Autodesk University 2012

Core Autodesk Revit API Explained

Arnošt Löbel – Autodesk, Inc.

CP3426 Understanding how core components of the Revit API behave and are entangled together is one important prerequisite of developing safe and robust external applications on top of Revit. There are many common rules a good application should follow no matter how it interacts with Revit. However, in this class we will cover only the most important ones of the most common Revit API frameworks. Though we will not have a chance to go into depths of each of the framework we mention, we'll try to point out things that are essential yet not commonly known, or even misunderstood often.

Learning Objectives

At the end of this class, you will understand and know how to utilize:

- Regeneration
- Transaction Modes
- Transaction Phases
- Document modifiability
- Element validity
- Object lifespan
- External commands
- API Events, Callbacks, Updaters
- API Contexts and Firewall

About the Speaker

Arnošt is a senior principal software engineer on the Revit development team. He joined Autodesk in 2006 as a new member of the growing API team and since his first day the API has been his primary focus and passion. His major contributions include the redesigned transaction framework, events, dynamic model updaters, and API safety. Arnošt has a master degree in structural engineering, but only a few years after college he switched professions to become a full-time software developer. He first started developing add-ons for AutoCAD® in a small start-up, and contributed to various 2D graphics and CAD oriented projects since.

Arnošt was born in the Czech Republic and relocated with his family to Miami, FL in 1998. He and his wife now live in Boston, which they really enjoy, though he has a hard time accepting the fact their daughters live in two other corners of the U.S. Arnošt is an avid bicyclist, and skier, and runner, and an occasional badminton and Ping-Pong player, and a seasonal gym-rat.

Table of Contents

Regeneration modes

It used to be that external applications had to specify how they wanted regeneration to behave in API methods. Programmers had to specify whether they wanted the model regenerated after each and every call to the API (the automatic mode) or if they preferred to call regeneration manually as needed (the manual mode). Actually, before those modes were available, the API always worked as in the Automatic regeneration mode.

Both those modes are now history – there is no regeneration mode to be declared. Since R2012, the API always works as if in the manual mode. Users can still declare the manual mode, but the declaration will be ignored. Users will get an error if they declare their applications with the Automatic mode.

There are few things to remember about using regeneration:

- Programmers invoke regeneration as they need it. The method to call is `Document.Regenerate`.
- Regeneration may be called only in a transaction; it does not make sense outside of it. In fact, it should throw an exception.
- A changed model should be regenerated before it is read from. It is to prevent getting obsolete data from a model that has not yet propagated all changes through out to all related elements.
- Regeneration is smart – it only causes regenerations of the elements that have actually been changed. If you call regeneration twice one right after the other, the second time it will be a virtual no-op.
- Ending a transaction always triggers regeneration; therefore there is (almost ever!) no need to regenerate manually before committing a transaction. On the other hand, ending a sub-transaction does nothing to the model, thus if you need to read from the model after a sub-transaction was committed or rolled back, you will likely need to regenerate explicitly.
- In fact, manual regeneration alone is sometimes not enough to guarantee the integrity of the model. In quite a few cases the current transaction must be committed. It is because there is a lot more going on at the end of transaction besides regeneration of the model and sometimes those other things are important in order to read from the model.

  Let's list just a few of the many things:

  1. Auto-joining of elements is processed
  2. Dynamic updaters are invoked, possibly in several iterations
  3. Element assemblies get synchronized
  4. Analytical model may need adjustments
  5. Many work-sharing actions take place
  6. Failures are handled and mistakes corrected as needed

Transaction modes

Unlike the regeneration modes, Transaction modes are still in use and they will stay in the future. Even though the automatic transaction mode may be phased out eventually, the Revit API will probably always have at least two modes – Manual and Read-Only.

Automatic mode

This mode is available for backward compatibility only. We do not recommend using it in the current and future external applications, for it will quite possibly not be supported in the future.

Pros:

a) Easy to use, not having to care about transactions at all, assuming you actually need them.

Cons:

a) A transaction is always created even when it is not going to be needed during a command. The good news is that the transaction will not be committed unless something actually changes – this was an improvement in R2012 over 2011.
a) You may not have your own transactions inside the command, because transaction may not be nested.
b) You may not even name the one transaction - it will bear the command's name.
c) You may be limited in the number of API methods allowed to invoke in your command, for some methods are valid only while outside of a transaction.
d) You may not use your command if there are not documents open in Revit.

