오늘은 RealAdmin 제품의 Interface 정의를 가지고 Interface 활용에 대해 설명해보겠습니다.

먼저 RealAdmin 의 하는 일을 살펴보면, 아시다시피, 리얼웹 모든 제품의 공통 모듈을 제공하고, 크게는 4 모듈로 구성되어 있다고 보시면 됩니다.

1. 조직 정보 관리
2. 시스템 계정 정보 관리
3. 접근 권한 관리
4. 서비스 관리 (Calendar, BackgroundService 등의 공통 서비스)

오늘 Interface 활용 예로 들려고 하는 것들은, 전체 틀 내에서 파악해야 할 것도 있고, 단독으로 파악해야 할 것도 있습니다.

먼저 단독으로 활용되는 예를 가지고 설명하겠습니다.

1. **UI 쪽에서 바라봤을 때의 Enity의 공통점**

먼저 대부분의 Entity 중에 부서와 같이 TREE 형태로 표현되는 Entity들이 많습니다. 이런 Entity들의 특징은 부모, 자식, Tree상에서의 위치 정보를 꼭 가지고 있습니다.

public interface ITreeNodeEntity<TEntity> where TEntity : ITreeNodeEntity<TEntity>

{

/// <summary>

/// Parent of this entity

/// </summary>

TEntity Parent { get; set; }

/// <summary>

/// Children of this entity

/// </summary>

ISet<TEntity> Children { get; }

/// <summary>

/// tree node position

/// </summary>

TreeNodePosition NodePosition { get; }

}

ITreeNodeEntity 가 Generic Type이라 좀 복잡하지만, 기본적으로 ITreeNodeEntity가 가지는 속성은 Parent, Children, NodePosition 정보입니다.

왜 위와 같은 interface를 정의했을까요?

RealAdmin 시스템에는 대략 100여 개의 Entity가 존재합니다. 이 중 Tree형태로 표현되는 Entity는 대략 15 개 정도가 됩니다. 흠… 만약에 UI상에 Tree형태를 가지는 Entity를 나타내고자 한다면 (ITreeNodeEntity에는 Title이나 Caption과 같이 나타낼 정보가 빠져있긴 합니다만…) 15 개의 Entity별로 모두 따로 처리해야 합니다.

물론 Abstract 를 사용해도 되지만, 상속관계를 가질 만큼 15개가 밀접한 관련이 없을 수 있습니다. 즉 Interface는 OOP의 상속 관계로 표현할 만큼 밀접한 관계가 있는 객체들의 정의에도 사용될 수 있지만, 전혀 상속관계로 표현할 수 없는 것들도 Interface를 이용하여 동질성을 유지할 수 있습니다. (Role 과 EmployeeGrade는 TreeView 형태로 제공되지만, 상속개념을 가질 근거가 없습니다.)

1. **Entity 자체로서의 동질성의 최고 수준의 추상화**

Entity 종류를 관찰하면, 동질성이 파악되고, 그 동질성의 본질을 정의해 보면, 가장 근본적인 특성이 발견되기도 합니다.

Enterprise (기업), Organization(조직), Employee(직원), Application(응용프로그램), User, Group, Role 등은 접근 권한 관리 모듈에서는 모두 “접근자”에 해당합니다. 접근 대상이 되는 것은 Resource, 접근 권한 정보는 Permission으로 개념화 할 수 있습니다.

즉 “누가 무엇에 접근하는 것을 허용(불허)한다” 라는 정보는 3가지 요소로 구성되고, 그 요소 중에 주어는 “접근자”이고, “무엇”에 해당하는 것이 “접근 대상”, 행동을 나타내는 접근이 Operation, 허용/불허한다 라는 서술어는 Permission에 해당합니다.

이런 논리적인 구분 후에 “접근자” 에 해당하는 Entity를 구분해 보면 위에 얘기한 Enterprise ~ Role 이 모두 해당됩니다.

하지만 Package 구성 상 Enterprise, Organization, Employee, Group은 조직관리에 정의되고, 나머지는 시스템 관리에서 정의했습니다. – 일반적으로 7개의 Entity가 추상화 단계를 거쳐 Base Class로부터 상속을 받게 해도 됩니다.

하지만 그렇게 하지 않은 이유는 크게 두가지입니다.

1. 7가지가 접근권한관리 모듈에서는 “접근자” 로서 공통점이 있지만, 다른 관점에서는 서로간에 소유관계를 나타내는 주어와 목적어로 나뉘고, 그 경우가 상당히 복잡하게 되어, 한 경우만을 생각해서 상속관계를 규정하는 것이 향후 족쇄가 될 소지가 크다고 판단했습니다.
2. RealAdmin 은 NHibernate 사용하므로, Entity의 Base Class는 이미 정해져 있습니다. (C#, Java, Object Pascal등은 다중 상속을 지원하지 않습니다.)   
   물론 1차 상속이 아닌, 2~3차 상속을 하여 7개의 Entity를 묶을 수도 있습니다. 이렇게 되면 상속관계 파악 및 계보를 유지보수하는 데 너무 많은 노력이 필요해 보였습니다.

