using System.Collections.Generic;

namespace ValuationEnvironment.BasicDeltaEngine

{

public class DeltaEngine

{

public IRepository Repository { get; private set; }

public ISpace Perspective { get; private set; }

public DeltaEngine(IRepository repository, ISpace perspective)

{

Repository = repository;

Perspective = perspective;

}

public Deltas<TD> CalculateDeltas<T, TD>(T element) where TD: class, IConstituent<TD>

where T : class, IElement<T>

{

// get the tops of the graph first

var tops = GetTopMost(element);

return Merge<T, TD>(null, tops);

}

public Deltas<TD> Merge<T, TD>(T top, IEnumerable<T> lower) where TD : class, IConstituent<TD>

where T : class, IElement<T>

{

// get the lower deltas first

var allLower = new Deltas<TD>();

foreach (var low in lower)

allLower.UnionWith(Merge<T, TD>(low, GetExpandedResembles(low)));

if (top != null)

allLower.OverwriteWith(top.GetDeltas<TD>());

return allLower;

}

/// <summary>

/// get the elements this resembles from the perspective

/// </summary>

/// <param name="element">the element we want, can be an evolution</param>

/// <returns>the element this resembles in the expanded graph</returns>

public IEnumerable<T> GetExpandedResembles<T>(T element) where T : class, IElement<T>

{

var add = new HashSet<T>();

var remove = new HashSet<T>();

if (element.Replaces != null && element.Resembles != null &&

element.Resembles.Contains(element.Replaces))

GetTopMost(

add,

remove,

Repository.LocateSpace(element.Identifier.Space).DependsOn,

(T) Repository.LocateElement(element.Replaces, element.Parameters),

true);

if (element.Resembles != null)

foreach (var resemble in element.Resembles)

{

if (resemble != element.Replaces)

GetTopMost(

add,

remove,

new[] {Perspective.Identifier},

(T) Repository.LocateElement(resemble, element.Parameters),

true);

}

return add.RemoveAll(remove);

}

/// <summary>

/// get the topmost element(s) for the particular element from the perspective

/// </summary>

/// <param name="element">the element we are looking for</param>

/// <returns>the topmost elements</returns>

public IEnumerable<T> GetTopMost<T>(T element) where T : class, IElement<T>

{

var add = new HashSet<T>();

var remove = new HashSet<T>();

GetTopMost(

add,

remove,

new[] {Perspective.Identifier},

element,

true);

return add.RemoveAll(remove);

}

private void GetTopMost<T>(

ISet<T> add,

ISet<T> remove,

IEnumerable<BasicIdentifier<ISpace>> spaces,

T lookingFor,

bool addNotRemove) where T : class, IElement<T>

{

foreach (var spaceId in spaces)

if (spaceId == lookingFor.Identifier.Space)

Add(add, remove, addNotRemove, lookingFor);

else

{

var replacer = Repository.LocatePossibleReplacement(

spaceId, lookingFor.Identifier, lookingFor.Parameters);

bool localAddNotRemove = addNotRemove;

if (replacer != null)

{

Add(add, remove, addNotRemove, (T) replacer);

// as we have matched something, we now want to just remove

localAddNotRemove = false;

}

ISpace space = Repository.LocateSpace(spaceId);

GetTopMost(add, remove, space.DependsOn, lookingFor, localAddNotRemove);

}

}

private static void Add<T>(ISet<T> add, ISet<T> remove, bool addNotRemove, T element)

where T : class, IElement<T>

{

if (addNotRemove)

add.Add(element);

else

remove.Add(element);

}

}

}

-------------------

using ValuationEnvironment.Utilities;

namespace ValuationEnvironment.BasicDeltaEngine

{

public interface IRepository

{

/// <summary>

/// locate the space object via its identifier

/// </summary>

/// <param name="id">the id for the space</param>

/// <returns>the space object</returns>

/// <throws>DeltaException if space not found</throws>

ISpace LocateSpace(BasicIdentifier<ISpace> id);

/// <summary>

/// locate an element via its identifier

/// </summary>

/// <param name="id">the id for the element</param>

/// <param name="parms">the parameters to substitute for $x$ entries</param>

/// <returns>the element object</returns>

/// <throws>DeltaException if element not found</throws>

IElement<T> LocateElement<T>(ElementIdentifier<T> id, Parameters parms)

where T : class, IElement<T>;

