Autodesk University 2011 LEARN. CONNECT. EXPLORE.

Extensible Storage in the Revit 2012 API

Jeremy Tammik – Autodesk

CP6760-L Store your application data in the Revit model using native Revit API extensible storage. This replaces the old technique of storing data in shared parameters. Access can be restricted to read-only or invisible to other applications. Create your own class-like schema data structures and attach instances of them to any Revit element. Schema-based data is saved with the model and allows for higher-level, metadata-enhanced, object-oriented data structures. This class explains the underlying concepts and real-world techniques. It presents both simple and advanced sample applications demonstrating how to create a schema, read, update, and delete extensible storage data on Revit elements, serialize and de-serialize schemas to XML, extract and display data using .NET generics and reflection, and handle versioning issues when upgrading and extending an existing schema. Please note that prior .NET programming and Revit API experience is required and that this class is not suitable for beginners. This is a hands-on lab associated with the lecture **CP4451** Extensible Storage in the Autodesk Revit 2012 API.

Learning Objectives

At the end of this class, you will be able to:

- Understand the underlying concepts and real-world techniques for working with extensible storage.
- Programmatically create a schema; read, update, and delete extensible storage data on elements.
- Handle versioning issues when upgrading and extending an existing extensible storage schema.
- Extract and display extensible storage data and serialize and de-serialize schemas to XML.

About the Speaker

Jeremy is a member of the AEC workgroup of the Autodesk Developer Network ADN team, providing developer support, training, conference presentations, and blogging on the Revit API. He joined Autodesk in 1988 as the technology evangelist responsible for European developer support to lecture, consult, and support AutoCAD application developers in Europe, the United States, Australia, and Africa. He was a co-founder of ADGE, the AutoCAD Developer Group Europe, and a prolific author on AutoCAD application development. He left Autodesk in 1994 to work as an HVAC application developer, and then rejoined the company in 2005. Jeremy graduated in mathematics and physics in Germany, worked as a teacher and translator, then as a C++ programmer on early GUI and multitasking projects. He is fluent in six European languages, vegetarian, has four kids, plays the flute, likes reading, travelling, theatre improvisation, carpentry, and loves mountains, oceans, sports, and especially climbing.

jeremy.tammik@eur.autodesk.com

Contents

What is Extensible Storage?

Extensible Storage or estorage is a way of programmatically storing arbitrary, auxiliary data in a document. It supports simple data types such as integer, double, string, element id, etc., and complex structures such as lists and dictionaries.

It can be used to replace the old technique of storing data in shared parameters.

Access can be restricted to read-only or invisible to other applications.

Evolution of Storage Options

Internally, Revit makes use of OLE Structured Storage, introduced in 1995. It requires system-level C++, COM and MFC knowledge, and data conversion to byte streams or CObject derived objects.

Ever since the introduction of Revit add-ins, add-in developers have been using shared parameters to store additional custom data. Thus data is stored as a loose collection of data fragments with no protection or permission options. It is not very direct. Still, it was possible to use for storing arbitrary, complex and voluminous structured data, as demonstrated by Miroslav Schonauer's Autodesk University presentations in previous years.

Revit 2012 introduces the new estorage mechanism with the following features:

- Easy to use
- Class-like structure
- Metadata enhanced
- Read and write permissions integrated into Vendor and App ID
- Natively supports element id, UV, and XYZ points and vectors
- Revit-API-centric

Estorage can be used to store data on any Revit Element.

It makes use of a schema, which is a class-like definition specifying the data types and structure of the data to store. The schema contains a list of fields, each of which specifies a name, data type, unit type, and description. The schema is associated with read/write permissions.

The actual data is stored in an entity class, an 'instance' of a schema filled with data.

Each Revit element can store one or more entity instances of different schemata.

Shared Parameters versus Estorage

The use of shared parameters to store add-in data comes with a number of limitations:

- No support for units – high potential for errors.
- No high-level objects such as XYZ, UV, and ElementId.
- No way to track elements when they are deleted or remapped in work sharing.
- No way to selectively hide or expose data.

Extensible storage overcomes these limitations. It provides:

- Unit support for floating point types.
- Support for XYZ, UV, and ElementId.
- ElementId updated during element deletion or work sharing synchronization.
- Read/write permissions at schema level.

