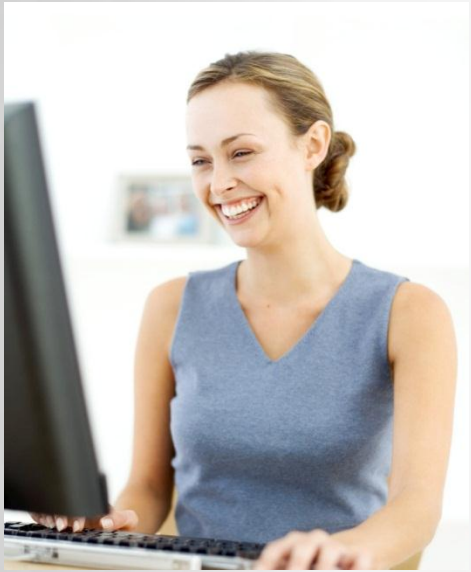
Facing the Elephant in the Room: Making Autodesk® Revit® Add-ins That Cooperate with Worksharing

Scott Conover
Software Development Manager, Revit API & Interoperability

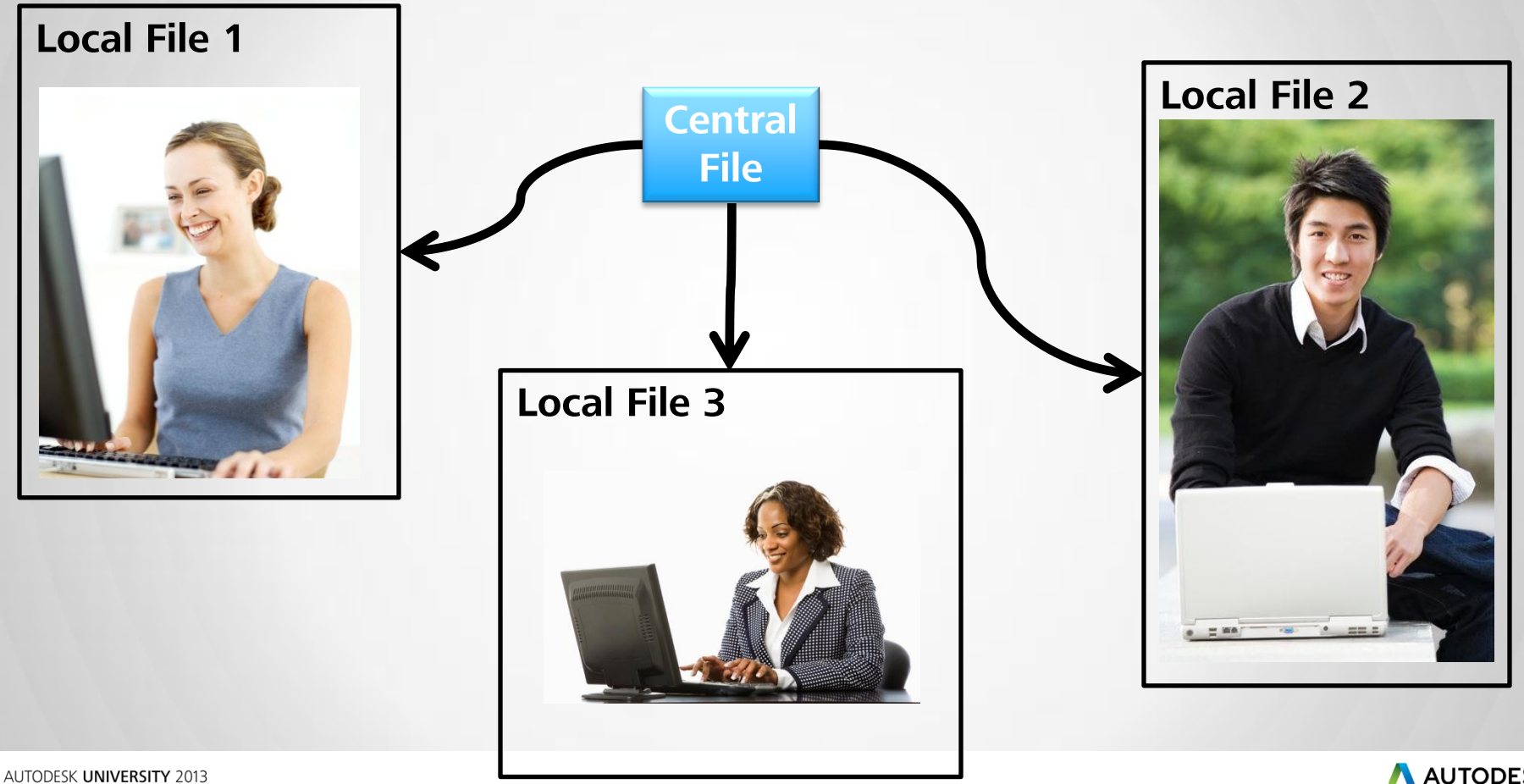
Typical thinking: add-ins & Revit



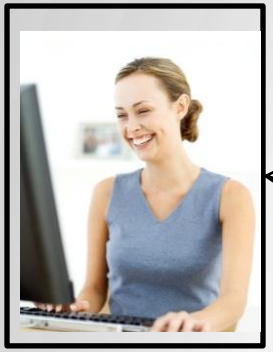
- One user diligently trying to accomplish their tasks
- We can help them get things done… with a killer add-in!

Automations	Standards	External data connection
Custom data	Events & updaters	Custom UI

But as add-in developers we don't always consider:



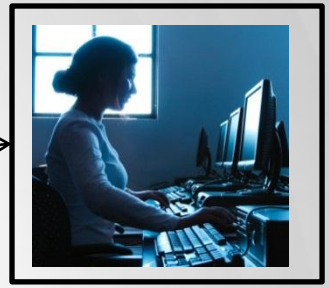
Or the even more complicated reality:

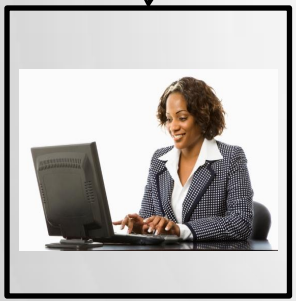


Revit server

Accelerator

Accelerator





Learning objectives

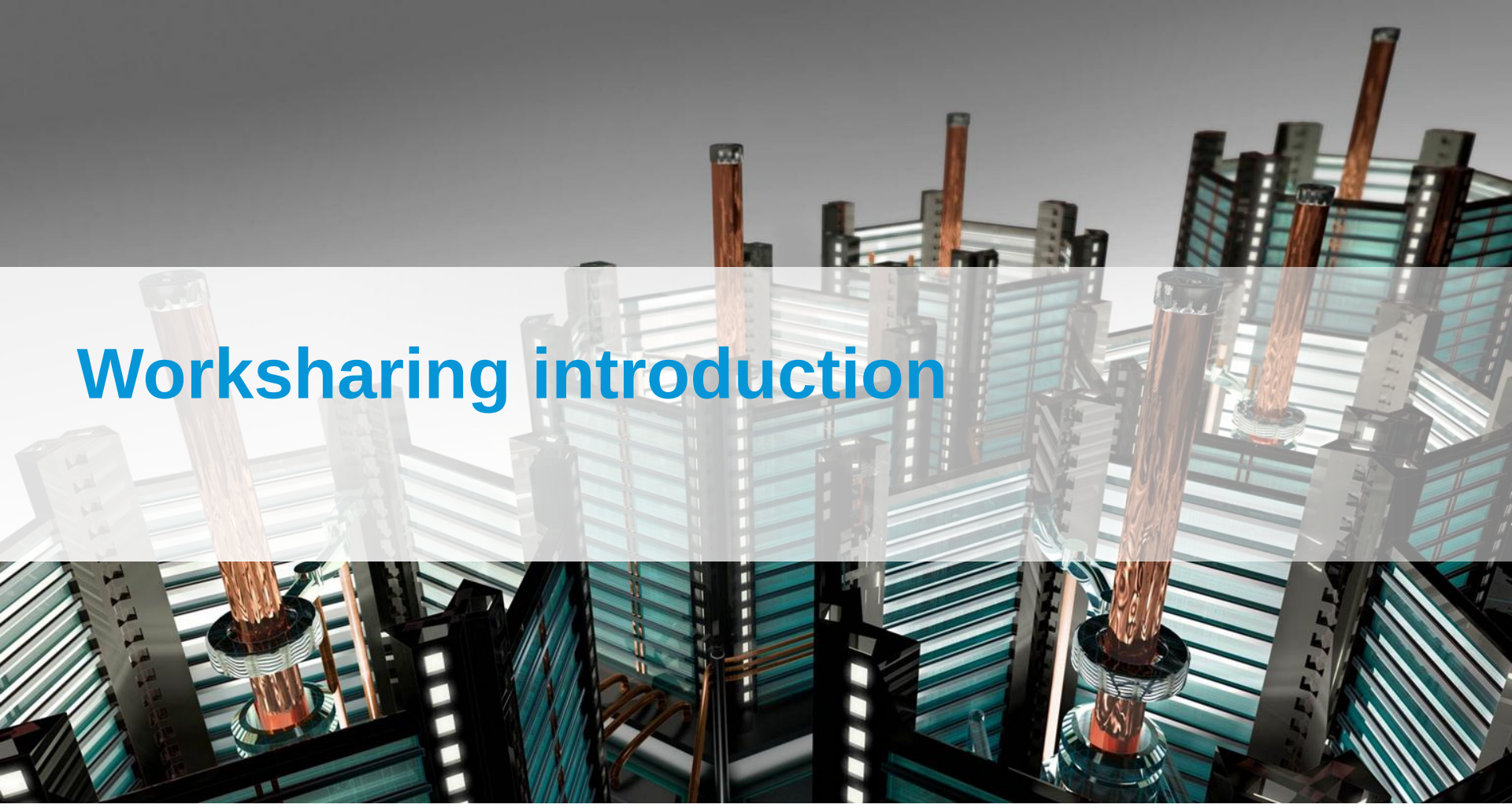
- Understand the basics of Revit worksharing techniques and terminology
- Discover the basic classes and methods related to worksharing in the API
- Enumerate suitable techniques to make a Revit add-in to work well in a workshared environment
- Identify the techniques related to management of projects, files and servers

