Dieser Abschnitt erweitert den Buchabschnitt der Abbildung der Zustandsautomaten in Code um die komplexeren Notationsmittel. Den kompletten Code für das Framework und für mehrere Beispiele können Sie auch als ein Microsoft Visual Studio .NET Projekt auf *www.uml-glasklar.com* unter Downloads herunterladen.

Zusammengesetzte Zustände

Durch die Einführung der zusammengesetzten Zustände müssen wir unser Framework erweitern. Das zu realisierende Metamodell ist in Abbildung A dargestellt. Die entscheidende Erweiterung besteht in der Einführung der Klasse `Region`.

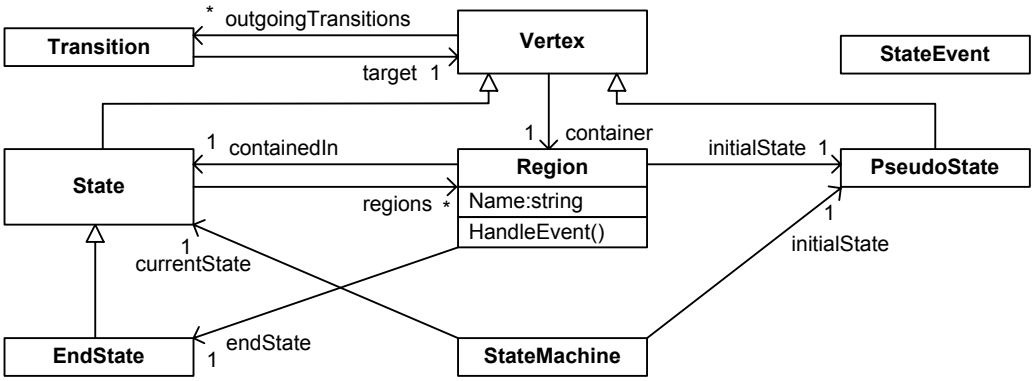


Abbildung A: **Das Metamodell für zusammengesetzte Zustandsautomaten**

Region

Die Klasse `Region` schafft die Verbindung zwischen den Zuständen und deren Unterzuständen. Sie sorgt dafür, dass die in ihr befindlichen Zustände wissen, zu welchem umfassenden Zustand sie gehören. Das stellt sicher, dass in den Methoden für die Eventverarbeitung (`HandleEvent()`), für das Betreten und Verlassen der Zustände (`EnterState(), LeaveState()`) das Wissen vorhanden ist, ob sie Aufrufe zu den übergeordneten Zuständen weiterleiten müssen.

14.6.5.1

Die Klasse Region enthält auch die Informationen über die in sich befindlichen Startzustände, Endzustände und History-Pseudozustände, auf die wir später eingehen werden.

```
public class Region {
    public Region(string name) {
        this.name = name;}
    private string name;
    public string Name {
        get {return name;}}
    // der Zustand in dem sich die Region befindet
    private State container = null;
    public State Container {
        get {return container;}
```

```
            set {container = value;}}
        // Jede Region kann einen Startzustand haben
        private PseudoState initialState = null;
        public PseudoState InitialState {
            get {return initialState;}
            set {initialState = value;}}
        // Jede Region kann einen Endzustand haben
        private EndState endState = null;
        public EndState GetEndState {
            get {return endState;}
            set {endState = value;}}
        /* Der Trigger in form von StateEvent wird hiermit zu dem
        darüberstehenden Zustandweitergeleitet */
        public void HandleEvent(StateEvent e) {
            if (container != null)
                container.HandleEvent(e);}
        // Berechnet den Pfad zu diesem Region von außen nach innen
        public ArrayList PathToRegion() {
            ArrayList tmpReturn;
            if (this.container != null)
                 tmpReturn = container.ParentRegion.PathToRegion();
            else
                tmpReturn = new ArrayList();
            tmpReturn.Add(this);
            return tmpReturn;}
        private bool hasShallowHistory = false;
        public bool HasShallowHistory {
            get {return hasShallowHistory;}
            set {hasShallowHistory = value;}}
        private bool hasDeepHistory = false;
        public bool HasDeepHistory {
            get {return hasDeepHistory;}
            set {hasDeepHistory = value;}}
    }
```