Estorage Basics

Estorage thus allows you to create your own class-like schema data structures and attach instances of them to any element in a Revit model, and can be used to replace the technique of storing data in hidden shared parameters. Schema-based data is saved with the Revit model and allows for higher-level, metadata-enhanced, object-oriented data structures. Schema data can be configured to be readable and/or writable to all users, just a specific application vendor, or just a specific application from a vendor. The extensible storage classes are all found in the Autodesk.Revit.DB.ExtensibleStorage namespace:

- Schema – contains a unique schema identifier, read/write permissions, and a collection of data Field objects.
- SchemaBuilder – create Schema definitions.
- Field – contains data name, type, and unit information and is used as the key to access corresponding data in an Entity.
- FieldBuilder – a helper class used with SchemaBuilder when creating a new field.
- Entity – an object containing data corresponding to a Schema that can be inserted into a Revit Element.

The following data types are supported:

- bool, short, int, float, double
- string
- Guid
- ElementId
- Autodesk.Revit.DB.UV
- Autodesk.Revit.DB.XYZ
- Array (as a System.Collections.Generic.IList<T>)
- Map (as a System.Collections.Generic.IDictionary<TKey, TValue> – all types are supported for keys except double, float, XYZ, and UV)
- Autodesk.Revit.DB.ExtensibleStorage.Entity (an instance of another Schema, also known as a SubSchema)

Addition of these data types and structures to a schema are supported by three SchemaBuilder methods

- AddSimpleField(string name, Type): Creates a field containing a single value in the schema, with given name and type.

- AddArrayField(string name, Type): Creates a field containing an array of values in the schema, with given name and type of contained values.
- AddMapField(string name, Type keyType, Type valueType): Creates a field containing an ordered key-value map in the schema, with given name and type of contained values.

They return a FieldBuilder instance, which allow you to specify further properties such as units and documentation for each data item.

Units

All floating-point fields (float, double, XYZ, UV and containers of these values) require a unit type (length, temperature, etc.). Conversely, all other field types may not have units. As stated in the documentation, estorage stores everything in metric for floating point type fields, but you never need to do any conversions on your side. If you want to store a double value in feet, for example, you specify UT_Length in FieldBuilder.SetUnitType, and DUT_DECIMAL_FEET when calling Entity.Set and Entity.Get.

Examples

The class materials include the following commands demonstrating examples of basic estorage usage:

- StoreSimple – estorage of a simple XYZ point
- List – list all storage schemata in memory
- StoreMap – estorage of complex data, a string map
- Delete – deletion of existing storage data
- StoreFile – storage of arbitrary file data on a selected element

Estorage of a Simple XYZ Point

This example demonstrates the very first steps of using estorage. It creates a data structure, populates it with data, and attaches it to a wall. The value is a simple XYZ point representing the stored location of a wiring splice in the wall. The individual steps are clearly visible:

- Instantiate a new SchemaBuilder with the given schema GUID.
- Define the read and write access levels and the vendor id.
- Add a simple XYZ data field.
- Set its unit type and documentation string.
- Register the schema.
- Create an entity to hold an instance of the schema data.
- Access and populate the data field.
- Store the entity on the Revit element.

To complete the full cycle, at the end the command also shows how to retrieve the data back from the wall

```
/// <summary>
/// Create an extensible storage schema,
/// attach it to a wall, populate it with data,
/// and retrieve the data back from the wall.
/// </summary>
void StoreDataInWall(
  Wall wall,
  XYZ dataToStore )
{
  Schema schema = Schema.Lookup( SchemaGuid );

  if( null == schema )
  {
    SchemaBuilder schemaBuilder
      = new SchemaBuilder( SchemaGuid );

    // Allow anyone to read the object

    schemaBuilder.SetReadAccessLevel(
```

```
        AccessLevel.Public );

      // Restrict writing to this vendor only

      schemaBuilder.SetWriteAccessLevel(
        AccessLevel.Vendor );

      // Required because of restricted write-access

      schemaBuilder.SetVendorId( "TBC_" );

      // Create a field to store an XYZ

      FieldBuilder fieldBuilder = schemaBuilder
        .AddSimpleField( "WireSpliceLocation",
        typeof( XYZ ) );

      fieldBuilder.SetUnitType( UnitType.UT_Length );

      fieldBuilder.SetDocumentation( "A stored "
        + "location value representing a wiring "
        + "splice in a wall." );

      schemaBuilder.SetSchemaName( "WireSpliceLocation" );

      // Register the schema

      schema = schemaBuilder.Finish();
    }

    // Create an entity (object) for this schema (class)

    Entity entity = new Entity( schema );

    // Get the field from the schema

    Field fieldSpliceLocation = schema.GetField(
      "WireSpliceLocation" );

    // Set the value for this entity.
    // For floating-point values, the unit type must
    // be specified. We use feet, to avoid the need
    // to convert from the Revit database values to
    // some other unit.

    entity.Set<XYZ>( fieldSpliceLocation, dataToStore,
      DisplayUnitType.DUT_DECIMAL_FEET );

    // Store the entity on the element

    wall.SetEntity( entity );

    // Read back the data from the wall

    Entity retrievedEntity = wall.GetEntity( schema );

    XYZ retrievedData = retrievedEntity.Get<XYZ>(
      schema.GetField( "WireSpliceLocation" ),
      DisplayUnitType.DUT_DECIMAL_FEET );
  }
```