Manual mode

This is the preferred mode and the only mode Revit internal programmers can use. It gives API clients freedom and full control over transactions in their commands.

Pros and Cons: The set of advantages and disadvantages is the exact opposite of the Automatic transaction mode. For a small price, the user has all the freedom in his or her external command. There are not restrictions over what to call; there are only restrictions over when to call what – for some methods must be in a transaction and some must be not (and the rest of methods do not care).

Read-Only mode

A command running in this mode guarantees the active document will not be modified, which means:

a) The command in its implementation may not modify it.
b) No other registered API client may modify it while the command is being executed.
c) Not even Revit internally may modify the document.
d) No transactions may be started. Not even a transaction group may be started.
e) Methods that expect the document to be modifiable will throw an exception.
f) Methods that attempt to start a transaction internally will also throw when invoked.

It should probably be mentioned that transactions modes apply to external commands only - they do not have any effect on anything else. For example, an event handler is completely on its own regardless of the transaction mode of the external command during which the event handler might have been registered. Also, declaring an external application with a transaction mode will be ignored.

It is recommended to use the Manual Transaction Mode in external commands.

External commands

We do not feel like we need to say a lot about external commands; most Revit API developers are intimately familiar with them and with all the ways of using them. Let's just sum up only those few things our Developer's Guide might have missed to point out:

- An external command may be invoked only when:
  a) No other command is being executed
  b) No edit mode or editor is in effect
  c) No transactions or groups are open in the active document (this is not very likely)
- Revit does not guarantee lifetime of commands beyond their execution scope, even though the common practice (in Revit) is to keep them around for the entire Revit session. Since it is not guaranteed, it is recommended not to store there anything that may need to be used elsewhere, such as global variables or event handlers.
- Changes made by failed commands are completely discarded. It does not matter how many successful transactions were committed during such command. If the command does not succeed, all its transactions will be rolled back. This applies to the active document only; inactive documents will stay as they are at the end of the command regardless of the command's result.
- There is a protective layer – a.k.a. the API Firewall - around every call out to a command. A command that breaks the rules of the firewall will be discarded. We will look closely at the rules later.

API Context

Revit may have the rule about API calls being allowed only during certain times, but there are no rules requiring one client to finish his command execution before another client can process his or her code. Not only are invocations allowed to be nested in each other, but executions can be done in different external applications.

For example, the following scenario is perfectly possible and allowed:

⇨ Revit invokes an external command, which does something…
  ↳ triggering an event to be handled by another application…
    ↳ triggering another event to be handled by yet another application…
      ↳ causing a dynamic updater to be executed back in the first application.

Application developers should not be surprised by such scenarios; they should expect it and anticipate it.

Revit keeps a FILO stack of the currently executing applications, thus it always knows the application running atop. Having this information available allows Revit to forbid certain operations to an application if it suspects that the application should not be invoking those operations. For example, an application cannot change the settings of another application's updater.

The bottom external process, the one Revit invoked first is sometime being held responsible for other application's faults. It may not be exactly fair, but it is how Revit works currently. In order to save time by not running every possible test after every application that leaves the invocations scope, some tests are only done at the end when there is only one executor left. For example, consider this case:

1. An external command is invoked by Revit.
2. The command makes a few changes in the active document and creates another document.
3. Then it starts a transaction there, makes changes, and commits the transaction.
4. After that, the command saves the new document
5. There is another application which subscribed to the *DocumentSaved* event.
6. The handler starts a transaction, makes changes, but forgets to commit the transaction.
7. The event framework lets it pass, because it is now at the second tier of execution.
8. The process comes back to the first application, which does nothing else but returns.

9. Revit checks and finds there is an open transaction in one of the documents. It rolls it back and marks the command as not successful, which means whatever changes the command might have done in the active document would also be discarded.

Unfortunately there is nothing that the first application could have done to avoid the problem. The programmer may only hope the end user will notice that he gets more problems when the second application is installed and uninstalls it.

API Firewall

In the just described scenario, the decision to discard the command's changes was made by the now infamous API firewall. What is it? Well, like any other firewall, its purpose is to protect, in this particular case to protect Revit application and its open documents from any harm caused by misbehaving external applications.