/// <summary>

/// Locate a possible replacement for the base element (identified via identifier)

/// in the space

/// </summary>

/// <param name="space">the space to look in</param>

/// <param name="id">the base element</param>

/// <param name="parms">the parameters to substitute for $x$ entries</param>

/// <returns>a possible replacement, or null</returns>

IElement<T> LocatePossibleReplacement<T>(

BasicIdentifier<ISpace> space, ElementIdentifier<T> id, Parameters parms)

where T : class, IElement<T>;

}

}

---------

using System.Collections.Generic;

using ValuationEnvironment.Utilities;

namespace ValuationEnvironment.BasicDeltaEngine

{

public interface IElement<T> where T: class

{

/// <summary>

/// each element must have a globally unique identifier

/// </summary>

ElementIdentifier<T> Identifier { get; }

/// <summary>

/// this can resemble another element. if this evolves an element, it automatically evolves

/// the previous definition - and it cannot resemble another item. may be null.

/// </summary>

List<ElementIdentifier<T>> Resembles { get; }

/// <summary>

/// the element can replace another in a different space. may be null.

/// </summary>

ElementIdentifier<T> Replaces { get; }

/// <summary>

/// get the deltas for a particular type

/// </summary>

/// <typeparam name="TD">the type we want deltas for</typeparam>

/// <returns>a collection of constituent deltas</returns>

Deltas<TD> GetDeltas<TD>() where TD : class, IConstituent<TD>;

/// <summary>

/// each element can be associated with the parameters used to expand it

/// </summary>

Parameters Parameters { get; set; }

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.Serialization;

using System.Xml.Serialization;

using ValuationEnvironment.Utilities;

namespace ValuationEnvironment.BasicDeltaEngine

{

public class Deltas<TD>: IXMLSerializationCallbacks where TD: class, IConstituent<TD>

{

[XmlElement(ElementName = "delete")]

public List<BasicIdentifier<TD>> Deleted { get; set; }

[XmlElement(ElementName = "replace")]

public Map<TD> Replaced { get; set; }

[XmlElement(ElementName = "add")]

public List<TD> \_Added { get; set; }

[XmlIgnore]

private Map<TD> Added { get; set; }

public void AfterDeserialization()

{

Added = new Map<TD>();

if (\_Added != null)

foreach (var add in \_Added)

Added.Add(add);

if (Replaced != null)

Replaced.AfterDeserialization();

}

public void BeforeSerialization()

{

\_Added = new List<TD>();

if (Added != null)

foreach (var add in Added.GetValues())

\_Added.Add(add);

if (Replaced != null)

Replaced.BeforeSerialization();

if (Replaced.IsEmpty())

Replaced = null;

}

public Deltas()

{

\_Added = new List<TD>();

Added = new Map<TD>();

Deleted = new List<BasicIdentifier<TD>>();

Replaced = new Map<TD>();

}

public Deltas<TD> OverwriteWith(Deltas<TD> deltas)

{

Deleted.AddAll(deltas.Deleted);

Deleted.RemoveAll(deltas.Replaced.GetKeys());

Added.DeleteKeys(Deleted);

Added.UnionWith(deltas.Added);

Replaced.OverwriteWith(deltas.Replaced);

return this;

}

public Deltas<TD> UnionWith(Deltas<TD> deltas)

{

Deleted.AddAll(deltas.Deleted);

Added.UnionWith(deltas.Added);

Replaced.UnionWith(deltas.Replaced);

return this;

}

public Map<TD> GetFinalConstituents()

{

// apply the deletions first to the adds, then add the replacements

var map = new Map<TD>(Added);

map.DeleteKeys(Deleted);

map.OverwriteWith(Replaced);

return map;

}

public void DeltaAdd(TD constituent)

{

Added.Add(constituent);

}

public void DeltaDelete(BasicIdentifier<TD> id)

{

Deleted.Add(id);

}

public void DeltaReplace(BasicIdentifier<TD> id, TD constituent)

{

Replaced.Add(id, constituent);

}

public bool AreAddsUnique()

{

return Added.GetBadKeys().Count == 0;

}

public bool IsFlat()

{

return

(Deleted == null || Deleted.Count == 0) &&

(Replaced == null || Replaced.GetConstituents().Count() == 0);