Vertex


14.6.3.1

Die Klasse `Vertex` wird so erweitert, dass jedes Objekt dieser Klasse eine Referenz auf die Region hat, in der sie sich befindet.

```
    protected Region parentRegion;
    public Region ParentRegion {
        get {return parentRegion;}}
```

Der Konstruktor und die Methoden `EnterState()` und `LeaveState()` müssen so umdefiniert werden, dass sie die dem Vertex zugeordnete Region als Parameter erhalten.

```
public Vertex(string name, Region parentRegion) {
    this.name = name;
    this.parentRegion = parentRegion;}
virtual public void EnterState(Region sourceRegion, StateEvent e) {}
virtual public void LeaveState(Region targetRegion, StateEvent e) {}
```

SOPHIST GROUP

Die Bearbeitung eines Ereignisses muss nun neu definiert werden. In der Hierarchie von Zuständen wird zunächst die unterste Ebene betrachtet. Wird das Ereignis dort nicht konsumiert, wird es an den übergeordneten Zustand gegeben. Wenn die oberste Ebene erreicht ist und das Ereignis dort auch nicht konsumiert wird, wird es verworfen.

```
// in der Klasse Vertex
virtual public void HandleEvent(StateEvent e) {
    Transition transition = getTransitionForEvent (e);
    if (transition != null)
        transition.Transit(this, e);
    else {
        allEvent = new StateEvent("all", e.Receiver));
        transition = getTransitionForEvent(allEvent);
        if (transition != null)
            transition.Transit(this, allEvent);
        else
            parentRegion.HandleEvent(e); }}
```

Zustand

Die Zustände besitzen nun eine Region, in der sich die Unterzustände befinden.

14.6.3.2

```
// in Klasse State
private Region region = null;
public void SetRegion(Region region) {
    this.region = region;
    this.region.Container = this;}
/* Manchmal ist es hilfreich zu wissen,
in welchem übergeordneten Zustand sich
ein Unterzustand befindet. */
public State ParentState {
    get {return parentRegion.Container;}}
```

Die Zustände unterstützen die in der Klasse `Vertex` definierte Abarbeitung der Ereignisse dadurch, dass sie die Methoden zum Durchlaufen der Zustandshierarchie zur Verfügung stellen. Dabei muss darauf geachtet werden, dass nur jene Regionen betrachtet werden, die beide an der Transition beteiligten Zustände beinhalten.

Zusätzlich muss beim Betreten eines Zustands nicht nur das Entry-Verhalten dieses Zustands, sondern zuvor das Entry-Verhalten des darüber liegenden Zustands ausgeführt werden.

Hier kommt die Art des Betretens (Explicit- und DefaultEntry, siehe 14.4.9) ebenfalls zum Tragen. Erreicht man direkt einen untergeordneten Zustand oder wird der zusammengesetzte Zustand über den Startzustand betreten?

14.4.9

```
public override void EnterState(Region sourceRegion,
    StateEvent e) {
    // Überliegende Zustände werden betreten
    if ((base.parentRegion != sourceRegion) &&
        (ParentState != null))
        ParentState.EnterState(sourceRegion, e);
    // Das Eintrittsverhalten wird ausgeführt
    if (enter != null)
        enter(e.Classifier);
```

```
        // Das Zustandsverhalten wird gestartet
        e.Receiver.StartDoActivities(this.Name);
        /* Der Zustand wird als Aktiv gesetzt,
        falls er sich ganz unten in der Hierarchie befindet */
        if (region == null)
            e.Receiver.ConnectCurrentState(this);
        /* Falls der Zustand eine Region besitzt,
        und die Transition auf diesen Zustand zeigt,
        dann wird sie mittels eines Startzustands betreten */
        if ((e.LastTarget == this) && (region != null))
            region.InitialState.EnterState(null,
                new StateEvent("Start", e.Receiver));}
```

`LastTarget` ist ein Zeiger auf den Zustand auf dem die Transition zeigt, und wird in `StateEvent` definiert.

```
    private Vertex lastTarget = null;
    public Vertex LastTarget {
        get {return lastTarget;}
        set {lastTarget = value;}
```

