content`“ behandelt werden soll. Durch Implementierung der Schnittstelle im Basis-`NodeGenerator` von `ContentControl` steht diese Funktionalität allen vererbten Klassen wie dem `NodeGenerator` für Button zur Verfügung und muss nicht mehrfach angegeben werden.

TextAsChildItemHandler

Diese Schnittstelle definiert für einen `NodeGenerator`, dass untergeordneter XmlText eines Xaml-Elements als Attribut eines eigenständigen Kindelements zu behandeln ist. Dazu ist über die Methoden „`GetItemName()`“ und „`GetAttribName()`“ anzugeben, wie dieses Kindelement und darin das Attribut heißen, in dem der XmlText als Wert zu setzen ist.

Beispiel:
Xaml:
```
<ListBox>Element</ListBox>
```
C#:
```
ListBox listBox1 = new ListBox();
ListBoxItem listBoxItem1 = new ListBoxItem();
listBoxItem1.Header = “Element”;
listBox1.Items.Add(listBoxItem1);
```

In diesem Beispiel würde der `NodeGenerator` für `ListBox` die Schnittstelle `TextAsChildItemHandler` implementieren, wobei `GetItemName()` den `string` „`ListBoxItem`“ und `GetAttribName()` den `string` „`Header`“ zurückgibt. Dadurch wird definiert, dass XmlText als Attribut „`Header`“ im eigenständigen Element „`ListBoxItem`“ zu behandeln ist. Im erzeugten Code wird demzufolge ein neues `ListBoxItem` erzeugt, seine `Header`-Eigenschaft gesetzt und zur `ListBox` hinzugefügt.

6 Einfügen neuer Xaml-Elemente in den Konverter

Die Struktur des XamlKonverter’s ist so ausgelegt, dass sich neue Xaml-Elemente und Klassen sehr schnell hinzufügen lassen, ohne dass die eigentliche Konverterlogik angepasst werden müsste.

Beim Einfügen neuer Elemente muss nur die als C#-Abbildung ansehbare `NodeGenerator`-Hierarchie erweitert werden. D.h. für neue Elemente werden neue `NodeGenerator`-Objekte angelegt und dem `NodeGeneratorManager` bekannt gemacht. Fortan stehen die neuen Elemente im XamlKonverter zur Verfügung und können aus Xaml in C# übersetzt werden.

Schritte

- **Schritt 1:** Erforschen der C#-Hierarchie des einzufügenden Elements
- **Schritt 2:** Basis-`NodeGenerator`-Klassen anlegen
- **Schritt 3:** `NodeGenerator`-Klasse des Elements erstellen
- **Schritt 4:** Element dem `NodeGeneratorManager` bekannt machen

Beispiel

Im folgenden Beispiel soll das GUI-Element „`CheckBox`“ in den XamlKonverter aufgenommen werden. Dabei wird davon ausgegangen, dass bereits die Hierarchie existiert, wie sie im obigen Klassendiagramm angegeben ist.

- **Schritt 1:** Erforschen der C#-Hierarchie des einzufügenden Elements
  Um heraus zu finden, welche Basisklassen `CheckBox` besitzt, schreibt man in einer beliebigen Methode „`System.Windows.Controls.CheckBox`“. Diese Zeile kann/muss nach Schritt 1 wieder gelöscht werden. Mit der rechten Maustaste klickt man auf `CheckBox` und wählt „Gehe zu Definition“. In der Definition sieht man nun, dass `CheckBox` von der Basisklasse `ToggleButton` erbt. Bei `ToggleButton` geht man wieder auf „Gehe zu Definition“ und erkennt nun, dass `ToggleButton` weiterhin von `ButtonBase` erbt. Für `ButtonBase` existiert allerdings bereits ein Basis-`NodeGenerator`.

Somit ist das Erforschen der C#-Hierarchie hier bereits abgeschlossen. Das Ergebnis: 1.) für die abstrakte Basisklasse `ToggleButton` ist ein abstrakter Basis-`NodeGenerator` zu erstellen, der vom existierenden `NodeGenerator` `BaseGenButtonBase` erbt. 2.) für das GUI-Element `CheckBox` ist ein konkreter `NodeGenerator` zu implementieren, der vom `ToggleButton-NodeGenerator` erbt.