Agenda

- Worksharing introduction
- API and worksharing basics
- Editing elements via API
- Document operations

Some notes on samples

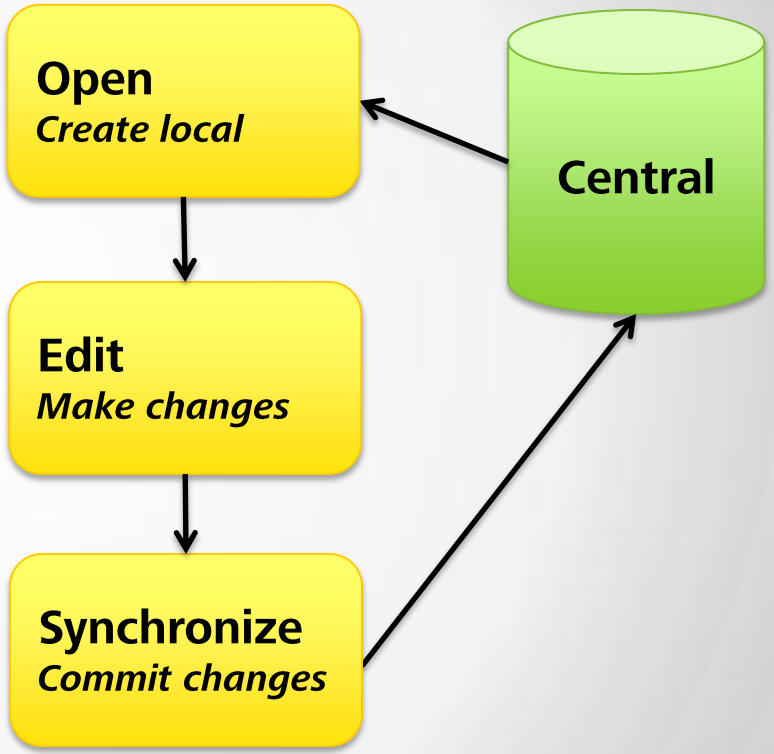
- This presentation incorporates many samples
  - In some places, we'll review the actual code flow in the debugger
  - Elsewhere, we'll see the results but not review the source code in detail
  - All samples are available for download
- Techniques are presented as "best recommendations" not "Autodesk says"
  - Not all techniques shown today are proven to work in shipping add-ins
  - These may not be the only way to construct a successful add-in
- Consider attending DV3464-R "Making Autodesk® Revit® Add-ins That Cooperate with Worksharing: A Roundtable Session" at 1:00 pm today for more discussion



Worksharing introduction

Workflow in a workshared environment (10000 ft view)

- User opens central model and gets a “local” copy
- User edits elements
- Elements are “checked out”/ “borrowed”
- Elements are fully edited
- “Synchronize with Central”
- Other users can get elements up to date with “Reload Latest”



Of course there's more to it

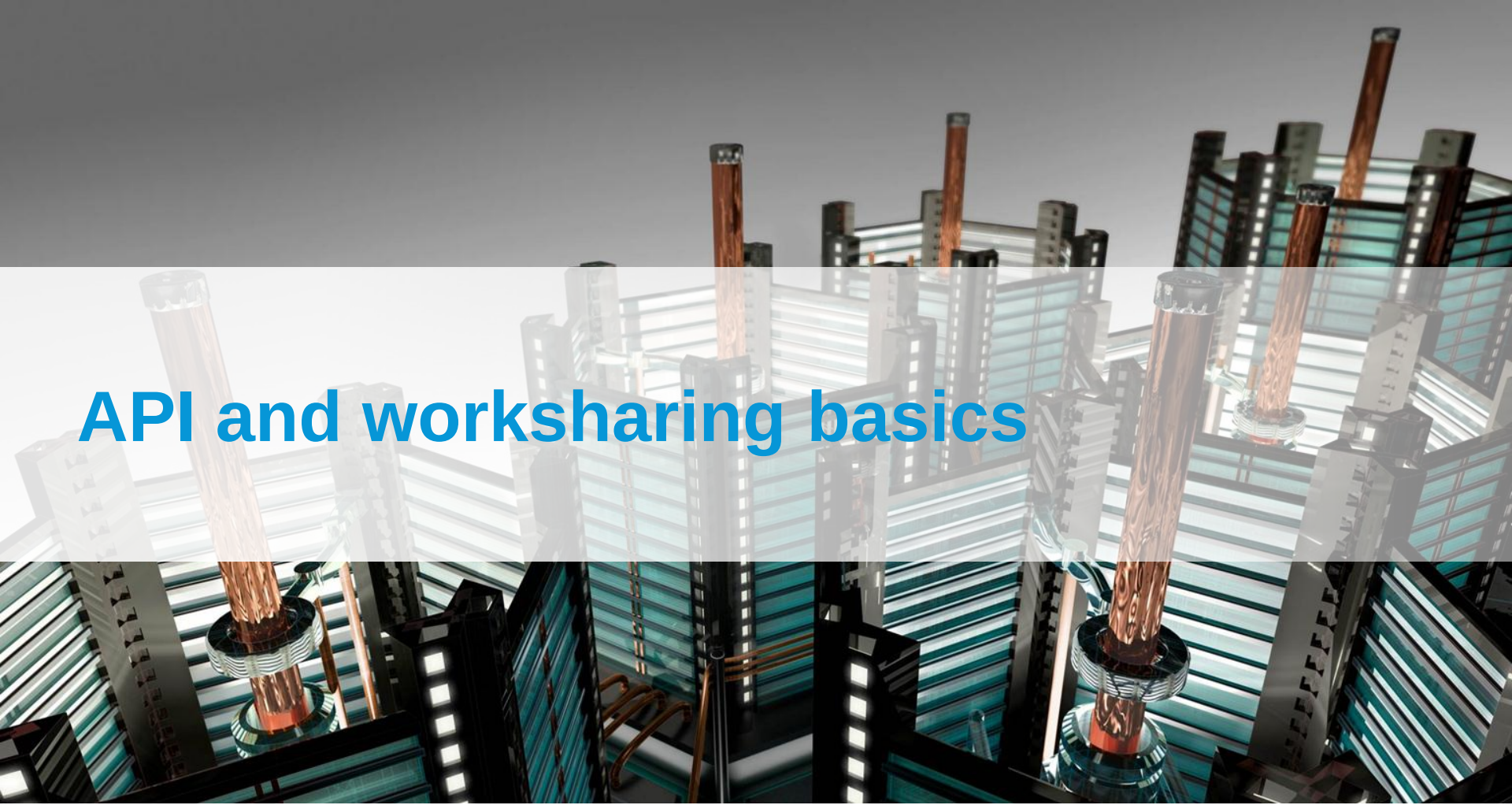
- Elements are placed in Worksets
  - Entire workset can be made "editable" – user has exclusive editing rights for all elements it contains
  - New elements are placed in the "active workset" in the user's local
  - Worksets can be "opened" or not – which affects visibility of elements, but the elements within are still available in the model
- Projects can be opened "detached" where there is no possibility to update the central with changes, and no workset management required

And to add to the complexity

- There are two types of worksharing
  - File-based – the central model is accessible on disk over the network
  - Server-based – Revit server manages the central model and possibly locally available accelerators

So that killer add-in you are writing needs to live in the workshared world….