Beim Verlassen eines Zustandes wird genau umgekehrt vorgegangen: von unten nach oben. Alle Zustände, angefangen von ganz unten, müssen verlassen werden.

```
    public override void LeaveState(Region targetRegion,
        StateEvent e) {
        /* falls dies der unterste Zustand ist,
        wird er als inaktiv gesetzt */
        if (region == null)
            e.Receiver.DisconnectCurrentState(this);
        /* Falls der Zustand eine Region besitzt,
        dann wird erst der unterste Zustand verlassen.
        Nachdem die Hierarchie durchlaufen ist, dann ist
        auch dieser Zustand schon verlassen */
        else if (e.Receiver.CurrentState != null) {
            e.Receiver.CurrentState.LeaveState(targetRegion, e);
            return; }
        // Do Aktivitäten werden gestoppt
        e.Receiver.StopDoActivities(this.Name);
        // Austrittsverhalten wird ausgeführt
        if (exit != null)
            exit(e.Classifier);
        // Verzögerte Trigger wandern zurück in die Trigger Schlange
        e.Receiver.ReAddDeferred();
        /* Falls die Transition regionübergreifend geschieht,
        dann wird auch der umfassende Zustand verlassen */
        if ((parentRegion != targetRegion) && (ParentState != null))
            ParentState.LeaveState(targetRegion, e);}
```

SOPHIST GROUP

Transition

Eine neue `TransitionKind` wird jetzt eingeführt: `Extern`. Sie bedeutet, dass mit der Transition eine oder mehrere Regionen betreten oder verlassen werden. 14.6.3.3

Bei dieser Art der Transition müssen folgende Dinge geschehen. Das Ziel der Transition muss gemerkt, die gemeinsame Region des Ursprungs- und Zielzustands berechnet (`CommonRegion()`) und an die Methoden `LeaveState()` des Ursprungszustands und `EnterState()` des Zielzustands übergeben werden.

```
// in Transition.Transit()
case TransitionKind.Extern:
    e.LastTarget = target;
    Region commonRegion =
        CommonRegion(source.ParentRegion, target.ParentRegion);
    source.LeaveState(commonRegion, e);
    if (!TargetIsJunction)
        ExecuteActivity(e);
    target.EnterState(commonRegion, e);
    break;

/* CommonRegion() berechnet die tiefste gemeinsame Region,
in der sich sourceRegion und targetRegion befinden */
public static Region CommonRegion(Region sourceRegion,
    Region targetRegion) {
    if (sourceRegion == targetRegion)
        return sourceRegion;
    ArrayList pathToSource = sourceRegion.PathToRegion();
    ArrayList pathToTarget = targetRegion.PathToRegion();
    for (int i = Math.Min(pathToSource.Count,
        pathToTarget.Count) - 1; i >= 0 ; i--)
        if ((Region)pathToSource[i] == (Region)pathToTarget[i])
            return (Region)pathToSource[i];
    return (Region)pathToSource[0];}
```

Endzustand

Der Konstruktor muss minimal angepasst werden, um die entsprechende Referenz in der Region zu setzen.

```
public EndState(string name, Region parentRegion):
    base(name, null, null, parentRegion) {
    parentRegion.EndState = this;}
```

Wenn in einem Unterzustand ein Endzustand erreicht wird, soll der umgebende Zustand verlassen werden, falls eine Transition ohne Trigger von dort wegführt. In dem Framework wird dies mit einem Pseudotrigger (ein Trigger, der in einem Zustandsautomaten nicht existiert, aber in der Realisierung irgendwie bezeichnet werden muss) "`RegionEndState`" erreicht. Das heißt, dass der Transition der Trigger "`RegionEndState`" zugewiesen werden muss, um das Verlassen des zusammengesetzen Zustandes nach dem Erreichen des in ihm erhaltenen Endzustands zu gewährleisten. Wenn der Endzustand erreicht ist, wird ein Ereignis 14.4.9

mit diesem Trigger erzeugt und am Anfang der `eventQueue` hinzugefügt. Dazu wird in der Klasse `StateMachine`, in der die `eventQueue` definiert ist, die Methode `AddFirstEvent()` definiert.

```
// in Klasse EndState
public override void EnterState(Region sourceRegion, StateEvent e) {
    e.Receiver.ConnectCurrentState(this);
    e.Receiver.AddFirstEvent
        (new StateEvent("RegionEndState", e. Receiver));}

// in Klasse StateMachine
public void AddFirstEvent(StateEvent e) {
    eventQueue.Insert(1, e);}
```