The specified vendor id is the same as the new required VendorId tag in the add-in manifest. ADNP is a vendor id used by Autodesk for ADN plug-ins. You should replace it by your own Autodesk Registered Developer Symbol RDS.

If the vendor id of the add-in manifest does not match the schema vendor id, an exception is thrown by the call to wall.SetEntity(entity): "Writing of Entities of this Schema is not allowed to the current add-in. A transaction or sub-transaction was opened but not closed. All changes to the active document made by External Command will be discarded."

List Loaded Schemata

Schemata are loaded into memory and accessible in all documents from that moment onward. The following method ListSchemas shows how all schemas currently loaded into Revit memory can be accessed and listed:

```
/// <summary>
/// List all schemas in Revit memory across all documents.
/// </summary>
void ListSchemas()
{
  IList<Schema> schemas = Schema.ListSchemas();

  int n = schemas.Count;

  Debug.Print(
    string.Format( "{0} schema{1} defined:",
      n, PluralSuffix( n ) ) );

  foreach( Schema s in schemas )
  {
    IList<Field> fields = s.ListFields();

    n = fields.Count;

    Debug.Print(
      string.Format( "Schema '{0}' has {1} field{2}:",
        s.SchemaName, n, PluralSuffix( n ) ) );

    foreach( Field f in fields )
    {
      Debug.Print(
        string.Format(
          "Field '{0}' has value type {1}"
          + " and unit type {2}", f.FieldName,
          f.ValueType, f.UnitType ) );
    }
  }
}
```

Determine Estorage Use in Document

Since ListSchemas returns all schemas in memory, it cannot be to check whether estorage exists in a specific given document. It will even return a schema from a previous document that is now closed.

To determine whether estorage exists in a document, you can use ListSchemas to determine whether any schemata are loaded in memory at all. If so, you need to verify whether any elements in the document actually use it, e.g. by iterating through all its elements and check each individually to see whether it contains any entities of any schema.

Here are two methods which implement this two-step process:

```
public static bool StorageExistsQuick()
{
  IList<Schema> schemas = Schema.ListSchemas();
  return null != schemas && 0 < schemas.Count;
}

public static bool StorageExistSlow( Document doc )
{
```

```
  IList<Schema> schemas = Schema.ListSchemas();

  IEnumerable<Element> elements
    = Util.GetAllElements( doc ).ToElements();

  foreach( Schema schema in schemas )
  {
    foreach( Element e in elements )
    {
      Entity entity = e.GetEntity( schema );

      if( entity.IsValid() )
      {
        return true;
      }
    }
  }
  return false;
}
```

This is an example of using these methods:

```
  if( !StorageExistsQuick() )
  {
    Debug.Print( "None of the open documents "
      + "contains any exstensible storage data." );
  }
  else
  {
    UIApplication uiapp = commandData.Application;
    Application app = uiapp.Application;
    DocumentSet docs = app.Documents;

    int n = docs.Size;

    Debug.Print( string.Format( "{0} document{1}{2}",
      n, Util.PluralSuffix( n ), ( 0 == n ? "." : ":" ) ) );

    foreach( Document doc in docs )
    {
      Debug.Print( string.Format(
        "  '{0}' does {1}contain extensible storage data.",
        doc.Title, ( StorageExistSlow( doc ) ? "" : "not " ) ) );
    }
  }
```