- Schritt 2: Basis-`NodeGenerator`-Klassen anlegen

Für `ToggleButton` muss ein abstrakter `NodeGenerator` angelegt werden, gemäß Namenskonvention `BaseGenToggleButton`. Zunächst wird der Klassenkopf erstellt:

```
abstract class BaseGenToggleButton : BaseGenButtonBase
{
  public BaseGenToggleButton() : base()  { }

  protected override List<PropertyValueNode>
      GenerateBaseNodes(List<ElemAttribute> attribs)
  {
    throw new NotImplementedException();
  }

  public override bool HasTagInHierarchy(string tag)
  {
    throw new NotImplementedException();
  }
}
```

Jeder Basis-`NodeGenerator` muss die Methode `GenerateBaseNodes()` implementieren, welche die übergebenen Attribute behandelt und daraus Attribut-Baumknoten erstellt. Außerdem muss `HasTagInHierarchy()` implementiert werden, um eine Aussage darüber treffen zu können, ob ein Xaml-Tag im Pfad zur Wurzel der `NodeGenerator`-Hierarchie enthalten ist.

Jetzt muss die Funktionalität noch implementiert werden. Dazu kann man bereits vorhandene Basis-`NodeGenerator`-Klassen betrachten und deren Implementation kopieren sowie anpassen.
`GenerateBaseNodes()` verarbeitet die Attribute/Properties eines `ToggleButton`. D.h. in C# ermittelt man durch das Gehen zur Definition von `ToggleButton` (wie in Schritt 1), welche Properties die Klasse `ToggleButton` beinhaltet. Dort erkennt man, dass `ToggleButton` 2 Properties hat: `IsChecked` und `IsThreeState`, beide `bool`-Werte und beide get/set-Properties. Wichtig: es müssen nur die get/set-Properties betrachtet werden, get allein reicht nicht, da sie in Xaml als auch in C# nicht gesetzt werden können und somit keine Rolle spielen. Die Implementierung von `GenerateBaseNodes()` sieht so aus:

```
protected override List<PropertyValueNode>
    GenerateBaseNodes(List<ElemAttribute> attribs)
{
  List<PropertyValueNode> nodes = new List<PropertyValueNode>();
  List<ElemAttribute> unhandled = new List<ElemAttribute>();
  foreach (ElemAttribute attrib in attribs)
  {
    switch (attrib.Name)
    {
      case "IsChecked":
      case "IsThreeState":
        nodes.Add(new PropertyValueNode(attrib.Name,
          AttBoolConverter.Convert(attrib.Value)));
        break;
      default:
        unhandled.Add(attrib);
        break;
    }
```

```
    }
    nodes.AddRange(base.GenerateBaseNodes(unhandled));
    return nodes;
  }
```

Kurz erklärt: es werden alle Attribute betrachtet. Sollte der Attributname „`IsChecked`“ oder „`IsThreeState`“ sein, so wird ein neuer Attributknoten erzeugt mit dem Namen des Attributs und dem konvertierten Attributwert. Da es sich bei den Attributen um `bool`-Werte handelt, wird die `AttBoolConverter`-Klasse verwendet, um einen C#-konformen `string` zu erhalten. Bei einigen Attributen (hier nicht) muss teilweise als 3. Argument des `PropertyValueNode`-Konstruktors noch eine Liste von Usings übergeben werden, welche das Attribut benötigt. Diese erhält man im Fall der Fälle durch ein dann verfügbares Property der Attribut-Konverter-Klasse. In der vorletzten Zeile findet der Aufruf der noch unbehandelten Attribute für die Basisklasse statt (hier: `ButtonBase`), dessen Ergebnis an die aktuelle Knotenliste angehangen und zurück gegeben wird.

Die Implementierung von `HasTagInHierarchy()` gestaltet sich einfacher:

```
public override bool HasTagInHierarchy(string tag)
{
  if ("ToggleButton" == tag)
    return true;
  return base.HasTagInHierarchy(tag);
}
```

Entweder handelt es sich beim aktuellen Element (`ToggleButton`) bereits um das gesuchte Tag oder es befindet sich weiter oben in der Hierarchie (rekursiver Aufruf für die Basisklasse) oder es ist nicht in der Hierarchie enthalten (rekursiver Aufruf liefert `false` zurück).