PseudoState


14.6.3.5

Der Konstruktor muss angepasst werden, um der Region einen Startzustand zuweisen zu können.

```
    public PseudoState(string name, PseudoStateKind pseudoStateKind,
        Region parentRegion) : base(name, parentRegion)
        switch (pseudoStateKind) {
            case PseudoStateKind.Initial:
                parentRegion.InitialState = this;
                break;}}
```

History-Pseudozustände


14.4.14

Mit den Unterzuständen müssen auch die Pseudozustände `ShallowHistory` und `DeepHistory` eingeführt werden. Die History-Pseudozustände gehören zu einer Region und können in einer Region nur einmal vorkommen. Dies gewährleistet eine eindeutige Zuordnung eines History-Pseudozustandes zu einer Region. Diese Eindeutigkeit ist wichtig, da beim Verlassen einer Region der zuletzt aktive Zustand gespeichert werden muss.

History Helper

Die Hilfsklasse `HistoryHelper` wird definiert, um die Zustände zu verwalten, auf welche die History-Pseudozustände zeigen. Dort wird die Region, in der sich ein History-Pseudozustand befindet, zusammen mit dem zuletzt aktiven Zustand gespeichert..

```
    public class HistoryHelper {
        public HistoryHelper(Region lastRegion, Vertex lastState) {
            this.lastRegion = lastRegion;
            this.lastState = lastState;}
        private Region lastRegion;
        public Region LastRegion {
            get {return lastRegion;}
            set {lastRegion = value;}}
        private Vertex lastState;
        public Vertex LastState {
            get {return lastState;}
            set {lastState = value;}}
    }
```

SOPHIST GROUP

PseudoState

In dem Konstruktor der Klasse `PseudoState` wird die Information über die History bzw. deren Präsenz für die übergreifende Region gesetzt.


14.6.3.5

```
// in PseudoState.PseudoState()
case PseudoStateKind.History:
    parentRegion.HasShallowHistory = true;
    break;
case PseudoStateKind.DeepHistory:
    parentRegion.HasDeepHistory = true;
    break;
```

Die Anfrage, die History zu betreten, wird an den entsprechenden Zustandsautomaten geschickt, wo die Anfrage verarbeitet wird.

```
// in PseudoState.EnterState()
case PseudoStateKind.History:
    e.Receiver.EnterShallowHistoryState(sourceRegion,
e,parentRegion);
    break;
case PseudoStateKind.DeepHistory:
    e.Receiver.EnterDeepHistoryState(sourceRegion, e, parentRegion);
    break;
```

State

Beim Verlassen des Zustandes wird der entsprechende Zustand gespeichert. Dies übernimmt die `StateMachine`.


14.6.3.2

```
// in State.LeaveState()
if (parentRegion.HasShallowHistory)
    e.Receiver.AddShallowHistoryState(parentRegion, this);
if (parentRegion.HasShallowHistory)
    e.Receiver.AddDeepHistoryState(parentRegion);
```

StateMachine

Da jedes Objekt, dessen Verhalten von einem Zustandsautomaten beschrieben wird, die Region auf eine unterschiedliche Weise verlassen kann, können wir den letzten Zustand nicht direkt bei dem History-Pseudozustand speichern. Instanzen der Klasse `StateMachine` übernehmen diese Aufgabe, da diese bekanntlich speziell sindfür einzelne, durch sie beschriebene Objekte.


14.6.4.7

```
private ArrayList shallowHistory = new ArrayList();
private ArrayList deepHistory = new ArrayList();
```

Beim Verlassen einer Region, in der sich ein History-Pseudozustand befindet, wird ein Zustand in einer History-Liste gespeichert. Bei der `ShallowHistory` wird der

Zustand gespeichert, der sich in derselben Region wie der `ShallowHistory`-Pseudozustand befindet. Bei `DeepHistory` wird der tiefste Zustand gespeichert. Da in unserem Framework der aktive Zustand immer in der Zustandshierarchie der tiefste Zustand ist, kann direkt der aktive Zustand gespeichert werden.

```
public void AddShallowHistoryState(Region region, Vertex state){
    foreach (HistoryHelper h in shallowHistory)
        if (h.LastRegion == region) {
            h.LastState = state;
            return;}
    shallowHistory.Add(new HistoryHelper(region, state));}
public void AddDeepHistoryState(Region region) {
    foreach (HistoryHelper h in deepHistory)
        if (h.LastRegion == region) {
            h.LastState = lastCurrentState;
            return;}
    deepHistory.Add(new HistoryHelper(region,
        lastCurrentState));}
```