Estorage of Complex Data

Arrays and maps are represented using the generic IList<T> and IDictionary<TKey, TValue> .NET classes. To actually populate them obviously requires a concrete instance of such a class, e.g. a List or a Dictionary. Here is an example of storing a dictionary mapping string-valued keys to string values. The code is identical to the simple XYZ example above, except for the call to add the field to the schema and to populate the entity with data. Instead of calling the AddSimpleField method, we use AddMapField:

```
  FieldBuilder fieldBuilder
    = schemaBuilder.AddMapField( "StringMap",
      typeof( string ), typeof( string ) );
```

To populate the field, we instantiate a dictionary, add values to it, and write it to the entity:

```
  IDictionary<string, string> stringMap
    = new Dictionary<string, string>();

  stringMap.Add( "key1", "value1" );
  stringMap.Add( "key2", "value2" );
```

```
  entity.Set<IDictionary<string, string>>(
    field, stringMap );
```

The rest of the code remains unchanged from above.

Estorage Deletion

You can delete an individual schema entity from a Revit element using the Element.DeleteEntity method. The UpgradeSchema sample discussed at the end demonstrates its use. To completely delete the schema itself and all its associated data, the Schema class static method EraseSchemaAndAllEntities takes a schema as an argument, erases all entities corresponding to it from all open documents, and erases the schema from memory. This allows users to control the amount of memory consumed by estorage data. Obviously, this method cannot erase the schemata from closed documents, so if a closed document contains entities of an erased schema, opening it will reintroduce it into memory.

Besides the schema argument, the method takes a Boolean argument enabling deletion of schema entities that an add-in normally would not have write permission for. Set this flag to true only if the user gave explicit permission to destroy the schema.

```
  bool overrideWriteAccessWithUserPermission
    = Question( "Also delete Entities that you"
      + " have no write permission to?" );

  foreach( Schema schema in schemas )
  {
    // Note: this deletes storage of this
    // schema in *all* open documents.

    Schema.EraseSchemaAndAllEntities( schema,
      overrideWriteAccessWithUserPermission );
  }
```

Storing a File on a Revit Element

Now that we know how to define a schema and store simple data types, arrays and maps, it is also easy to store an arbitrary file in estorage. Simply convert it to a byte stream and store the bytes in an array. Here is the code to define an appropriate schema, which can also hold the original file name and folder:

```
  FieldBuilder fieldBuilder = schemaBuilder
    .AddSimpleField( "Filename", typeof( string ) );

  fieldBuilder.SetDocumentation( "File name" );

  fieldBuilder = schemaBuilder.AddSimpleField(
    "Folder", typeof( string ) );

  fieldBuilder.SetDocumentation( "Original file folder path" );

  fieldBuilder = schemaBuilder.AddArrayField(
    "Data", typeof( byte ) );

  fieldBuilder.SetDocumentation( "Stored file data" );
```

Once the schema has been defined, an entity can be populated with the file data like this:

```
  // Prepare the data to store

  byte[] data = File.ReadAllBytes( filename );

  string folder = Path.GetDirectoryName( filename );

  filename = Path.GetFileName( filename );

  // Create an entity (object) for this schema (class)

  Entity entity = new Entity( schema );
```

```
// Set the values for this entity

entity.Set<string>( schema.GetField( "Filename" ), filename );
entity.Set<string>( schema.GetField( "Folder" ), folder );
entity.Set<IList<byte>>( schema.GetField( "Data" ), data );

// Store the entity on the element

e.SetEntity( entity );
```

To complete the round trip, here are the steps to restore a file from estorage data saved on a selected Revit element ‘e’:

```
Entity ent = e.GetEntity( schema );

string filepath = Path.Combine(
  ent.Get<string>( schema.GetField( "Folder" ) ),
  ent.Get<string>( schema.GetField( "Filename" ) ) );

byte[] data = ent.Get<IList<byte>>( schema.GetField( "Data" ) ).ToArray<byte>();

File.WriteAllBytes( filepath, data );
```

Additional Observations

Now that we have covered the basics and more, here is a list of some noteworthy storage features and background information, partly in the form of questions and answers, e.g. on the handling of large amounts of data and element ids.

1. Intended Use of Estorage

Question: When should I use estorage versus the existing technique of storing my data in text form in an XML-based hidden shared parameter?