자 그럼 7개의 Entity의 권한관리 시스템에서 공통점은 무었일까요? 앞에서도 말했듯이 “접근자” 이므로 IAccessor 라 표현하면 되겠죠?

근데 너무 권한관리에 치중한 느낌이 있습니다. 어쨌든 위의 Entity들은 모두 주어로 쓰이고, 능동적으로 행동한다는 의미로 IActor 를 정의하고, 이로부터 상속을 받게 했습니다.

/// <summary>

/// 행위자의 기본 Interface (고유 Identity와 시스템 생명 주기 감시 정보가 있어야 한다.)

/// </summary>

public interface IActor : IActorIdentity, IAuditable, INamedEntity

{

/// <summary>

/// Description of Actor

/// </summary>

string Description { get; set; }

/// <summary>

/// Get ActorIdentity

/// </summary>

/// <returns></returns>

IActorIdentity GetActorIdentity();

/// <summary>

/// Is enabled actor?

/// </summary>

bool? IsEnabled { get; set; }

}

/// <summary>

/// 여러 종류의 Actor를 구분할 수 있는 Identity

/// </summary>

public interface IActorIdentity

{

/// <summary>

/// Actor의 구체적인 Entity의 고유 Guid 값

/// </summary>

[ValidateNonEmpty]

Guid Id { get; }

/// <summary>

/// Actor의 종류

/// </summary>

[ValidateNonEmpty]

ActorKind Kind { get; }

}

/// <summary>

/// Name 속성을 가진 Entity에 대한 Interface

/// </summary>

public interface INamedEntity : IDataObject

{

string Name { get; set; }

}

IActor가 최상위인줄 알았는데, 여러가지 Interface로부터 다중 상속을 받았군요…

구조가 마음에 안들 수도 있습니다. INamedEntity를 정의하는 것보다 INamedActor를 정의하고 Name 속성을 제공하는 것이 더 좋은 방법일 수 있습니다.

물론 이렇게 하지 않은 이유는 다른 Entity들도 IActor와는 상관없이 Id와 Name 속성이 중요 속성으로 추상화 할 수 있기 때문에 분리한 것입니다.

자 크게 두가지 방식의 Interface 정의 방법을 살펴봤습니다.

이제는 위에 정의된 Interface를 어떻게 활용하는 지 보겠습니다.

먼저 ITreeNodeEntity는 부모/자식/TREE위치에 대한 조작을 담당하는 Extension Method를 통해, 전혀 다른 상속관계라도 공통적으로 적용할 수 있습니다.

[소스 : RealWeb.RealAdmin.Utils.Util.TreeView.cs ]

/// <summary>

/// Node Position을 갱신합니다.

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="node"></param>

public static void UpdateNodePosition<T>(this T node) where T : ITreeNodeEntity<T>

{

node.IsNull("entity");

if (Equals(node.Parent, default(T)) == false)

{

node.NodePosition.Level = node.Parent.NodePosition.Level + 1;

var index = node.Parent.Children.IndexOf(node);

node.NodePosition.Order = (index > -1) ? index : 0;

}

else

{

node.NodePosition.Level = 0;

node.NodePosition.Order = 0;

}

}

/// <summary>

/// TreeNode의 자식의 숫자를 구한다. (TreeView 구성 시 Populating을 구현할 때 필요하다)

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="node"></param>

/// <returns></returns>

public static int GetChildCount<T>(this T node) where T : ITreeNodeEntity<T>

{

return UnitOfWork.CurrentSession.CreateFilter(node.Children, "select count(\*)").List<int>()[0];

}

/// <summary>

/// 현재 노드의 Order를 지정된 값으로 설정한다.

/// </summary>

public static void SetNodeOrder<T>(this T node, int order) where T : ITreeNodeEntity<T>

{

node.IsNull("node");

order.IsNegativeNumber("order");

if (Equals(node.Parent, default(T)))

{

node.NodePosition.Order = order;

}

else

{

int oldOrder = node.NodePosition.Order.GetValueOrDefault(order);

var nexts = node.Parent.Children.Where(child => child.NodePosition.Order >= oldOrder);

nexts.ForEach(child => child.NodePosition.Order++);

}

}

/// <summary>

/// 노드의 자식 노드의 순서를 재정렬 한다.

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="parent"></param>

public static void AdjustChildOrders<T>(this T parent) where T : ITreeNodeEntity<T>

{

parent.IsNull("parent");

int order = 0;

foreach (T node in parent.Children.OrderBy(child => child.NodePosition.Order))

node.NodePosition.Order = order++;

}

/// <summary>

/// Node의 부모를 변경합니다.