Beim Betreten einer Region mit einem History-Pseudozustand wird zunächst nachgesehen, ob die History-Liste einen passenden Eintrag für die entsprechende Region hat. Falls nicht, wird die Region mit dem Startzustand gestartet. Falls sich der Eintrag in der Liste befindet, wird der Zustand betreten, der zu diesem Eintrag gehört. Diese Methoden werden aus `PseudoState.EnterState()` aufgerufen.

```
public void EnterShallowHistoryState(Region sourceRegion,
    StateEvent e, Region region) {
    foreach (HistoryHelper h in shallowHistory)
        if (h.LastRegion == region) {
            e.LastTarget = h.LastState;
            h.LastState.EnterState(sourceRegion, e);
            return;}
    e.LastTarget = region.Container;
    region.Container.EnterState(sourceRegion, e);}

public void EnterDeepHistoryState(Region sourceRegion,
    StateEvent e, Region region) {
    foreach (HistoryHelper h in deepHistory)
        if (h.LastRegion == region) {
            e.LastTarget = h.LastState;
            h.LastState.EnterState(sourceRegion, e);
            return;}
    e.LastTarget = region.Container;
    region.Container.EnterState(sourceRegion, e);}
```

Parallelität


14.4.11

Parallelität bei Zustandsautomaten setzt voraus, dass ein Zustand mehrere Regionen besitzt. Als Folge daraus kann es mehrere aktive Zustände in einem Zustandsautoma-

ten geben. Dies zieht Änderungen in der Klasse `State` nach sich. Das Attribut `region` und die Methode `SetRegion()` werden gelöscht und entsprechend der Tatsache, dass es jetzt mehrere Regionen in einem Zustand enthalten sein können, wird das Attribut `regionList` und die Methode `AddRegion()` eingefügt.

```
private ArrayList regionList = new ArrayList();
public void AddRegion(Region region) {
    regionList.Add(region);
    region.Container = this;}
public ArrayList GetRegions {
    get {return regionList;}}
```

Auch in `State.EnterState()`müssen einige Änderungen vorgenommen werden, um mehrere Unterregionen zu behandeln. Die Methoden `StateEvent.IsHandled()` und `StateEvent.SetHandled()` werden benutzt, um jeden Zustand nur einmal zu behandeln. Außerdem werden sie benutzt, um festzustellen, ob die Abarbeitung des Events an die übergreifenden Zustände weiter geschickt werden soll.

Alle Regionen, die in einem Zustand enthalten sind, müssen durch den entsprechenden Startzustand betreten werden, falls die Transition auf diesen Zustand zeigt.

```
public override void EnterState(Region sourceRegion, StateEvent e){
    if (e.IsHandled(this.name)) return;
    e.SetHandled(this.name);
    if ((base.parentRegion != sourceRegion)&&(ParentState != null))
        ParentState.EnterState(sourceRegion, e);
    if (enter != null)
        enter(e.Classifier);
    e.Receiver.StartDoActivities(this.Name);
    if (regionList.Count == 0)
        e.Receiver.ConnectCurrentState(this);
    if ((e.LastTarget == this) && (regionList.Count != 0))
        foreach (Region region in regionList)
            region.InitialState.EnterState(null,
                new StateEvent("Start", e.Receiver));}
```

Entsprechende Änderungen müssen in `State.LeaveState()` vorgenommen werden. Beim Verlassen des Zustandes müssen alle aktiven Unterzustände auch verlassen werden.

```
if (e.IsHandled(this.name))
    return;
e.SetHandled(this.name);
if (regionList.Count == 0)
    e.Receiver.DisconnectCurrentState(this);
else if (e.Receiver.CurrentStates.Count != 0) {
foreach (Vertex state in e.Receiver.CurrentStates)
    if (Transition.CommonRegion(parentRegion,
        state.ParentRegion) == parentRegion)
        state.LeaveState(targetRegion, e);
e.Receiver.StopDoActivities(this.Name);
// weiter ohne Änderungen
```