- **Schritt 3:** `NodeGenerator`-Klasse des Elements erstellen

Für den konkreten `NodeGenerator` der `CheckBox` mit dem Namen `GenCheckBox` wird ebenfalls zunächst der Basiscode erstellt. Von konkreten Node-Generatoren müssen auf jeden Fall der Konstruktor und die beiden Methoden `GenerateAttribNodes()` und `HasTagInHierarchy()` implementiert werden:

```
class GenCheckBox : BaseGenToggleButton
{
  public GenCheckBox() : base()
  {
    throw new NotImplementedException();
  }

  public override List<PropertyValueNode>
    GenerateAttribNodes(List<ElemAttribute> attribs)
  {
    throw new NotImplementedException();
  }

  public override bool HasTagInHierarchy(string tag)
  {
    throw new NotImplementedException();
  }
}
```

Im Konstruktor muss definiert werden, wie die Klasse/das Xaml-Tag heißt, für welche dieser `NodeGenerator` zuständig sein soll. Außerdem müssen der Basisname für automatisch erstellte Variablen gesetzt und eine Liste von Usings angegeben werden, welche von diesem Element im C#-Code benötigt werden (diese Elemente sind als private Felder in `NodeGenerator` definiert):

```
public GenCheckBox() : base()
```

```
{
  tagName = "CheckBox";
  varBaseName = "checkBox";
  usings = new List<UsingStmt>()
    { new UsingStmt("System.Windows.Controls") };
}
```

In `GenerateAttribNodes()` werden die Attribute/Properties behandelt, welche direkt in `CheckBox` definiert sind. Dazu muss man zur Definition von `CheckBox` (wie in Schritt 1 bzw. 2) gehen und nachschauen, welche get/set-Properties `CheckBox` hat. Und das ist hier einfach: es ist erkennbar, dass `CheckBox` keine get/set-Properties besitzt, somit der Aufruf direkt an die Basisklasse weiter gegeben werden kann:

```
public override List<PropertyValueNode>
    GenerateAttribNodes(List<ElemAttribute> attribs)
{
  return base.GenerateBaseNodes(attribs);
}
```

In `HasTagInHierarchy()` wird nur geprüft ob das aktuelle Tag dem übergebenen entspricht und ansonsten der Aufruf an die Basisklasse weiter gereicht:

```
public override bool HasTagInHierarchy(string tag)
{
  if (this.tagName == tag)
    return true;
  return base.HasTagInHierarchy(tag);
}
```

- Schritt 4: Element dem `NodeGeneratorManager` bekannt machen

Hat man die neuen Node-Generatoren erstellt, so müssen sie dem XamlKonverter nur noch bekannt gemacht werden. Die Unterscheidung, welcher `NodeGenerator` für welchen Baumknoten zuständig ist findet einzig und allein im `NodeGeneratorManager` statt, sodass nur dort das `Dictionary` verwalteter Node-Generatoren im Konstruktor um `GenCheckBox` erweitert werden muss:

```
addGenerators(new List<NodeGenerator>(){
  new GenButton(),
  ...
  new GenCheckBox()
});
```

- Schritt 5: Testen und freuen :-)

Jetzt darf getestet werden. Z.B. ist nun folgendes Beispiel übersetzbar.

Beispiel:

Xaml:

```
<Grid Background="White"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
  <CheckBox IsChecked="true" Width="150" Height="20">
    CheckBox-Fun
  </CheckBox>
</Grid>
```

C#:

```
using System;
using System.Text;
using System.Windows.Controls;
using System.Windows.Media;

class GUI {
    public GUI() {
        InitializeComponent();
    }
```

```
    void InitializeComponent(){
        Grid grid1 = new Grid();
        grid1.Background = Brushes.White;
        CheckBox checkBox1 = new CheckBox();
        checkBox1.IsChecked = true;
        checkBox1.Content = "CheckBox-Fun";
        checkBox1.Width = 150;
        checkBox1.Height = 20;

        grid1.Children.Add(checkBox1);
    }
}
```

Na das sieht doch richtig gut aus! `IsChecked` wurde vernünftig gesetzt, doch nicht nur das: alle Attribute, die in den Basisklassen behandelt werden, können direkt hier behandelt werden, ohne dass weitere Änderungen erforderlich werden. Diese leichte Erweiterbarkeit zeigt wunderbar das Potential des umgesetzten Konzepts.