- API Firewall wraps around API invocations, particularly about such invocation that may maintain their own transactions, namely External commands and Events.
- The main objective is to make sure clients do not leave anything open that was opened during the invocation. Mostly transactions, sub-transactions and transaction groups, but also edit modes, potentially, or simply anything that has a start and an end.
- During any invocation, nothing is allowed to be left open in the active document (and/or the document an event was raised for). When such situation occurs, the violating procedure is rendered as a failure and everything that was done during the process is discarded. In events this applies to the one event handler only; other event handlers will still be called.

  There is one and only one exception to this rule, but it comes with another rule attached: External commands which finished with a pending modeless transaction are allowed to do that, but when the failure dialog is closed, the firewall looks if the command left any transaction groups open around that pending transaction. If it finds that is the case, it discards everything what the command did despite the fact it had ended already.
- If the client is allowed to change the active document at all (by calling *OpenAndActiveDocument*), the firewall validates that the formerly active document is free of transactions and transaction groups.

Object lifespan

Most native programmers (and certainly C++ programmers) are well aware of the fact that once an object leaves the scope of a program block in which it was locally created (meaning – put on the stack), the object ceases to exist. It works differently in managed code when the garbage collector takes care of most of the programmer's worries, but it does not mean it leaves the programmer worry-free. That is especially true in mixed environment such as Revit's where managed object are mixed with and related to native objects.

Let's take a look at some of the most important facts:

- Most managed objects are light-weight wrappers around native instances.
- Some wrappers own native resources, some don't; most don't.
- Objects that may be explicitly destroyed (they have a Destroy method defined) are better if controlled with the **`using`** block (see using the transaction in the example below).

  This is particularly important for transaction phases - Transaction, Transaction Group and Sub Transaction, because the **`using`** block (or calling Destroy explicitly) will invoke the object's

destructor which will then make sure the transaction phase is not left open. If it has not been finished explicitly before reaching the end of the using block, the transaction phase will be rolled back. That is particularly useful when exceptions are somehow more likely to be thrown.

- Garbage collector takes care of managed objects, not necessarily their native resources. Also, be aware that garbage collector may claim "abandoned" (at least as far as the GC is concerned) objects much sooner than a poor programmer may think. It is not uncommon for an object being collected while the program is still executing the object's method. Scoping the objects with the **`using`** block will prevent that from happening (as would explicitly dispose the object).

  Example of scoping a transaction:

```
private void method(Document doc)
{
   using( Transaction transaction = new Transaction(doc, "Operation") )
   {
      transaction.Start();
      methodModifyingDocument(doc);
      transaction.Commit();
   }
}
```

- Revit does not immediately delete native resources of collected managed object. Instead, native resources are destroyed only when Revit leaves an API context, for example upon returning from an external command or event. That is to avoid a couple of problems:
  1. Complications from switching thread context, because collecting of resources is invoked from other than the main thread.
  2. Premature destroying of native resource due the garbage collector being too eager claiming what it "thinks" has been abandoned already.

Purging released API objects

If you worry that your application may need more memory than what you originally anticipated because Revit seems to keep native objects alive longer, your worries may not be without a reason. Since Revit only releases collected native objects at the end of standard API invocations, there may be a quite a few (thousand) object hanging, waiting to be destroyed. If your application is object hungry and needs to create and delete a lot of objects, it is likely you could be reaching some limits earlier than expected.

Fortunately the API now has a method you can use to purge collected native object explicitly: **`PurgeReleasedAPIObject`**. It does exactly what it sounds like – it looks at the pool of native objects left behind by their managed counterparts, and destroys them.

Note: *Native objects are put into the waiting room only when the managed objects are collected by the garbage collector. When you destroy an object explicitly, the native resources of it will be destroyed too.*

Note: Not every managed cause it corresponding native resource to be deleted when the managed brethren is disposed (or collected). In fact, it is a minority of the objects in the Revit API. Most objects have native resources which live permanently in a Revit model and for which is the managed object is only a wrapper allowing access to it (e.g. all element classes).

The Transaction Framework

The Transaction Framework certainly is one of those 20-80 things where with just 20% knowledge you gain 80% of all the benefits. We certainly cannot cover all the features and rules of that framework in this

short section, but you may rest assured that if you follow the few rules we do mention here, your applications will likely be all right.