Answer: Estorage is for storing a lot of complex bits of data that you want to organize into a class-like structure, complete with units, documentation, etc. If you already have an XML file that does all of that already, and you don't find using a single hidden shared parameter to be a burden, you might want to go ahead and keep using it. On the other hand, if you didn't already have an XML schema in place and had a huge variety of data you didn't want to convert to a text representation, I'd recommend starting out with estorage.

2. Handling Large Amounts of Data in Estorage

Question: I am thinking of storing a large amount of data on a number of BIM elements. What approach would you recommend?

Answer: The intended and recommended use of estorage does not include storing huge amounts of data in the model. For instance, we have seen issues with developers trying to store very heavy analysis data in hundreds of MB spread across thousands of elements, all loaded at start-up. Such usage will degrade performance. If you store large amounts of data across thousands of elements, do not expect an instant load time.

If you wish to store a large amount of data on individual elements, this should not be a problem. For instance, storing something like a .png file on a wall element is no issue at all, whereas it would be an issue to store a dozen .png files on **every** wall element and extract all of them at once, even if you only need the data for one at a given time.

One of the strengths of estorage is its handling of arrays of objects or sub-entities and different schemas in general rather than one large segment of data that needs to be read in its entirety and de-serialized all at once. I recommend taking advantage of this and only loading what you actually need for a given task. For small loads of data, this might not seem necessary, but when you get into many MB of data, it makes a difference.

3. Estorage is Self-documenting

When you create a schema, you are creating documentation. Be sure to fill out the documentation strings for each field – they will help you in your development process and others when they use your schema. What's more, since you can look up a schema by Guid, if you want to share a document with a schema with someone else, all they need is the Guid to look it up and read your structure and documentation comments. This might be a good opportunity to either use the SchemaWrapperTools included with the ExtensibleStorageManager SDK sample (or a simpler, similar tool) to print out a schema's field definitions and documentation strings from a single GUID input.

4. Estorage is Object-oriented in Two Ways

Not only is a given schema entity structured into named fields, but each entity is placed on elements relative to the data itself, as opposed to one large blob that you must read in its entirety to unpack. This goes along with what the recommendation above about not reading all storage at start-up. Since you can choose which elements to process, you have the opportunity to only load what you need an automatically have an association between data and a specific element – shared parameters can't do this.

5. Read/Write Permissions

This is another area that goes beyond shared parameters. While your schema definition is public, you can restrict who reads and writes schema data based to a specific vendor or a specific application from that vendor.

6. Modification of Estorage Data on an Element Type

Question: When transferring types from one project to another using Transfer Project Standards, it only copies across types that are different. If you change a parameter on a wall type that exists in both projects, then it gives you the option of copying over "new" types or overwriting existing types. However, if the parameters are unchanged but the schema data is different, it treats the wall types as identical, so does not give the option of copying over the wall type.

Are element types with different estorage data attached to them treated as the same or not?

Answer: Any parameter based change, hidden or not, will trigger a new type to be recognized. Estorage, however, is not a parameter-based transaction, so those rules don't apply. Therefore, element types with differing estorage attached to them are still treated as the same in this case.

7. Handling of ElementId Data in Estorage

Question: What happens to element ids stored in estorage?

Answer: When you store an ElementId using estorage and that ElementId gets remapped because the element is deleted or updated, e.g. because of a work sharing update, your stored ElementId is also automatically remapped to the new ElementId value. This is one strong advantage for using estorage over text or raw numbers to store ElementIds – the tracking for element updates is handled automatically, so you can be sure that your ElementIds will remain valid. If the element is deleted, your ElementId will be set to ElementId.InvalidElementId.

8. Retrieving Elements with a Specific Schema Entity

Question: How can I retrieve all elements that have data from a certain schema attached to them? Optimally, I think that should be a filtered element collector option.

Answer: Right now, you must do a manual iteration of all elements. There may be a project to add a filter as you describe in the future, but we make no promises about any future features whatsoever.

9. Checking for a Valid Entity on an Element

Question: If I register a schema and then select an element that has no entity for that schema attached to it, I would expect the following call to return null:

```
  Entity ent = e.GetEntity( schema );
```

It does not. Instead, it returns a valid entity pointing to a null schema. To handle this, I expanded my check to this:

```
if( null == ent || null == ent.Schema ) ...
```

Answer: Use the Entity.IsValid method to check to see if the entity you received from GetEntity actually has data of a given schema.

10. One Entity per Element

Question: Is only one entity per schema possible per Revit element?