- Consider that elements the add-in wants to change
  - May be checked out by others
  - May not be up to date
- Consider the impact changes your add-in makes on other users
- Consider interactions with project files that may be
  - Local
  - Central
  - Managed by Revit Server



API and worksharing basics

Workset and Workset id

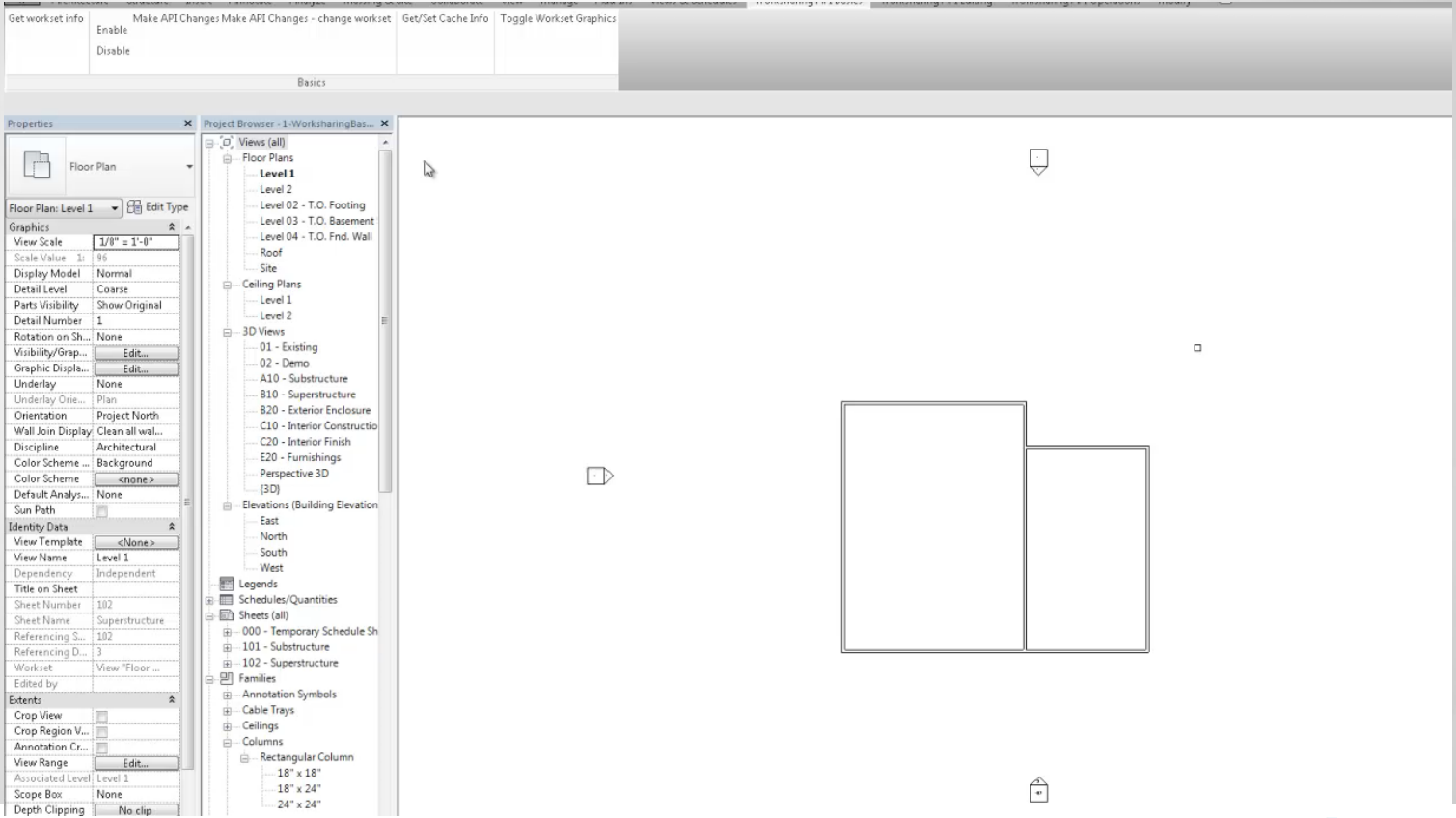
- Workset
  - Represents a workset in the document
  - Not an element, find these using FilteredWorksetCollector or WorksetTable
  - Read-only properties (Name, Owner, IsOpen, IsEditable, Id, UniqueId)
  - WorksetTable.GetActiveWorksetId()
- WorksetId
  - Integer id representing a workset
  - Read-only property of Element
    - Set this for an element using parameter ELEM_PARTITION_PARAM
  - Find elements with a given WorksetId using ElementWorksetFilter

WorksetKind

- A subdivision of worksets
  - User – user managed worksets for 3D instance elements
  - Family – where family symbols & families are kept
  - Standard – where project standards live including system family types
  - Other – internally used worksets which should not typically be considered by applications
  - View – contain views and view-specific elements
    - 2 different worksets per view, one containing view and related elements, the other containing view-specific annotations, dimensions and similar elements

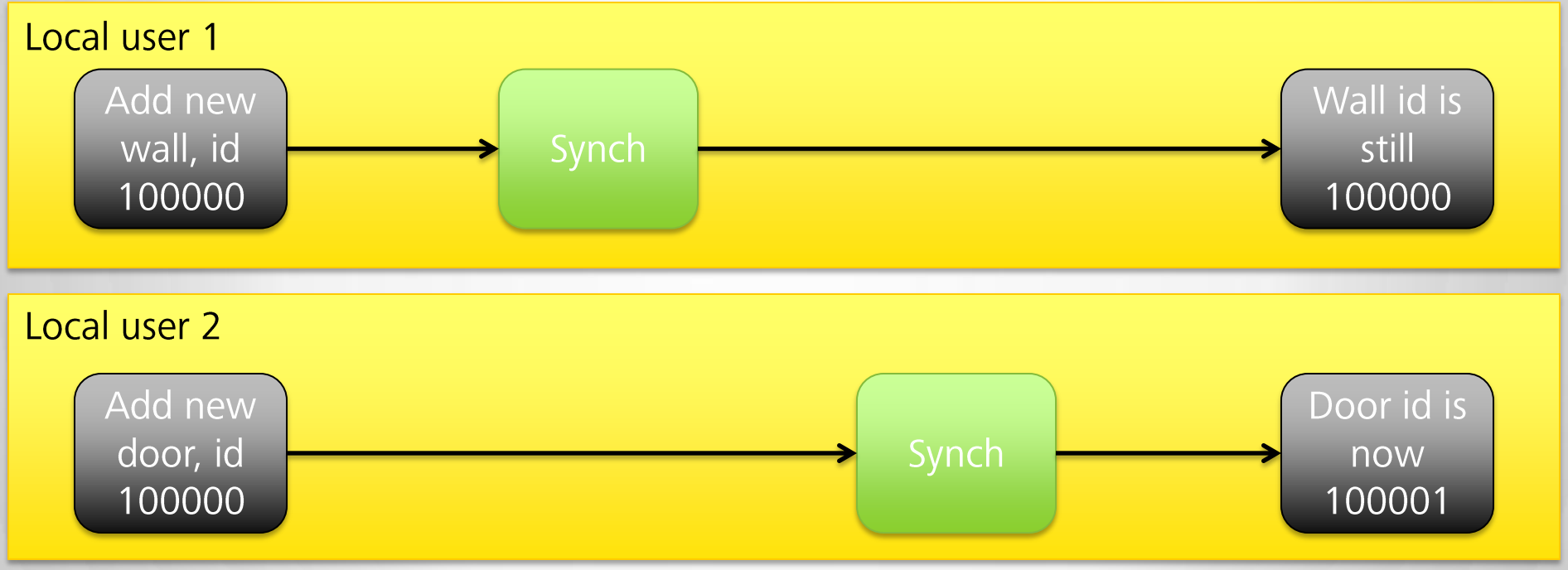
Worksharing basics examples

Look at worksets and the elements they contain, and set a newly created element's workset