- There are three major components of the framework:
  a) **Transaction** (T)
  b) **Transaction Group** (TG)
  c) **Sub-Transaction** (ST)
- Only Transactions are necessary in order to make changes to a document. Neither TG nor ST is actually needed when modifying a document. They are only used to better "organize" the changes.
- Transactions may not nests, which means only one transaction per document at any time is allowed to be open/active.
- An ST can only be started in an open transaction and it must be closed before the transaction ends.
- Conversely, a TG can only be started when no transaction is open yet and must be closed only after all transactions were already closed. In other words: If you want to group a couple of transactions together, you have to start the group first.
- Both ST and TG can nest, but overlapping is not allowed. That means you can start one group inside another one, but the child group must be closed before the parent group is. The same applies to sub-transactions.
- Methods like `Start`, `Commit`, `RollBack`, and `Assimilate` can fail. The caller should test the status value these methods return.
- Transaction being committed may actually return the `RolledBack` status instead. It is because of failures that may have been posted during the transition. If they are severe or if the end-user decides they cannot be ignored, the transaction cannot be committed. This is particularly important to watch if there is a sequence of transactions which all need to be committed successfully in order for the sequence to be considered successful.
- Assimilating a transaction group is very like committing it, but not exactly. When a TG is committed, all transactions successfully finished inside the group will stay as they are, which means the end-user will be able to see them in the Undo menu. If a TG is assimilated instead, all enclosed transactions will be merged into just one which will bear the group's name (unless there was only one transaction inside the group in which case the transaction will keep its name).
- API transaction failures are handled modally unless explicitly set to modeless handling. Please keep in mind that although switching to modeless handling is possible, it requires more careful approach to finishing transactions. If there were failures posted inside a transaction and of the failure handling was set to be modeless, the transaction status returned from the Commit method will not be `Committed`, but **`Pending`**. The calling code get control immediately (because the failure handling is working on another thread), but it is utterly important not to do anything in the model until the transaction is finally and completely resolved. This will most likely not be possible without using Transaction Finalizers.

Transaction Finalizer

Transaction finalizers are particularly useful (practically a necessity) when using modeless failure handling during transactions, but they can be used with any transaction that needs a follow-up action after it is completed (committed or rolled back).

A transaction finalizer is a call-back object (a delegate, of sort) that is attached to a transaction object, and of which methods get invoked when the transaction gets finished. One of the methods – `OnCommited`, is called when the transaction is committed, while the other method – `OnRolledBack`

is not surprisingly called when the transaction was rolled back. Only one of the methods is invoked, naturally, and only when the transaction is finally completed, that is after pending state is finished in case failure handling was being done modelessly.

Note: Mo matter what an eventual transaction finalizer does (it can even re-start the completed transaction again), the status returned from the call to Commit or RollBack a transaction stays unaffected. It is the status that was current before the finalizer was invoked.

Utilizing a Transaction Finalizer

```
// A very simple finalizer that just closes an outer transaction group
// after a transaction is closed, depending on the result
class MyFinalizer : ITransactionFinalizer
{
   private TransactionGroup m_group;

   public MyFinalizer(TransactionGroup group)          { m_group = group; }
   public void OnCommitted(Document doc, string name)  { m_group.Commit(); }
   public void OnRolledBack(Document doc, string name) { m_group.RollBack(); }
};

private TransactionStatus CommitTransactionAndGroup
(
   Transaction       trans,   // the transaction to commit
   TransactionGroup  group    // an outer group (enclosing the transaction)
)
{
   MyFinalizer finalizer = new MyFinalizer(group);

   // setting the transaction soit calls the finalier when it is finished
   trans.SetFailureHandlingOptions(
      trans.GetFailureHandlingOptions().
         SetTransactionFinalizer(finalizer));

   return trans.Commit();
}
```

Document modifiability

I do not think transactions in Revit are confusing, but many people do (☺). What they usually are confused with is how and when one can actually modify a document. "*Do I have to start a transaction now?*" and "*May I start a transaction now?*" or "*Do I need to end my transaction now?*" are their typical questions. Let's see if we can shed some light on that by describing three useful properties of the Document class.