Answer: Yes and no. There is no way to have more than one entity of a given schema per element via the GetEntity/SetEntity operations. However, you could always create another schema with more than one sub-entities of a given schema type, including array fields and map fields.

11. Schemata Remain in Memory

Question: Is it true that once a schema has been loaded into Revit memory, it never disappears again until the session ends?

Answer: That is true. A schema is available per use on the session level, even if a new document is created. Entities are associated with specific documents.

Sample Applications

Here are a few complete sample applications demonstrating more advanced estorage topics:

- FamilyStorage – store data in a family file and use it when inserted into a project.
- UpgradeSchema – manage schema versioning and upgrading.
- Dynamic Section View – simple, practical use of storing an XYZ point.
- ExtensibleStorageManager – in-depth example to manage, share, and store schemas including generic schema wrapping and data display.

FamilyStorage

The FamilyStorage sample shows how to create estorage in a family document and retrieve its data in a project containing an instance of that family. It defines two external commands to implement this, AddDataToFamily and GetFamilyData.

AddDataToFamily defines a simple estorage schema and adds an entity for it to the family itself, accessed through the document OwnerFamily property:

```
Family documentFamily = doc.OwnerFamily;
Entity newEntity = MakeEntity( new XYZ( 1, 2, 3 ) );
documentFamily.SetEntity( newEntity );
```

GetFamilyData retrieves the stored data from the family by asking each family instance for its family:

```
Family family = familyInstance.Symbol.Family;
Entity entity = family.GetEntity( schema );
```

To test this, start by creating a new family and run the command AddDataToFamily. Note that the schema GUID is stored in a separate class. Save the document. Create a new project and insert an instance of the family into it. Run the command GetFamilyData.

UpgradeSchema

The UpgradeSchema sample defines an external application and command demonstrating a simple estorage upgrade scenario, i.e. maintenance and migration of estorage data from an old version of a data schema to an enhanced updated version.

The two schemata involved are labelled v1 and v2. The sample application implements an external command which adds data for schema v1 to all walls in the document.

Furthermore, it implements an external application which does two things: create a ribbon panel to launch the command, and subscribe to the DocumentOpened event. In the event handler, all elements with v1 schema entities attached to them are retrieved and their data is automatically updated to schema v2.

Here are steps to use it:

- Build and install the application.
- Create a document with a few walls. Save, close, and re-open it.
- Note that the event looking for elements containing entities of schema v1 does not find any yet.
- Click the command button to add entities of schema v1 to each wall. Save and close.
- Reopen the document, and note how:
  - We only need to create schema v2 – schema v1 is already saved and does not need to be re-created.
  - Elements containing entities of schema v1 are found, data is copied into new entities of schema v2, and original data is deleted.

To find elements with a specific schema entity attached to them, we iterate over a given element set and query each for valid entity of that schema:

```
foreach( Element e in collector )
{
  Entity entity = e.GetEntity( schema );

  if( entity.IsValid() )
  {
    retval.Add( e.Id );
  }
}
```

Deleting the old schema data and adding the new is straightforward, and uses the DeleteEntity method:

```
Entity oldEntity = element.GetEntity( oldSchema ); // v1
Entity newEntity = new Entity( newSchema ); // v2

newEntity.Set<XYZ>( "Location", oldEntity.Get<XYZ>(
  "Location", DisplayUnitType.DUT_METERS ),
  DisplayUnitType.DUT_METERS );

newEntity.Set<string>( "Id", oldEntity.Get<string>( "Id" ) );

newEntity.Set<float>( "ScaleFactor", 1.0f,
  DisplayUnitType.DUT_METERS );

element.DeleteEntity( oldSchema );  // delete old entity
element.SetEntity( newEntity );  // set new entity
```

Dynamic Section View

This sample is part of the Revit SDK, provided in the DynamicModelUpdate SDK sample. It ensures that a section line follows a window, so that the resulting section view always displays the position of the window in the wall. Regardless of how the wall and window are moved around in the model, the section view will always follow them to display the correct result. How is this achieved?

- Store the window position.
- Use an updater to run code when the position changes.
- Compare the old position to the new position.
- Translate and rotate based on the difference.
- Store the updated position.

The window position is stored using a simple estorage schema with two XYZ fields for the window location point and facing orientation.