ElementIds in a workshared environment

- ElementIds are not stable in workshared environment

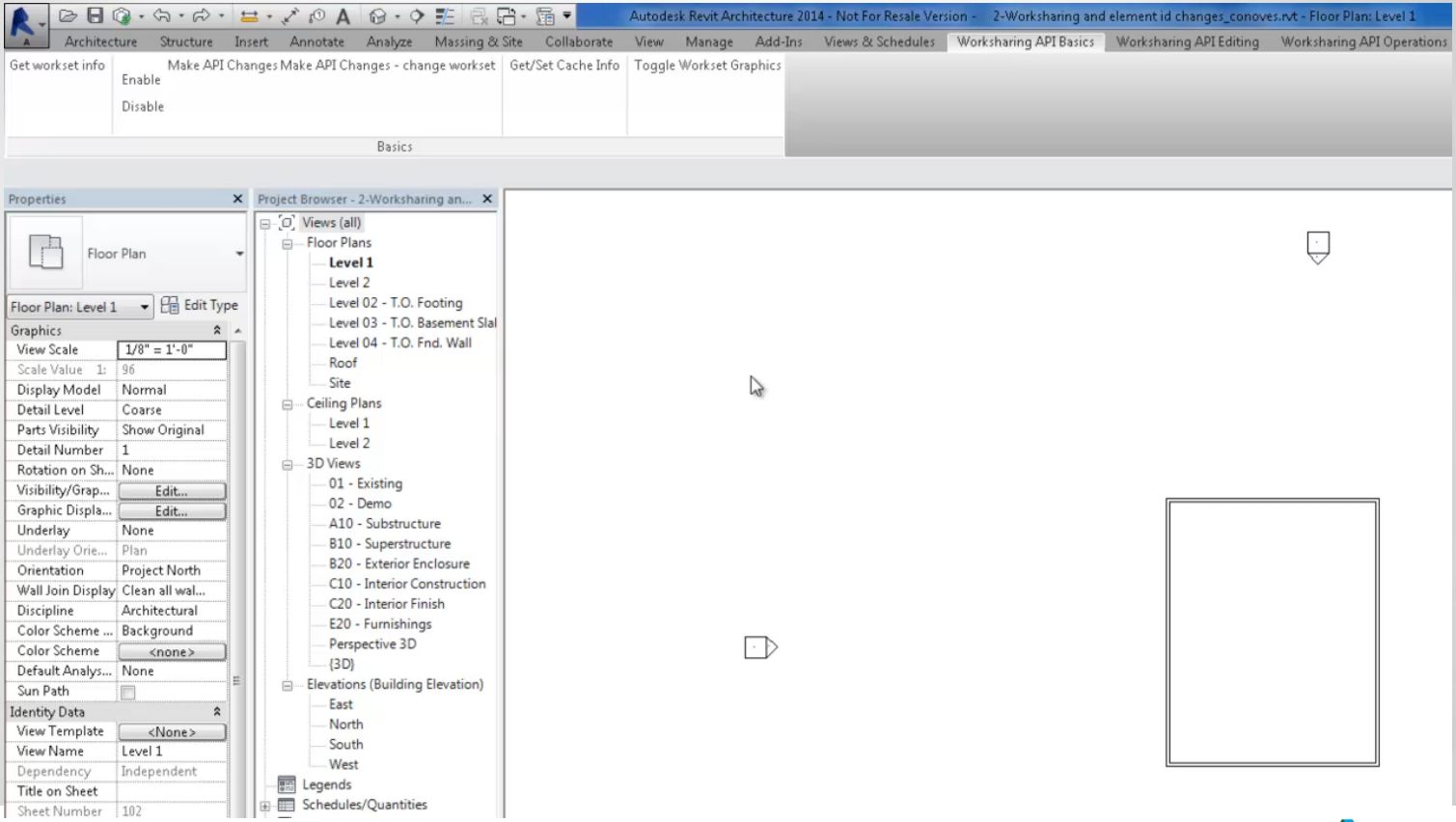


ElementIds in workshared environment

- Do not
  - Cache element ids for newly created elements in your add-in
  - Store element ids outside of Revit
  - Store element ids as text or integer values, even inside Revit
  - Rely on cached Elements either (cache may not fully update)
- Do
  - Store element ids as ElementId parameter values
  - Store element ids as ElementId fields in extensible storage – Revit will remap automatically
  - Use Element.UniqueId to cache ids or store outside of Revit