Die Methoden `StateEvent.IsHandled()` und `StateEvent.SetHandled()` sehen in der Klasse `StateEvent` so aus:

```
private ArrayList handled = new ArrayList();
public void SetHandled(string stateName) {
    handled.Add(stateName);}
public bool IsHandled(string stateName) {
    foreach (string a in handled)
        if (a == stateName)
            return true;
    return false;}
```

Eine Änderung muss auch in `Vertex` vorgenommen werden, damit `StateEvent.IsHandled()` und `StateEvent.SetHandled()` benutzt werden können, um die Abarbeitung des Ereignisses in der Zustandshierarchie richtig zu behandeln. Das hängt damit zusammen, dass das Ereignis nur dann an den übergreifenden Zustand weiter geschickt werden soll, wenn keiner der Unterzustände auf das Ereignis reagiert. Wenn mindestens ein Unterzustand auf den Event reagiert, wird die Methode `StateEvent.SetHandled()` für den übergreifenden Zustand ausgeführt, damit er als schon behandelt gemerkt wird und das Ereignis nicht mehr bearbeitet.

```
public void ReactsToEvent(StateEvent e) {
    foreach (Transition transition in outgoingTransitions)
        if (transition.GuardTrue(e.Classifier) &&
            transition.ReactsToTrigger(e.Trigger) &&
            ParentState != null)
                e.SetHandled(ParentState.Name);}
```

Am Anfang von `Vertex.HandleEvent` muss geprüft werden, ob das Ereignis schon behandelt wurde:

```
if (e.IsHandled(this.name))
    parentRegion.HandleEvent(e);
```

Um mehrere aktive Zustände behandeln zu können, muss die Klasse `StateMachine` angepasst werden. Die aktiven Zustände werden in der Liste `currentStates` gespeichert. Damit sich der Zustandsautomat immer in einer richtigen Zustandskonfiguration befindet, muss der Zustandswechsel an einer zentralen Stelle immer zur gleichen Zeit geschehen. Dafür werden zwei weitere Listen definiert. In der Liste `lastCurrentStates` werden die Zustände gespeichert, die verlassen wurden. Die Liste `currentStatesToBe` beinhaltet alle Zustände, die während der Abarbeitung des Events betreten wurden.

```
private ArrayList currentStates = new ArrayList();
private ArrayList lastCurrentStates = new ArrayList();
private ArrayList currentStatesToBe = new ArrayList();
public void ConnectCurrentState(Vertex state) {
    currentStatesToBe.Add(state);}
public void DisconnectCurrentState(Vertex state) {
    lastCurrentStates.Add(state);}
public string CurrentStateName {
    get {
```

SOPHIST GROUP

```
            string ret = "";
            foreach (Vertex vertex in currentStates)
                ret += vertex.Name + ",";
                    return ret.Substring(0, ret.Length - 1);}}
    public ArrayList CurrentStates {
        get {return currentStates;}}
    public ArrayList CurrentStatesToBe {
        get {return currentStatesToBe;}}
```

Auch `StateMachine.DispatchEvent()` muss entsprechende angepasst werden.

```
    void DispatchEvent() {
        while (running) {
            StateEvent e = GetFirstEvent();
            if (e != null) {
                foreach (Vertex state in currentStates)
                    state.ReactsToEvent(e);
                foreach (Vertex state in currentStates)
                    if (!e.IsHandled(state.Name))
                        state.HandleEvent(e);
                eventQueue.RemoveAt(0);
                foreach (object o in lastCurrentStates)
                    currentStates.Remove(o);
                lastCurrentStates.Clear();
                currentStates.InsertRange(0, currentStatesToBe);
                currentStatesToBe.Clear();}}
```

Da die neuen Zustände erst in `currentStatesToBe` gesetzt werden, müssen sie in `StateMachine.StartStateMachine` sofort nach dem Betreten des Startzustandes in `currentStates` integriert werden:

```
    // in StateMachine.StartStateMachine()
    currentStates.InsertRange(0, currentStatesToBe);
    currentStatesToBe.Clear();
```