Document is Modifiable

a) If it is not in a read-only state
b) and has an open, uncommitted transaction

Document is Read-Only

a) During an external command declared with Read-Only transaction mode
b) Under other circumstances that temporarily put a document into a read only state - such as resolving posted transaction failures, or during document upgrade
c) During some events – rather unpredictably – some documents are put into this mode because they were modifiable at the time of the event and they have to stay untouched.

Document is Read-Only File

a) This property of a document only refers to the actual file on disk
b) Changes can be made to the document, but file cannot be saved (only saved-as)
c) (not always set dependably, it appears)

To be able to open a transaction, the document

1. must not be read-only
2. and must not be modifiable yet (only one transaction at a time is permitted)

Note: *The same applies to transaction groups.*

Possibly useful code snippets:

```
// To check whether changing the model is currently possible

if( document.IsModifiable )
{
   MakeChanges();
}

// To check if a transaction or transaction group can be started

if( !(document.IsModifiable || document.IsReadOnly) )
{
   using( Transaction transaction = new Transaction(document) )
   {
       transaction.Start( "Doing something" );
       MakeChanges();
       transaction.Commit();
   }
}
```

Element validity

As it has been already mentioned before, there is a close relation between the managed object an external programmer works with in his or her managed code and corresponding native instance in Revit. It is mostly one-to-one relation between those two brethren and while a native object can exist without its managed part, it is rarely the case the other way around. So, while you can keep your instances of simple classes such as XYZ for as long as you wish without any side effect, objects like elements (and especially elements!) are valid in the API only as long as the corresponding native instance still exists in Revit.

To sum it up, access to elements ceases to exist when:

a) Element is deleted from the model
b) Creation of an element is undone
c) Deletion of an element is redone
d) Transaction during which an element was created was rolled back (i.e. not committed.) (Note: *The same applies for sub-transactions within a transaction.*)

On other hand, a deleted element can come back to life if:

a) Deletion of the element was undone
b) Creation of the element was redone

When an element is deleted, an API application should not attempt to access the element's properties. If such an attempt is made, Revit will throw an *InvalidObjectException*.

Note: The code guarding access to deleted objects works for all kind of objects, not just elements (try it with a document, for example), but the ability to bring an object back to life with undoing its deletion is only applicable to elements and their derivatives.

Note: It is probably obvious that should the above statements be true (and they are) one would not gain any memory benefits by deleting elements from the model. Deleted elements always wait for a chance to be resurrected. The cache of removed elements is cleared at some situations only, such as synchronizing with the central document in work-sharing environment, or when a document is closed.

Example using of an expired element:

```
private void ElementTest(Document doc, Element element)
{
   string name = element.Name;       // OK, accessing a valid element
   using( Transaction trans = new Transaction(doc) )
   {
      trans.Start( "Deleting" );     // allowing modifications of the model
      name = element.Name;           // ok, reading is allowed in a transaction
      doc.Delete(element);           // this deletes the element from the model
      try
      {
         name = element.Name;        // wrong! expect an exception!
      }
      catch( InvalidObjectException )
      {
         TaskDialog.Show( "Revit", "Accessing a deleted element." );
      }
      trans.RollBack();              // this effectively undoes the deletion
   }
   name = element.Name;              // OK; element's alive again
}
```

API Events

We are quite confident to bet that there are few Revit API developers who have not tried to use Revit events yet. We are not as confident when it comes to guessing how much our users know and do not know about the mechanism of Revit events. Sure, from the outside Revit events look and work much like any other events in the .NET framework. However there is a lot more under the uniform interface that probably deserves a closer look, mostly because – once again – Revit events are rather complicated a mixture of native and managed implementation.