Schema Wrapper Tools

SchemaWrapperTools is a C# library included in the Revit SDK ExtensibleStorageManager sample. It provides a simpler level of access to the estorage Schema, SchemaBuilder, Field, and FieldBuilder objects and provides easy serialization of schema data to XML for the user, providing support to

- Serialize schema data to and from disk
- Display all data of a particular schema in a schema entity

It defines the following classes:

- FieldData – wraps the data of a schema field.
- SchemaDataWrapper – contains a collection of FieldData objects and top-level schema information.
- SchemaWrapper – contains a SchemaDataWrapper object and high-level operations for creating, saving, and reading schemata.

Extensible Storage Manager

The ExtensibleStorageManager is one of the advanced Revit SDK samples. It shows how to create complex schemas and gives a framework for saving and loading schemas to a file and thus easily sharing and communicating schemas.

It demonstrates the estorage classes and related APIs by creating a wrapper class and control dialog that creates schemas, serializes and de-serializes them to XML, and extracts and displays the data contained in the schema entities. It thus enables you to create, read, update, and delete third party extensible storage in Revit Elements.

If you define a schema using C# and then want to share it with a VB.NET user, and your code doesn't factor well for reuse, this provides an easy way to do so. Here is an example walkthrough:

- Create a schema and fill with data
- Examine the saved schema
- Save the file
- Reload and query

The ExtensibleStorageManager application contains two separate projects:

- The SchemaWrapperTools C# library described above.
- ExtensibleStorageManager, a Revit external application and command that launches a WPF dialogue allowing interaction with the SchemaWrapper class and view data and schemata serialized into elements in a Revit document.

This sample makes heavy use of generics and reflection to query types, create types, call methods, query generic parameters, and create generic container classes based on types discovered at runtime. This use of reflection may be a challenge at first, but it allows the user to serialize schemata to XML containing any number of any data types, any supported container type of any supported data type, or any number of different sub-schemas containing any of these types.

A recommended approach is to create simple schemas at first with just a few fields and seeing how the SchemaWrapper creates Schema objects and XML from them before moving onto lists of objects, maps of objects, and sub-schemas or lists or maps of sub-schemas.

Related Blog Posts

- http://thebuildingcoder.typepad.com/blog/2011/04/extensible-storage.html
- http://thebuildingcoder.typepad.com/blog/2011/05/extensible-storage-of-a-map.html
- http://thebuildingcoder.typepad.com/blog/2011/06/extensible-storage-features.html

Acknowledgements

I thank Steven Mycynek for his ideas, support, and material, and my ADN DevTech colleagues for being such a great team to work with.

Appendix: Hands-on Lab Instructions

The storage lab machines are preinstalled with

- Revit Architecture 2012
- Microsoft Visual Studio 2010 Development Environment incl. C# and VB.NET

You are provided with additional lab materials in the following subdirectories of the folder named **Estorage**:

- **Doc** – these instructions, the estorage Powerpoint presentation and sample code
- **Sdk** – the Revit SDK including API documentation
- **Addins** – prebuilt RevitLookup and RvtSamples executables
- **Wizards** – a set of Visual Studio add-in wizards for the storage exercises

Revit SDK

The most important content in the Revit SDK for our purposes is the API documentation, which you require to complete the exercises. It also includes a hundred-odd sample applications, a Visual Studio solution to compile them all, and two important utilities that you should be aware of:

- Revit API help file "RevitAPI.chm"
- Developer guide "Revit 2012 API Developer Guide.pdf"
- Visual Studio solution for all samples "SDKSamples2012.sln"
- RevitLookup – interactively explore the Revit database
- RvtSamples – load and launch all Revit SDK samples

I advise you to set up shortcuts on your desktop for the help file and the developer guide right away.

Precompiled versions of RevitLookup and RvtSamples are provided in the \Estorage\Addins folder. These can be copied to the Revit Addins folder on your system as described in the developer guide section 3.4.1 Manifest Files.

Visual Studio Add-in Wizards

The Visual Studio add-in wizards generate entire C# solutions providing a beginning point for each of the exercises 1-5:

- Ex_1_StoreSimple
- Ex_2_List
- Ex_3_StoreMap
- Ex_4_Delete
- Ex_5_StoreFile

For each of the five exercises, a solution version is also provided, which you can use to peek into if you run into a problem that has you stumped. Alternatively, you can always have a look at the completed application estorage sample code.