ElementIds in workshared environment example

- Caching element ids

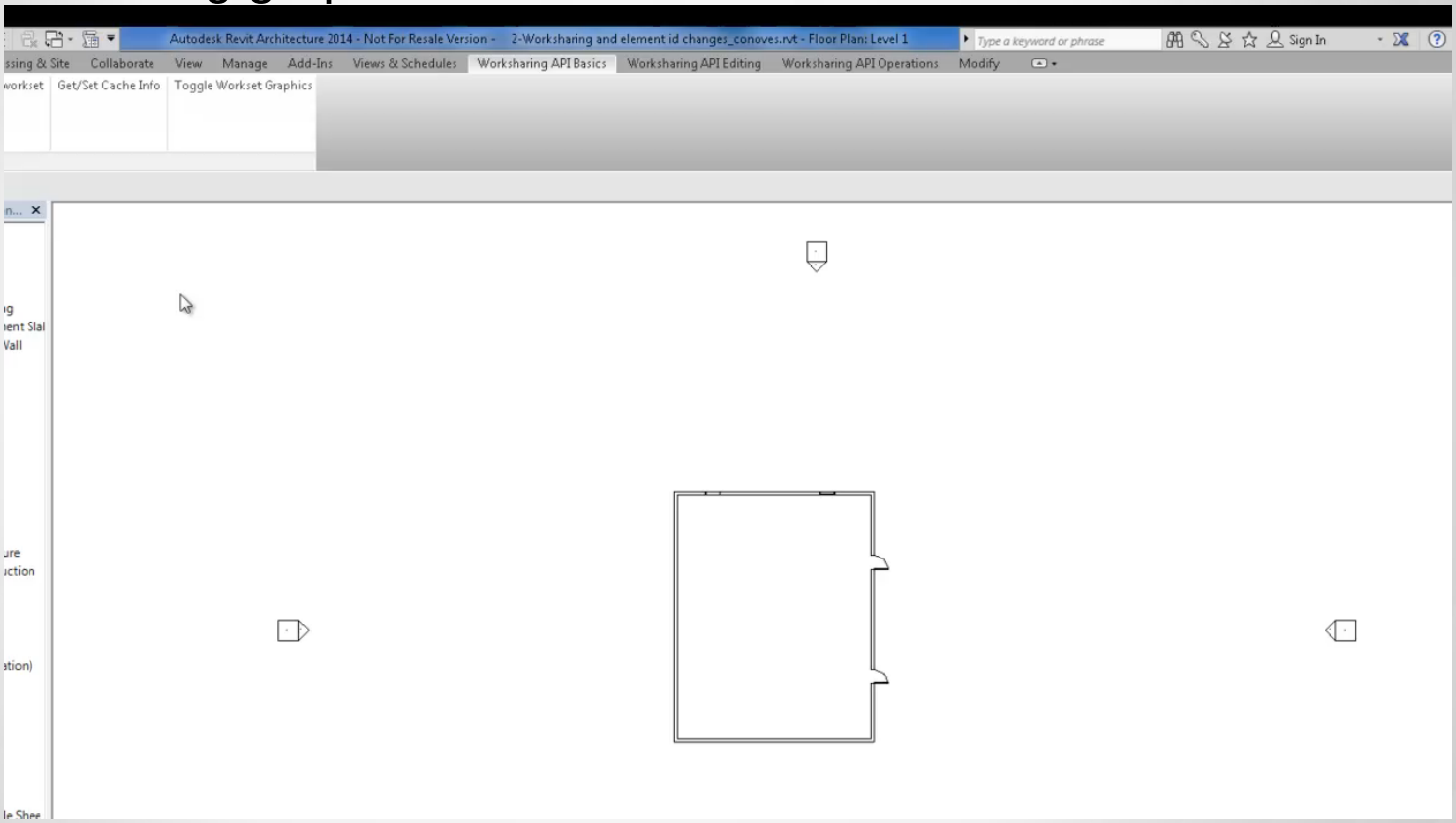


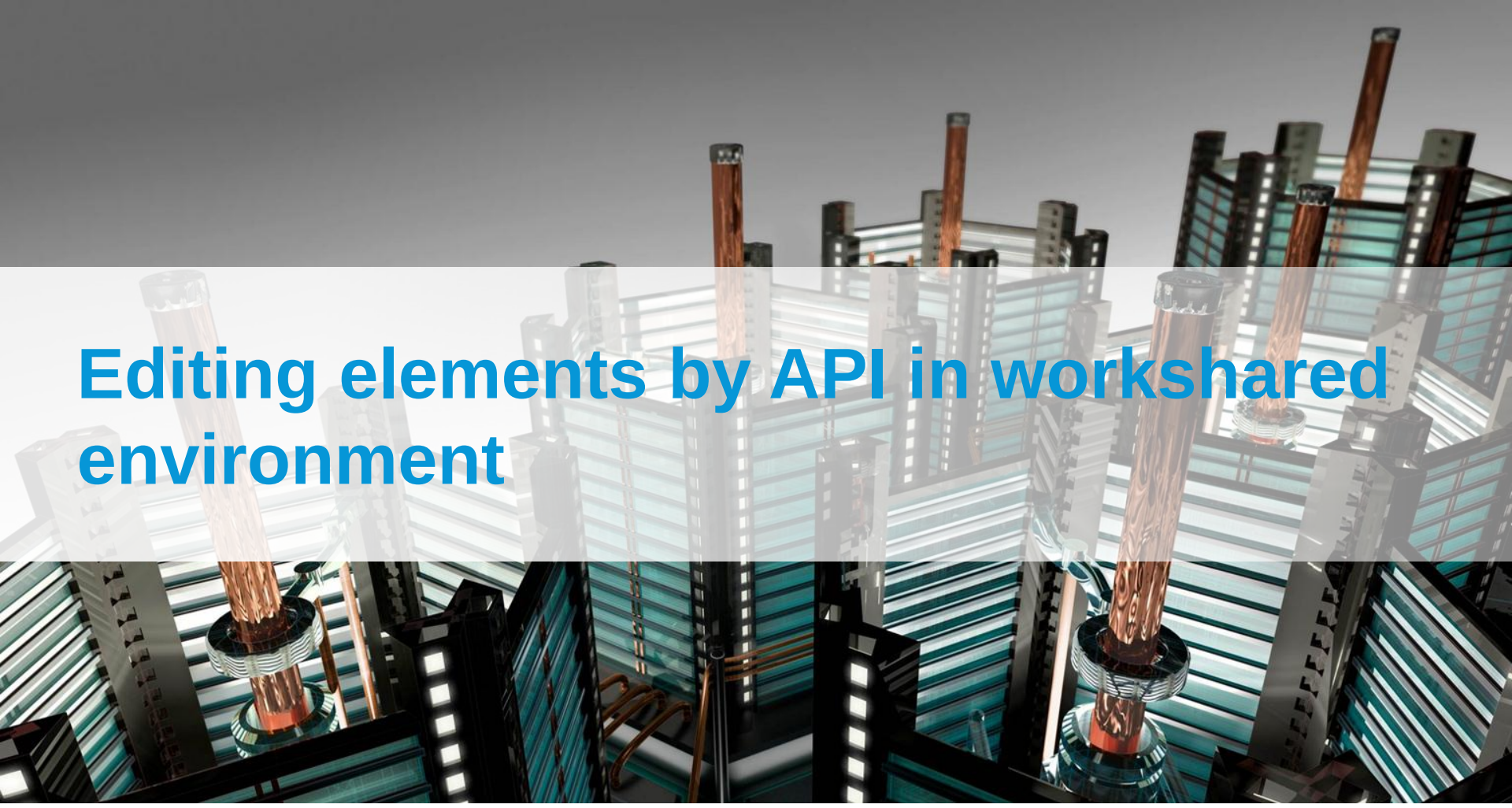
Worksharing graphics settings

- Graphics settings
  - WorksharingDisplaySettings
    - Remembers colors applied to different settings
    - WorksharingDisplayGraphicsSettings
  - View.SetWorksetVisibility()
- Tooltips
  - WorksharingUtils.GetWorksharingTooltipInfo()
    - This is where you can find the name of the borrower of a given element

Worksharing graphics settings example

Toggle worksharing graphics



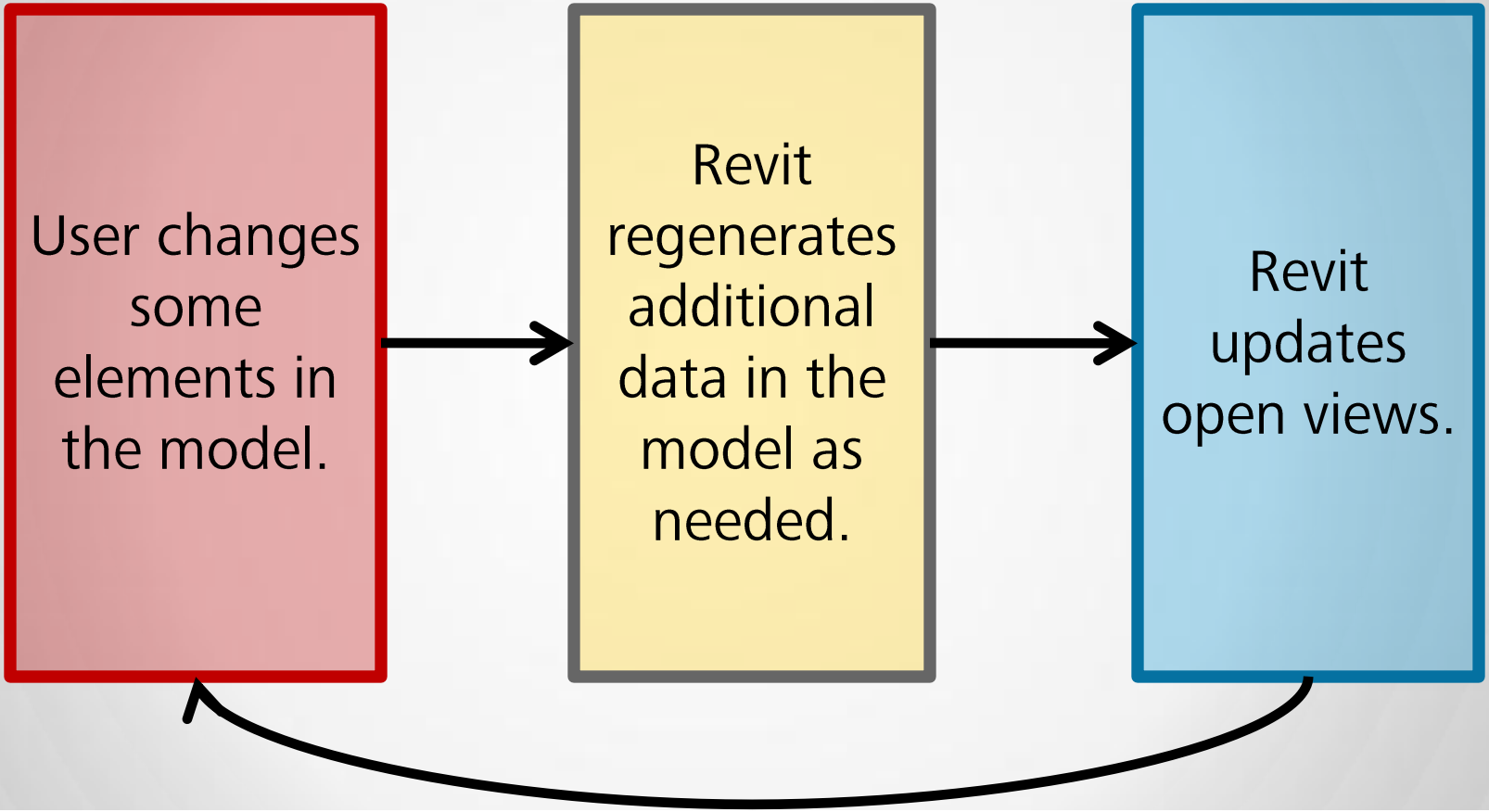


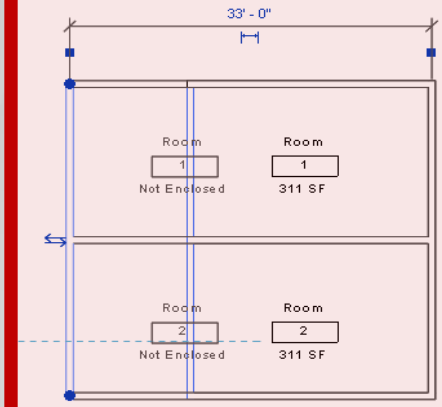
Editing elements by API in workshared environment

Impact of add-ins on workshared environments

- Users working in teams can encounter roadblocks and usability issues with add-in features beyond what a single user might experience
- How you architect an add-in can impact how well that feature works in a team environment
- In particular, the architecture can protect users from, or lead users toward, the dreaded “editing conflict”

Basic model editing workflow





User moves a wall by dragging.