}

}

}

----------------------------------------------------supporting functions

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Xml.Serialization;

using ValuationEnvironment.Utilities;

namespace ValuationEnvironment.BasicDeltaEngine

{

public class Map<TD>: IEquatable<Map<TD>>, IXMLSerializationCallbacks where TD: class, IConstituent<TD>

{

[XmlElement(ElementName = "element")]

public List<MapPair<TD>> \_All { get; set; }

[XmlIgnore]

private Dictionary<BasicIdentifier<TD>, HashSet<TD>> All { get; set; }

public void AfterDeserialization()

{

All = new Dictionary<BasicIdentifier<TD>, HashSet<TD>>();

if (\_All != null)

foreach (var pair in \_All)

Add(pair.Id, pair.Element);

}

public void BeforeSerialization()

{

\_All = new List<MapPair<TD>>();

if (All != null)

foreach (var pair in All)

foreach (var val in pair.Value)

\_All.Add(new MapPair<TD>(pair.Key, val));

}

public Map()

{

All = new Dictionary<BasicIdentifier<TD>, HashSet<TD>>();

}

public Map(Map<TD> other)

{

All = new Dictionary<BasicIdentifier<TD>, HashSet<TD>>(other.All);

}

public void Add(TD td)

{

var id = td.Identifier;

if (All.ContainsKey(id))

All[id].Add(td);

else

All[id] = new HashSet<TD>() { td };

}

public void Add(params TD[] tds)

{

foreach (var td in tds)

{

var id = td.Identifier;

if (All.ContainsKey(id))

All[id].Add(td);

else

All[id] = new HashSet<TD>() {td};

}

}

public void Add(BasicIdentifier<TD> id, TD td)

{

if (All.ContainsKey(id))

All[id].Add(td);

else

All[id] = new HashSet<TD>() {td};

}

public void OverwriteWith(Map<TD> other)

{

foreach (var pair in other.All)

All[pair.Key] = pair.Value;

}

public void UnionWith(Map<TD> other)

{

foreach (var pair in other.All)

{

if (All.ContainsKey(pair.Key))

All[pair.Key].AddAll(pair.Value);

else

All[pair.Key] = pair.Value;

}

}

public void DeleteKeys(IEnumerable<BasicIdentifier<TD>> delete)

{

foreach (var del in delete)

All.Remove(del);

}

public ICollection<BasicIdentifier<TD>> GetBadKeys()

{

return All.Where(st => st.Value.Count > 1).Select(pair => pair.Key).ToList();

}

public IEnumerable<TD> GetConstituents()

{

var every = new List<TD>();

foreach (var one in All)

every.AddRange(one.Value);

return every;

}

public ICollection<BasicIdentifier<TD>> GetKeys()

{

return All.Keys;

}

public IDictionary<BasicIdentifier<TD>, HashSet<TD>> GetPairs()

{

return All;

}

public override bool Equals(object obj)

{

if (!(obj is Map<TD>))

return false;

return Equals((Map<TD>) obj);

}

public bool Equals(Map<TD> other)

{

if (other == null)

return false;

return

All.Keys.Count == other.All.Keys.Count &&

other.All.Keys.All(key => (All[key].SetEquals(other.All[key])));

}

public override int GetHashCode()

{

return All.GetHashCode();

}

public List<TD> GetValues()

{

var vals = new List<TD>();

foreach (var val in All.Values)

vals.AddAll(val);

return vals;

}

public void SetValues(List<TD> vals)

{

foreach (var val in vals)

Add(val);

}

private void InternalSetPairs(List<MapPair<TD>> value)

{

Console.WriteLine("$$ calling internal set pairs: " + value);

foreach (var mapPair in value)

Add(mapPair.Id, mapPair.Element);

}

private List<MapPair<TD>> InternalGetPairs()

{

var ls = new List<MapPair<TD>>();

foreach (var pair in All)

foreach (var val in pair.Value)

ls.Add(new MapPair<TD>(pair.Key, val));

return ls;

}

public bool IsEmpty()

{

return All.Count == 0;

}

public override string ToString()

{

var bld = new StringBuilder();

bld.Append("{");

foreach (var pair in All)

foreach (var elem in pair.Value)

bld.Append(pair.Key + " -> " + elem + ", ");

bld.Append("}");

return bld.ToString();

}

}

}