The wizards are provided as zip files. In order for them to be picked up, recognised and loaded into Visual Studio, all that needs to be done is to copy the zip files into the Visual Studio C# project template folder

- C:\My Documents\Visual Studio 2010\Templates\ProjectTemplates\Visual C#

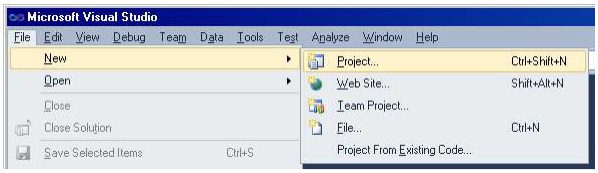
Hands-on Exercises

The hands-on exercises correspond to the sample commands discussed above, so please refer to those descriptions for further details and background information on their functionality.

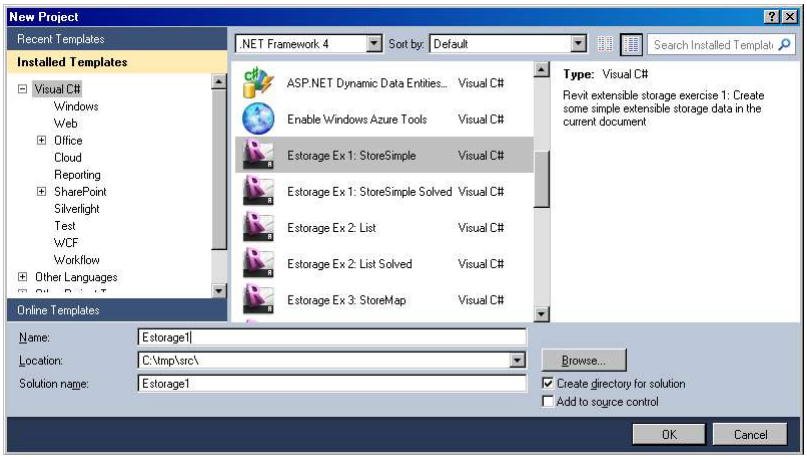
The actual exercise for you to complete consists of fleshing out the code in the regions marked by a comment starting with “// Todo: ”. In general, the comment will mention what Revit API method you need to use and what arguments to supply to it:

For further details on what these methods do and how to use them, please refer to the explanations in the Revit API help file RevitAPI.chm and the developer guide "Revit 2012 API Developer Guide.pdf", both of which are part of the Revit SDK.

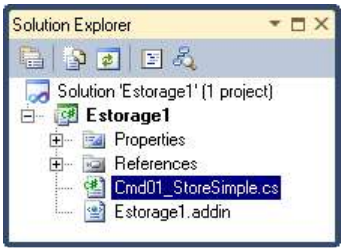
To start working on a new exercise, select File > New > Project in Visual Studio:



Pick the desired exercise template:



The exercise wizard will generate a ready-to-compile solution for you:

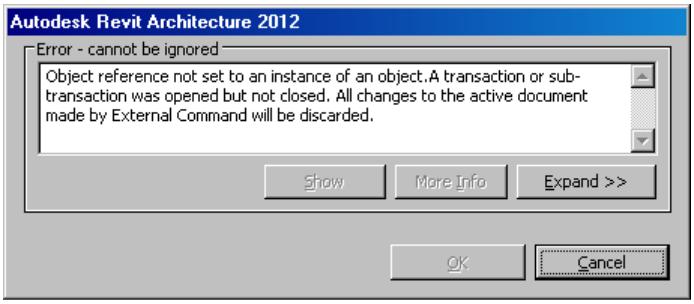


The C# modules are pre-populated with the code to define an external command completing most of the required actions. The most pertinent code statements relating to estorage have been removed, and it is your job to find out how they should be implemented to make it work again. Look for the todo comments such as this one:

```
        // Todo: Use the AddSimpleField method on the
        // schema builder to create a field named
        // "WireSpliceLocation" to store an XYZ
        // data item

        FieldBuilder fieldBuilder = null;
```

In general, the external command of the exercise cannot be executed successfully without completing the exercise, because some object initialisation code is part of the exercise. Some objects are initialised to null, and it is you task to assign an appropriate value to them. Here is an example of an exception being thrown in an uncompleted exercise:



To fix this, simply follow the instruction in the todo comments, referring to the Revit API help file, developer guide, the handout description of the storage sample commands, and, if all else fails, the solved versions of the exercises or the completed estorage sample application code.

Good luck, keep at it, and above all have fun!