Auch die Methoden, die DeepHistory behandelt, müssen so angepasst werden, dass bei der Transition in eine DeepHistory alle Unterzustände betreten werden, die zum Zeitpunkt des Verlassens des Zustandes aktiv waren.

```
    // in StateMachine Klasse
    public void EnterDeepHistoryState(Region sourceRegion,
        StateEvent e, Region region) {
        foreach (HistoryHelper h in deepHistory)
            if (h.LastRegion == region) {
                if (h.LastState is EndState) {
                    e.LastTarget =
                        h.LastState.ParentRegion.InitialState;
                    h.LastState.ParentRegion.InitialState.
                        EnterState(sourceRegion, e); }
                else {
                    e.LastTarget = h.LastState;
                    h.LastState.EnterState(sourceRegion, e); }}
        e.LastTarget = region.Container;
        region.Container.EnterState(sourceRegion, e);}
```

```
public void AddDeepHistoryState(Region region)
{
    for (int i = deepHistory.Count - 1; i >= 0; i--)
        if (((HistoryHelper)deepHistory[i]).LastRegion==region)
            deepHistory.RemoveAt(i);
    foreach (Vertex state in currentStates)
        if (Transition.CommonRegion(region, state.ParentRegion)
            == region)
            deepHistory.Add(new HistoryHelper(region, state));}
```

Es ist sehr einfach parallele Zustandsautomaten zu bauen. Man muss nur zu jedem Zustand mehrere Regionen hinzufügen. Den Rest übernimmt der Framework.

Mit dem so definierten Framework ist es sehr einfach, parallele Zustandsautomaten zu konstruieren: Mehrere orthogonale Regionen werden einem Zustand zugewiesen. Die korrekte Einordnung und Definition übernimmt das Framework.

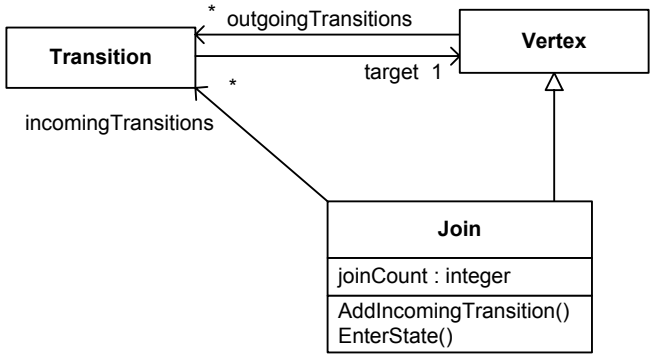
Gabelung und Vereinigung


14.4.10

Mit der Parallelität wird auch der Pseudozustand um zwei neue `PseudoState-Kind` erweitert: Gabelung (`Fork`) und Vereinigung (`Join`). Gabelung ist sehr einfach zu realisieren. Dafür wird `Pseudo-State.EnterState()` erweitert:

```
case PseudoStateKind.Fork:
    foreach (Transition transition in outgoingTransitions)
        transition.Transit(this, e);
    break;
```

Die Realisierung der Vereinigung aber bedarf viel mehr Arbeit. Da der Zustandsübergang an der Vereinigung nur in dem Fall geschieht, wenn alle Zustände, die eine Transition zu dieser Vereinigung haben, auch aktiv sind, müssen für die Vereinigung die eingehenden Transitionen explizit referenzierbar gemacht werden. Am besten erzeugt man dafür eine neue Klasse `Join`, die von `Vertex` abgeleitet ist. Diese Klasse erhält auch die eingehenden Transitionen damit die richtigen Zustände verlassen werden.




```
public class Join : Vertex {
    public Join(string name, Region parentRegion) :
        base(name, parentRegion){}
    ArrayList incomingTransitions = new ArrayList();
    public int joinCount = 0;
```

```
    public override void EnterState(Region sourceRegion,
        StateEvent e) {
        e.LastTarget = this;
        foreach (Transition transition in incomingTransitions)
            transition.Transit(transition.FixedSource, e);
        ((Transition)outgoingTransitions[0]).Transit(this, e); }
    public void AddIncomignTransition(Transition t) {
        incomingTransitions.Add(t);
        joinCount++; }  }
```