Let's start with the obvious:

- Event handlers are called in a loop on first-come-first-served basis. Whichever event handler was registered first it will also be called first when the event is raised. There is no re-sorting of any kind.
- There are three kinds of events:
    1. Single events
    2. Pre-events
    3. Post events (always paired with the corresponding pre-event)
- Another way of categorizing Revit events is:
    1. Application-level events – they are either independent of any document, or they are raised no matter what the related document of the event is.
    2. Document-level events – they are only raised to those who subscribed to them for a particular document.
- Pre-event are raised before post-events. Post events are always raised, even when the corresponding pre-event cancelled the action, or if the action failed and was not finished.
- It used to be (pre-R2012) that document-level handlers were always invoked before corresponding application-level handlers, but that is no longer the case. Though all document-level handlers of an event will be called in one batch and application-level handlers of the same level in another batch, the order is not predefined. Like with individual handlers of an event, these groups will be processed on first-come-first-served basis. For example, if it was a document-level event a client first subscribed to, than it will be the batch of document handlers that will be invoked first.
- Some events are cancellable, some are not. Some events are never cancellable (post events), some are always cancellable, and some are cancellable conditionally. You can test the **`IsCancellable`** property in an event's argument to see if the action for which the event is being raised can be cancelled. To cancel the action, call the **`Cancel`** method, which is also available in the event's argument class.
- If a handler of a cancellable pre-event requests cancelling the action, other handlers queued up for the pre-event will not get called, but the related post-event will still be raised to anyone who subscribed to it.

Now let's dive a bit deeper under the hood

- There is a connection of managed events and their native brethren in Revit code. Internally, each event can have many observers, one of which serves all managed handlers of that particular event. There may be other observers for the same event in Revit - the API observer is just one of them. An API observer (of a particular event) is only instantiated when there is at least one API handler registered for the event. The observer is destroyed after the last API event handler unregisters.
- Depending on the event and situation, event handlers can use transactions, but they are not permitted to use modeless failure handling in transaction. Any attempt to do so will be simply ignored. This is for the sake of other handlers who would not be aware of a pending transaction hanging around and they could easily get an exception for doing nothing obviously wrong.
- Open documents other than the currently active document and the document of the event (if they are not the same) can typically be modified if the event handler wishes so. Other documents can also be freely opened and closed (with some exceptions – e.g. during the *Application Closing* event.) There is one restriction to when a non-active document may be modified during an event:

If an inactive document is already modifiable at the time an event is raised, the document is put temporarily into a read-only state where it stays for the entire durations of the event. That means such document will not be modified during the event.

- There is an API Firewall in effect around every call to every handler. The firewall is to make sure an API client does not leave open what is not supposed to be left open, such as an unfinished transaction or Edit Scope.
- Unsubscribing from events during events is possible. One can unsubscribe from any events, even from the one currently being handled. This comes quite handy particularly for the *Idling* event, which should be unsubscribes from it as quickly as possible.

Application vs. Controlled-Application events

I have been asked about this several times: "Does it matter if I subscribe to an event in the *ControlledApplication* object and how is it different from subscribing through the *Application* object?" The answer is: It does not matter. These subscriptions are virtually identical. Application event Subscribers are held at one common repository; there is just one common observer for both. One can even subscribe to an event in the application object and unsubscribe from it in the *ControlledApplication* object, and vice versa (which is more likely.) The effect is the same.

API Callbacks

Starting with R2012, Revit API introduced a new kind of objects – API Callbacks (a.k.a. API interfaces). Callback objects are delegates that are given to Revit to be executed later. A few examples of commonly used callbacks:

- Updaters (`IUpdater`)
- Transaction finalizers (`ITransactionFinalizer`)
- Failures processors and Failures preprocessors (`IFailuresProcessor`)

The callbacks are not pure managed objects. They always have their native implementation too. Revit actually calls the native part first, which then passes the calls onto the managed object (if is still exists) and returns back the obtained result, which may be both good and bad.

Callbacks are typically owned (held) by Revit for the necessary time being, but it is recommended that callers maintain their managed instances alive. When the native part finds the managed instance does not exist anymore, nothing should happen.

Dynamic Updaters

Like the transaction framework, the Dynamic Model Updates framework is too vast to be fully explained in this concise section. Here we can focus on just the details that are more likely to be overlooked elsewhere.

In a sense, Dynamic updaters are an extension of the internal regeneration process. Regeneration essentially makes sure that all parts of a model are valid according to dependencies set between related parts. Revit programmers have built Revit with certain rules about those dependencies and those rules may not be broken (the model would be considered invalid). So the rules may not be broken, but they can be extended, and that is when updaters come in.

- Updaters are invoked near the end of a committed transaction, but still before *DocumentChanged* event is raised.