Joined walls extend

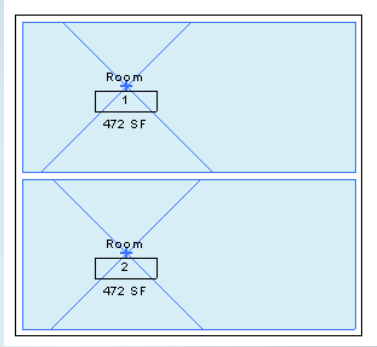
Roof updates

Floor updates

Rooms update

Room tags check if they're still in their rooms

Revit regenerates the model



Revit updates open views.

Why is this important

These changes are "**user changes**". The user must borrow these elements to make the change.

User actions

These changes are "**system changes**". Even though they are changed, they are still available to other users.

Regeneration

The content of views is not stored in the document. It is recreated when the view is opened.

View Update

Editing elements with API

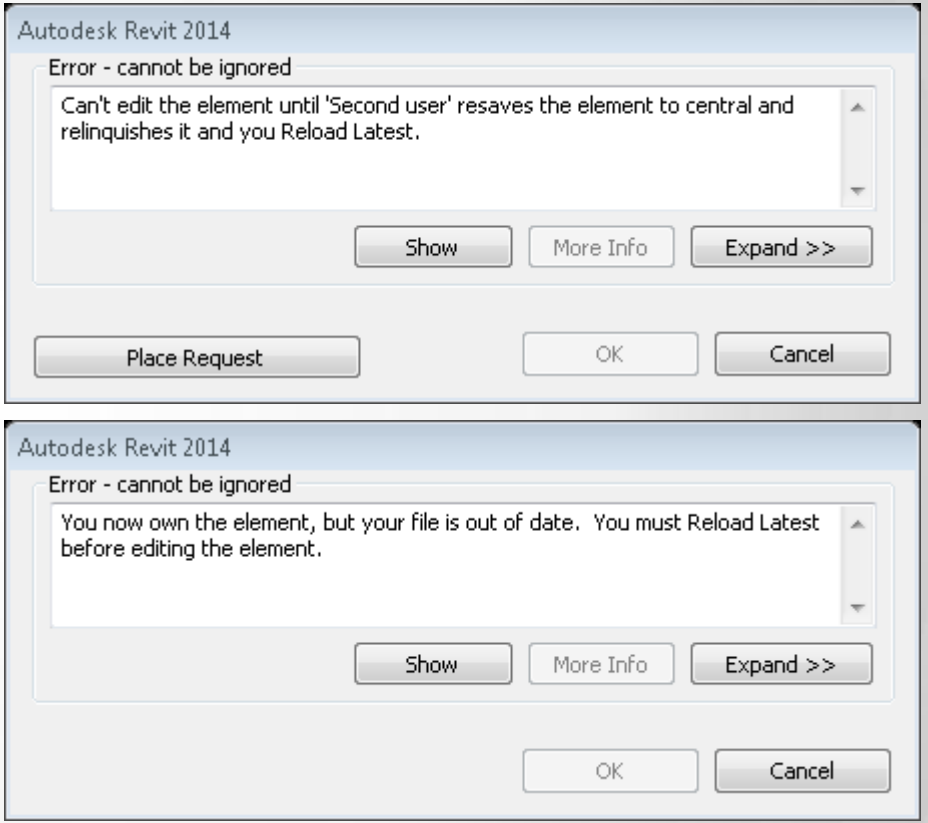
- API changes are user changes…most of the time
  - Treated as if the local user is making changes manually
  - This applies whether from External Command, macro, event, etc.
  - Exception: changes made from updaters (system changes)
- If the add-in tries to change thousands of elements
  - All these elements will be checked out to the local user
  - Not all elements may be available to modify because they are checked out
  - Either way, this may become a mess for users to deal with

Editing elements with API

- Techniques for API add-ins to make element changes
  1. Allow any errors to display to the user
  2. Make explicit changes, but catch and suppress the editing conflict errors automatically
     - Changes will not be made, but users don't see the errors
  3. Try to checkout elements in advance
     - Only make changes if all elements could be checked out
  4. Keep API data separate from standard elements where possible
  5. Don't make user changes, use system changes (via updaters)

Allow errors to display to the user

- No special API techniques required to make this happen
- Elements may not be obtainable, or not up to date
- Errors can result and will display to the user



Allow the errors to display

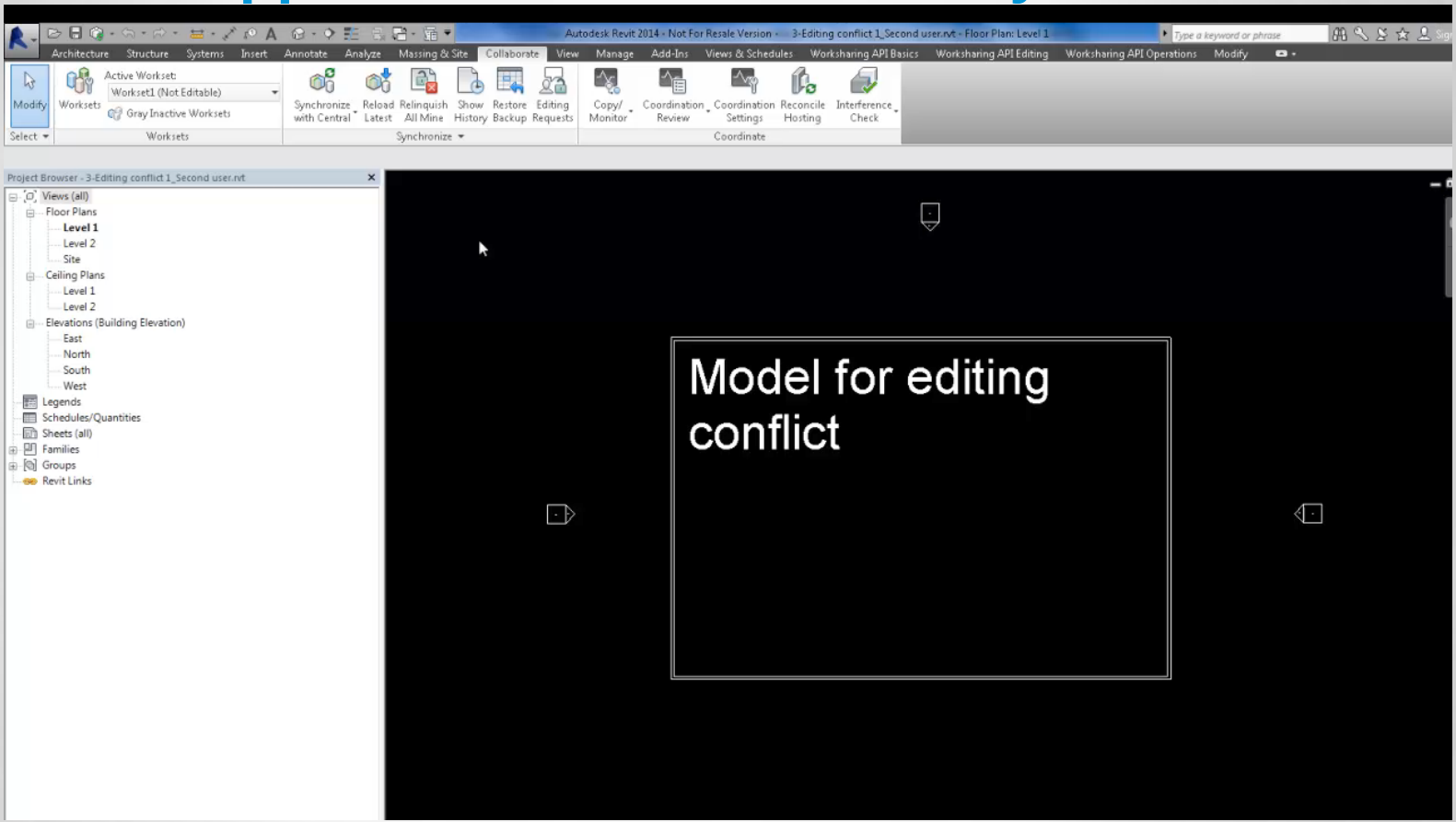
- Changes will not commit
  - Your code should check the status of Transaction.Commit()
  - Most samples do not do this
- Consider user experience when choosing this option
  - Should be obvious API tool was intended to edit element(s)
    - (Selection, tool name, etc)
  - Don’t use this option if resulting errors are unexpected/confusing
    - E.g. Editing a hidden element
    - E.g. Tool looks like a “read” tool but will also make changes