Da die Transitionen, die an die Vereinigung anschließen, etwas anderes behandelt müssen als die normalen Transitionen, müssen auch in der Klasse `Transition` einige Änderungen vorgenommen werden.

```
// in der Klasse Transition
private Vertex fixedSource = null;
public Vertex FixedSource {
    get {return fixedSource;}
    set {fixedSource = value;}}
// in Transition.Transit()
case TransitionKind.Extern:
    if (target == null)
        throw new Exception
            ("target für die Transition ist nicht definiert");
    Region commonRegion =
        CommonRegion(source.ParentRegion, target.ParentRegion);
    source.LeaveState(commonRegion, e);
    if (!TargetIsJunction && !(target is Join))
        ExecuteBehavior(e);
    e.LastTarget = target;
    if (!(target is Join)))
        target.EnterState(commonRegion, e);
    break;
```

In `StateMachine` werden die folgenden Änderungen nötig. Damit die Prüfung, ob die Transition an Vereinigung ausgeführt werden soll, einfacher geschieht, wird geprüft, ob die richtigen Zustände aktiv sind.

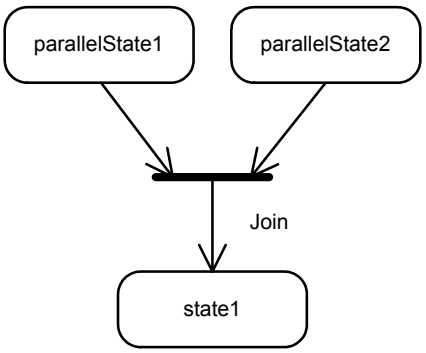
```
private static ArrayList joinList = new ArrayList();
public static void AddJoin(Join join) {
    joinList.Add(join);}
private bool CheckJoins(StateEvent e) {
    if (joinList.Count == 0)
        return false;
    foreach (Join join in joinList)
        if (((Transition)join.Transitions[0]).ReactsToTrigger
            (e.Trigger)) {
            int joinCounter = 0;
            foreach (Vertex state in currentStates)
                foreach (Transition transition
                    in state.Transitions)
                    if (transition.Target.Name == join.Name)
                        joinCounter++;
            if (join.joinCount == joinCounter) {
                join.EnterState(join.ParentRegion, e);
                return true;}}
    return false;}
```

`CheckJoins()` wird in `StateMachine.DispatchEvent()` jedes Mal aufgerufen, wenn ein neues Ereignis ankommt. Sollte die Vereinigung aktiviert werden, dann wird das Ereignis sofort verbraucht und nicht weiter verschickt.

```
if (!CheckJoins(e)) {
    foreach (Vertex state in currentStates)
        state.ReactsToEvent(e);
    foreach (Vertex state in currentStates)
        if (!e.IsHandled(state.Name))
            state.HandleEvent(e);}
eventQueue.RemoveAt(0);
```

Nun können wir einen Zustandsautomat definieren, in dem Gabelungen und Vereinigungen vorkommen. Gabelungen sind ein normaler `PseudoStateKind.Fork`-Pseudozustand mit mehreren ausgehenden Transitionen.

Da für die Vereinigung auch die eingehende Transitionen verlinkt werden müssen, wird für deren Aufbau es etwas mehr Arbeit gebraucht. Um die Vereinigung in der nächsten Abbildung zu definieren, wird der folgende Code benötigt.




```
// PseudoStateKind.Join Pseudozustand anlegen
PseudoState join;
join = new PseudoState("join", PseudoStateKind.Join, regionA);
/* Ausgehende Transition anlegen und mit dem
entsprechenden Trigger versehen */
Transition fromJoin;
fromJoin = new Transition(null, null, TransitionKind.Local, state1);
fromJoin.AddTrigger("Join");
/* Eingehende Transitionen anlegen und
FixedTarget Eigenschaft setzen */
Transition toJoin1;
Transition toJoin2;
toJoin1 = new Transition(null, null, TransitionKind.Local, join);
toJoin1.FixedSource = parallelState1;
toJoin2 = new Transition(null, null, TransitionKind.Local, join);
toJoin2.FixedSource = parallelState2;
// dem Pseudozustand die eingehende Transitionen hinzufügen
join.AddIncomingTransition(toJoin1);
join.AddIncomintTransition(toJoin2);
// Den Join zu dem Zustandsautomaten hinzufügen
AddJoin(join);
```