  - The `Execute` method of an Updater will be called for every registered updater that got triggered by a particular change, depending on how the updater was set up.
  - Execute may be invoked more than once in a single transaction depending on whether the changes propagate to and back from other dependent elements in the model.
  - But there is a limit to the number of cycles (= total number of registered updaters + 2). Updates at the limit and above are considered illegal and those updaters which made them will be removed.
- Unlike the *DocumentChanged* event, dynamic updates are not invoked for undone and redone transactions. The same applies when a group is rolled back, which effectively undoes committed transactions it contains. In neither of these cases updaters are invoked, while a DC event will be raised.
- Unlike with events (any event), registering and unregistering updaters is not allowed during execution of a dynamic update, and it does not matter whether the registration is for the updater being executed or some other one. For the same reason, adding or removing triggers, and changing updaters' priority are also prohibited.

  Strictly technically speaking, it is not disallowed *per se*, but Revit would likely give an exception or would get into serious problems if someone tries to do so. Between us, unregistering updaters during execution of a dynamic update does make a little sense anyway.
- Obviously, trying to start or commit a transaction in the document that is being dynamically updated is not permitted, but it may be less obvious that committing transactions in other documents is not allowed either. Even though it is not technically disabled, updating will not work correctly due to the fact dynamic updates are not designed as re-entrant (currently).
- Updates provide access to "changed" elements:
  - Added and Deleted sets rule each other out – an "Added" element cannot be among those "Deleted", and vice versa.
  - Changed elements were neither added nor deleted.
- Certain API methods are forbidden in updater execution. The methods have comments about this in their description, but we can summarize the kinds of methods:
  - Methods that need a transaction internally (some exports), or methods that must be out of a transaction (Save)
  - Methods that introduce elements interdependency. This limitation is to avoid synchronization problems in work-sharing environments.
  - UI methods (picking, selections, etc.)
  - Transaction groups cannot be used either (obviously), but sub-transaction can if needed.
- Memory footprint and execution time of updaters is not limited, but it is wise to keep an updater light; do not invoke methods that take a long time (opening document, exporting, iterating through all elements in the document, etc.)
- Please be aware that by simply having an updater registered, even without triggers and therefore not being even invoked, will have influence of Revit performance. For that reason it is recommended (like with event handlers) to register updaters only when they are needed.
- And since you probably wanted to ask, yes, it is possible to find out in an updater's execution what trigger was it that actually got hit. It is what the **`IsChangeTriggered`** method is for. Keep in mind it only makes sense to use it if there is more than one trigger on an updater.

Dynamic Updater (DMU) vs. Document Changed Event (DCE)

Since both DMU and DCE provide ways for a developer to be notified about changes in the model, it is not a surprise that many developers have started using them to achiever their objectives. Unfortunately, sometimes the wrong tool gets used to do a particular job. Though both DMU and DCE are similar, they are not the same (both on the outside and under the hood), and they are not interchangeable. They both have their pros and cons. By knowing those, developers can apply the right tool to achieve their goals, which in turn makes their applications simpler and less error prone.

Before we get onto general recommendations, let's summarize main differences of the two tools:

	DMU	DCE
Relation to transaction and regeneration	Updaters are essential part of a transaction and regeneration. They add onto the rules Revit has about validity of a model.	Is not (practically) part of transaction and happens only after completing all regenerations.
Trigger actions	It only runs when a transaction is actually committed. It is not executed when a transaction is undone / redone.	It runs when transaction is either committed or rolled back and also when it is undone or redone.
Scope of changes	Invoked for desired changes only, controlled by filters.	Invoked for all changes (even for non-element objects).
Document Modifiability	User can make changes (as long as they do not compete with Revit's rules or some other application's rules).	User may not make any changes to the model during the event.

So, even though both DMU and DCE are kind of close in what they do, they do server different purpose, which should be kept in mind when designing an application that utilizes them. There are a lot of tricky and sophisticated applications out there (to which no standard concepts apply), but for the average application the high-level objective can be summarized as follows:

Time to use DMU	Time to use DCE
When you need to react to changes in the model by **making other changes to the same model**. In other words – you are adding rules about what it means for a model to be technically correct.	When you need to react to changes in the model by **modifying external data or another model**. In other words – you need to keep a Revit model in sync with something on the outside (of the model). It could even be the UI.

--- THE END ---