Catch and suppress errors automatically

- Setup FailureHandlingOptions in transaction
- Custom IFailureHandlingPreprocessor
  - In the preprocessor, if error is a worksharing error, return Rollback
- SetClearAfterRollback(true);
- Should typically include some error messaging to the user (tailored specifically to the tool)

- BorrowElementsAtRisk
- CannotBorrowBusyContention
- CannotCheckoutWorksets
- CannotEditDeletedElements
- CannotEditDeletedWorkset
- CannotEditEditingElements
- CannotEditElements
- CannotEditRenamedWorkset
- DeleteRequestedWarn
- HaveNoPermissionToEdit
- OutOfDateElements
- OwnedByOther
- OwnElementsOutOfDate
- ReloadBeforeMakeEditable
- ReloadNoGood
- TemporaryChangesOutOfDate
- TemporaryChangesTransparentPermissions
- UnableToMakeSourceWorksetEditable
- WorksetHasChangedInCentralFile

Catch and suppress errors automatically



Try to checkout in advance

- WorksharingUtils.CheckoutElements()
  - Return value indicates if checkout succeeded (if elements are in the list)
  - Performance intensive – try to combine multiple checkouts into one call
  - However, do not preemptively check out many elements which might not end up being changed – other users will be unhappy
- If successful to checkout, check if element is up to date
  - WorksharingUtils.GetModelUpdatesStatus()
- If checked out and up to date, proceed to edit

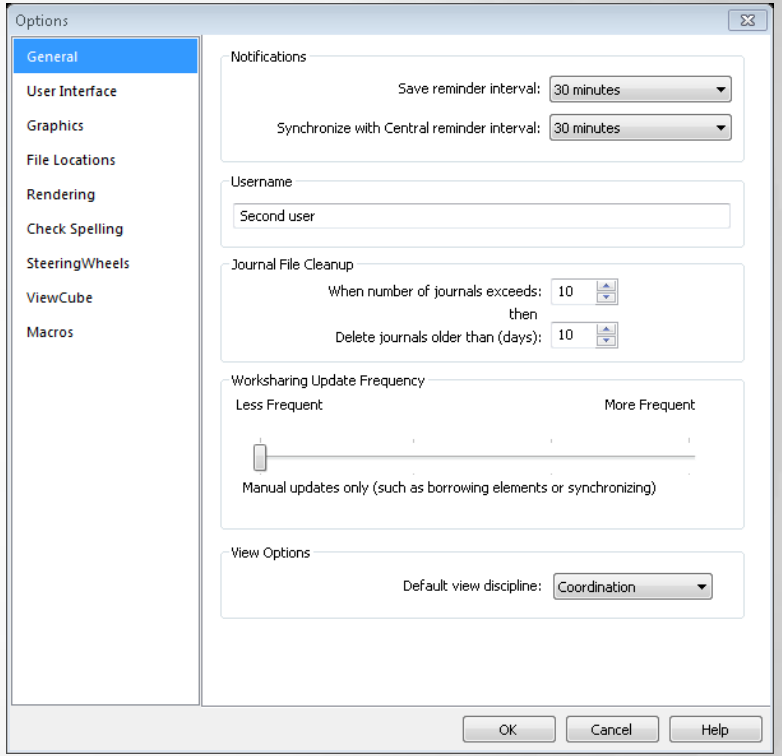
Try to checkout in advance

- Considerations
  - If the element is not checked out, but also not updated, the checkout operation will succeed, but you'll also need to reload latest before opening
  - Some Revit objects are not Elements and don't expose ids you can use to check them out (mostly these are settings):

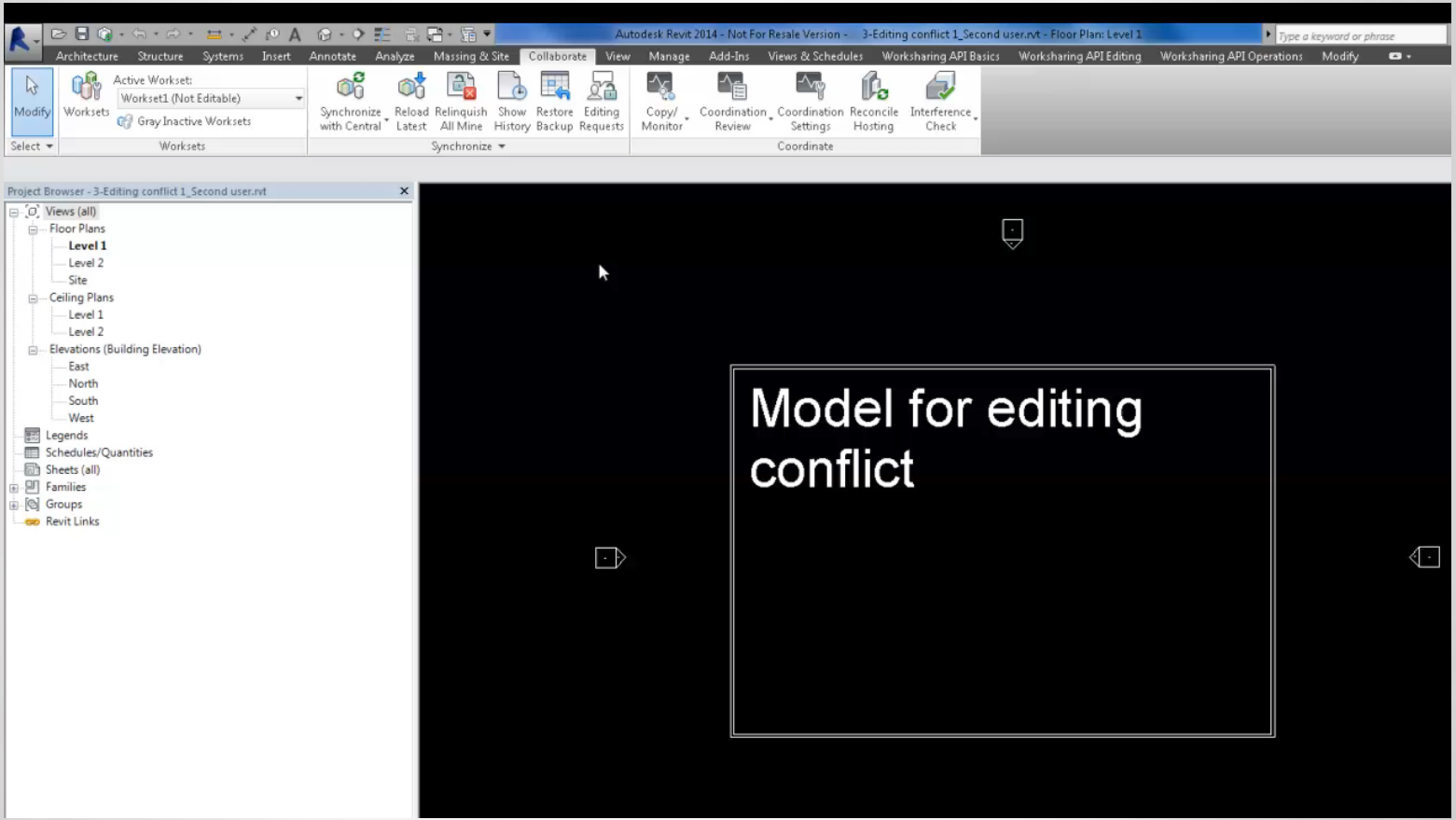
DefaultDivideSettings
StructuralSettings
ElectricalSetting
DuctSettings
DuctSizeSettings
PipeSettings
ReinforcementSettings
WorksetDefaultVisibilitySettings
RevisionSettings
ConceptualSurfaceType
EnergyDataSettings
StartingViewSettings
AreaVolumeSettings

Don’t bypass the checkout step

- Don’t use element status checks alone
  - WorksharingUtils.GetCheckoutStatus()
  - WorksharingUtils.GetModelUpdatesStatus()
  - Document.HasAllChangesFromCentral()
- Can return cached value depending on update frequency settings
- Return values not dependable in the middle of a local transaction
- These are intended only for “FYI” reporting where 100% accuracy is not required (e.g. via a mechanism like Worksharing display mode)