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="node">자식 노드</param>

/// <param name="oldParent">기존 부모</param>

/// <param name="newParent">새로운 부모</param>

public static void ChangeParent<T>(this T node, T oldParent, T newParent)

where T : ITreeNodeEntity<T>

{

node.IsNull("node");

if (Equals(oldParent, default(T)) == false)

{

oldParent.Children.Remove(node);

}

if (Equals(newParent, default(T)) == false)

{

newParent.Children.Add(node);

}

node.Parent = newParent;

node.UpdateNodePosition();

}

/// <summary>

/// 자식노드의 부모를 지정한 부모 노드로 설정한다.

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="node">자식 노드</param>

/// <param name="parent">부모가 될 노드</param>

public static void SetParent<T>(this T node, T parent) where T : ITreeNodeEntity<T>

{

node.ChangeParent(node.Parent, parent);

}

public static void InsertChildNode<T>(this T parent, T child, int order)

where T : ITreeNodeEntity<T>

{

parent.IsNull("parent");

child.IsNull("child");

order = Math.Max(0, Math.Min(order, parent.Children.Count - 1));

child.Parent = parent;

parent.Children.Add(child);

child.SetNodeOrder(order);

}

몇가지 NHibernate 관련 코드를 빼고도, Generic과 ITreeNodeEntity로 TreeView관련 공통 작업이 가능합니다. RealAdmin이 확장되어 새로운 TreeView 로 표현될 Entity가 필요하다면, ITreeNodeEntity를 사용하면, 재 활용할 수 있습니다.

좀 더 깊게 들어가면 ITreeNodeEntity와 유사한 특성을 가지는 IHasHierarchy라는 Interface도 있습니다.

/// <summary>

/// Hierarchy 특성을 표현한 Interface

/// </summary>

/// <typeparam name="TEntity"></typeparam>

public interface IHasHierarchy<TEntity>

{

///// <summary>

///// Parent of this entity

///// </summary>

//TEntity Parent { get; set; }

///// <summary>

///// Children of this entity

///// </summary>

//ISet<TEntity> Children { get; }

/// <summary>

/// Ancestors of this entity

/// </summary>

ISet<TEntity> Ancestors { get; }

/// <summary>

/// Descendents of this entity

/// </summary>

ISet<TEntity> Descendents { get; }

}

코드에서 보셨듯이 IHasHierarchy는 “조상”, “자손” 정보를 가지고 있습니다. TreeView상에서 자식의 자식, 부모의 부모 등의 모든 정보를 가집니다. 근데 코드를 보시면 Parent, Children을 comment로 막아 놓은 것을 보실 수 있을 겁니다. 처음에는 IHasHierarchy를 ITreeNodeEntity와 별개로 취급하려고 했습니다만, 기존의 Parent/Children 속성을 ITreeNodeEntity가 가지고 있으므로 굳이 IHasHierarcy에 또 표현할 필요가 없습니다.

위의 ITreeNodeEntity와 IHasHierarchy 둘 다를 구현하는 Class로는 Organization 이 있고, ITreeNodeEntity만을 구현하는 Entity 들(GroupTreeNode, OrganizationGrade 등)이 있습니다.

필요에 따라 Interface로 Entity의 성격을 결정하고, 그에 따른 구현을 해주면 됩니다.

다음으로는 IActor의 활용법을 살펴보겠습니다.

[소스: RealWeb.RealAdmin.Utils.Util.Actor.cs]

public static IActor GetActor(string actorName, ActorKind actorKind)

{

if (IsDebugEnabled)

log.DebugFormat("Actor를 로드합니다. actor name={0}, kind={1}", actorName, actorKind);

var actors =

DetachedCriteria.For(actorKind.GetActorType())

.AddNameEq(actorName)

.GetExecutableCriteria(UnitOfWork.CurrentSession)

.SetMaxResults(1)

.List();

Guard.Assert(actors.Count > 0,

string.Format("지정한 Actor를 찾지 못했습니다. ActorName={0}, ActorKind={1}",

actorName, actorKind));

return (IActor)actors[0];

}

앞에서도 말한 7가지 IActor를 구현한 Entity들을 Name과 ActorKind 정보만으로 찾을 수 있도록 했다.

즉 각 Entity별로 구현할 필요도 없고(상속을 통해서도 이렇게 할 수도 있지만), 상속에 얽매이지 않고도 이런 작업이 가능하는 것입니다.

C#,Java의 경우 Reflection 덕분에, 형식(Type)에 대한 정보 추출이 다양해져서, 더 많은 일들을 할 수 있습니다만… 결국은 형식을 알 수 없거나, 지정할 수 없는 경우는 object로 casting됩니다.

Boxing/Unboxing, 형 안정성을 고려한다면, 당연히 정확한 형을 사용하는 것이 좋습니다. 특히나 .NET 2.0 이상에서는 Generic 을 활용하는 것이 더욱 추천할 만한 방법입니다.

이제까지는 Entity 즉 정보를 표현하는 객체로부터 Interface를 활용하는 방식을 설명했습니다.

다음에는 일반적으로 Service라 불리는 Action이 주가 되는 class에 대한 interface에 대해서 살펴보겠습니다.