Try to checkout in advance



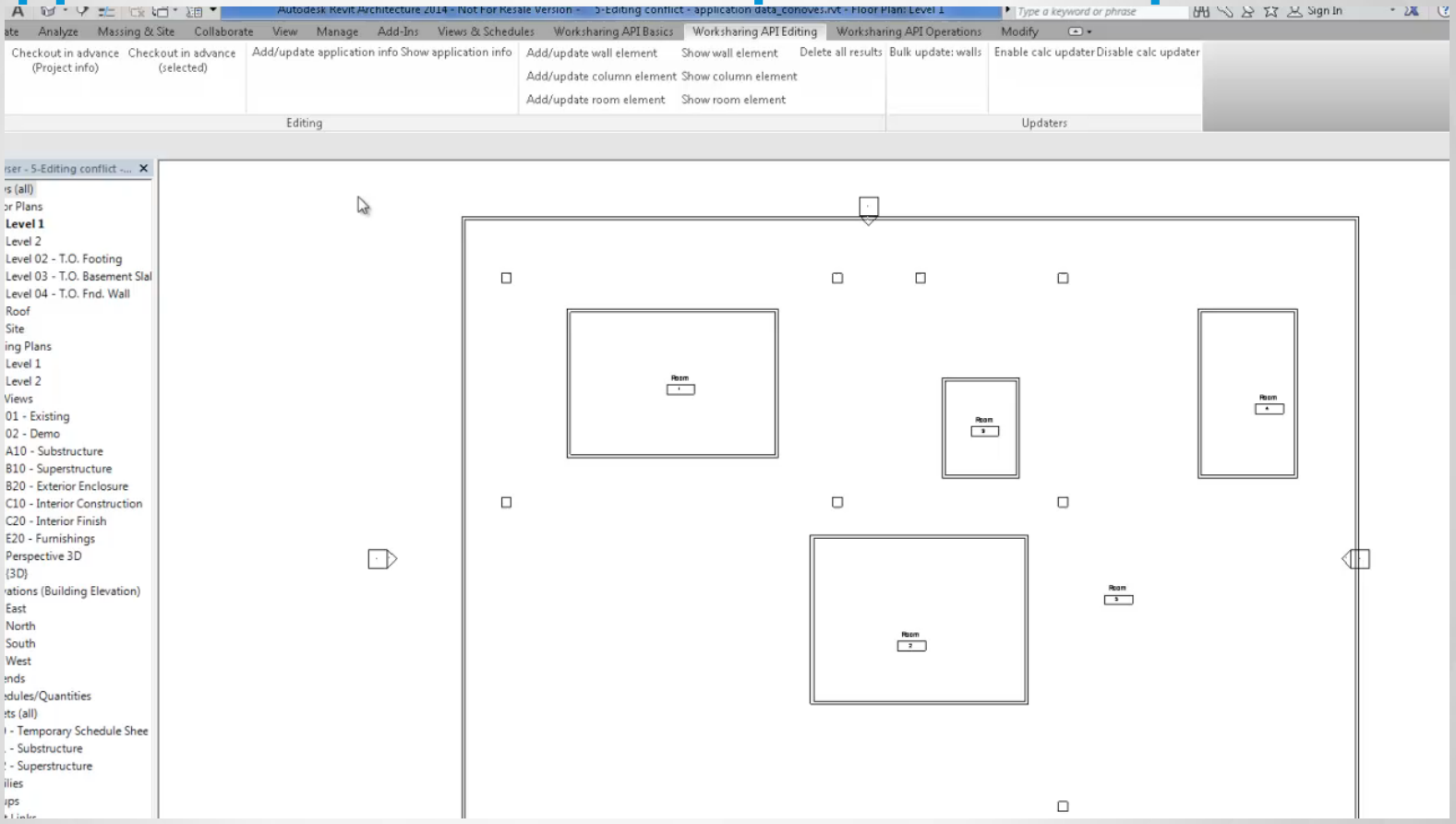
Keep application-owned data separate

- Minimize conflicts among user actions and among different instances of your own application
  - “Don’t store data on ProjectInfo”
  - Store application data external to user editable elements
    - DataStorage element and ExtensibleStorage
    - Use a schema including ElementId reference(s) to other elements
    - Revit will handle id updates

Keep application-owned data separate

- Options
  - Use one application-wide data element
  - Use several DSEs and schemas for unrelated data
    - This may be better to avoid conflicts between local users of different parts of your application
- Use care to populate elements initially to avoid duplication
  - (If element is added by multiple local users, all these elements will end up in central)

Keep application-owned data separate - examples



Make system changes instead of user changes

- Dynamic Model Update
  - Make changes in reaction to other changes
  - Changes made by Updaters are system changes
    - No need to checkout elements
    - No conflicts with user actions
    - If user action and system change is made to same element, only the user change is preserved

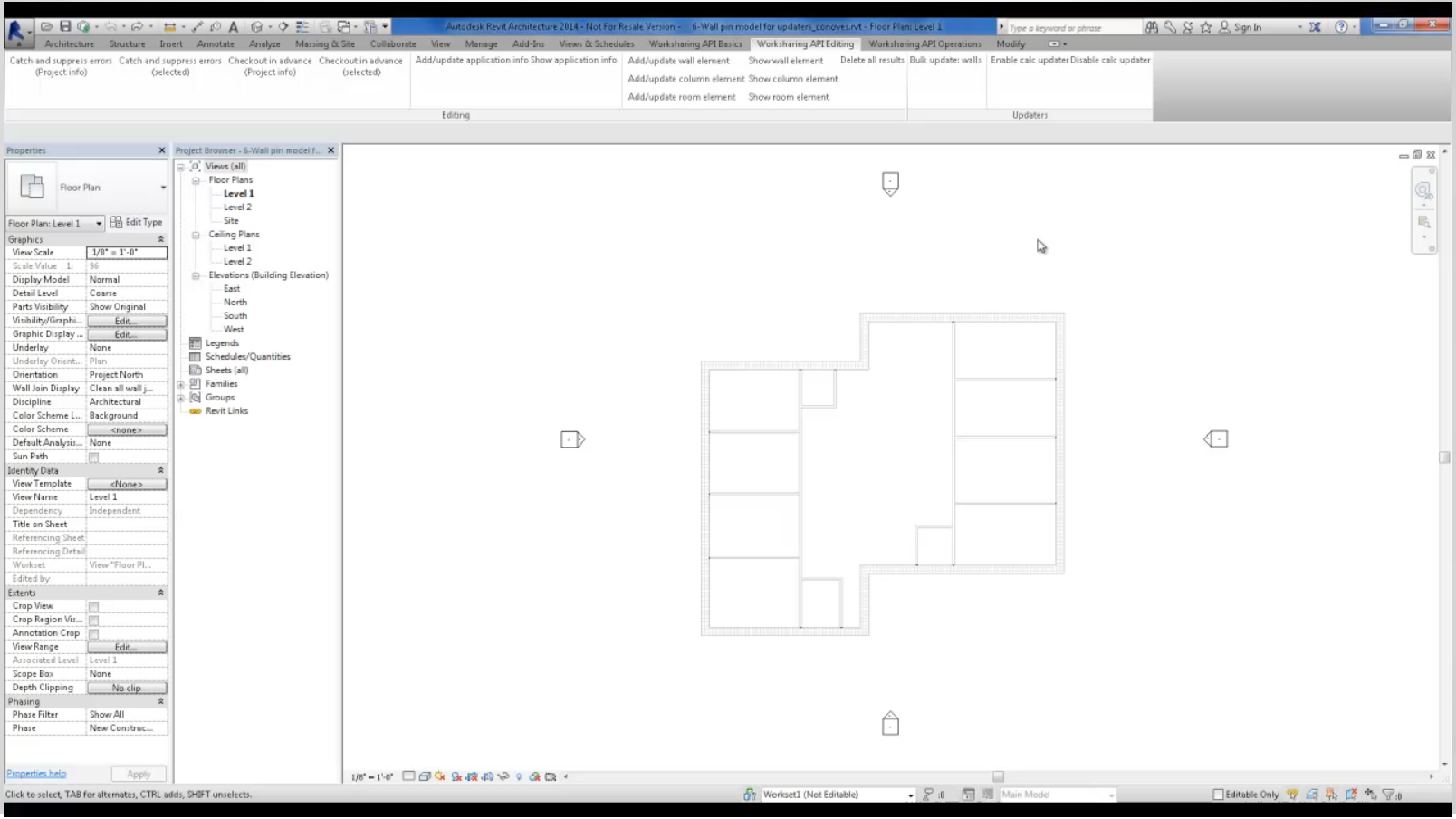
Make system changes instead of user changes

- If your add-in should modify hundreds of elements
  - E.g. set a parameter
  - Add or delete external data
  - Assign material
  - Switch type
- *And* if its OK if not all of the changes made will be preserved due to other users editing the elements
  - (Model is still “valid” even if all updates are not made)
- Use an optional updater to do a “bulk update” as a system change

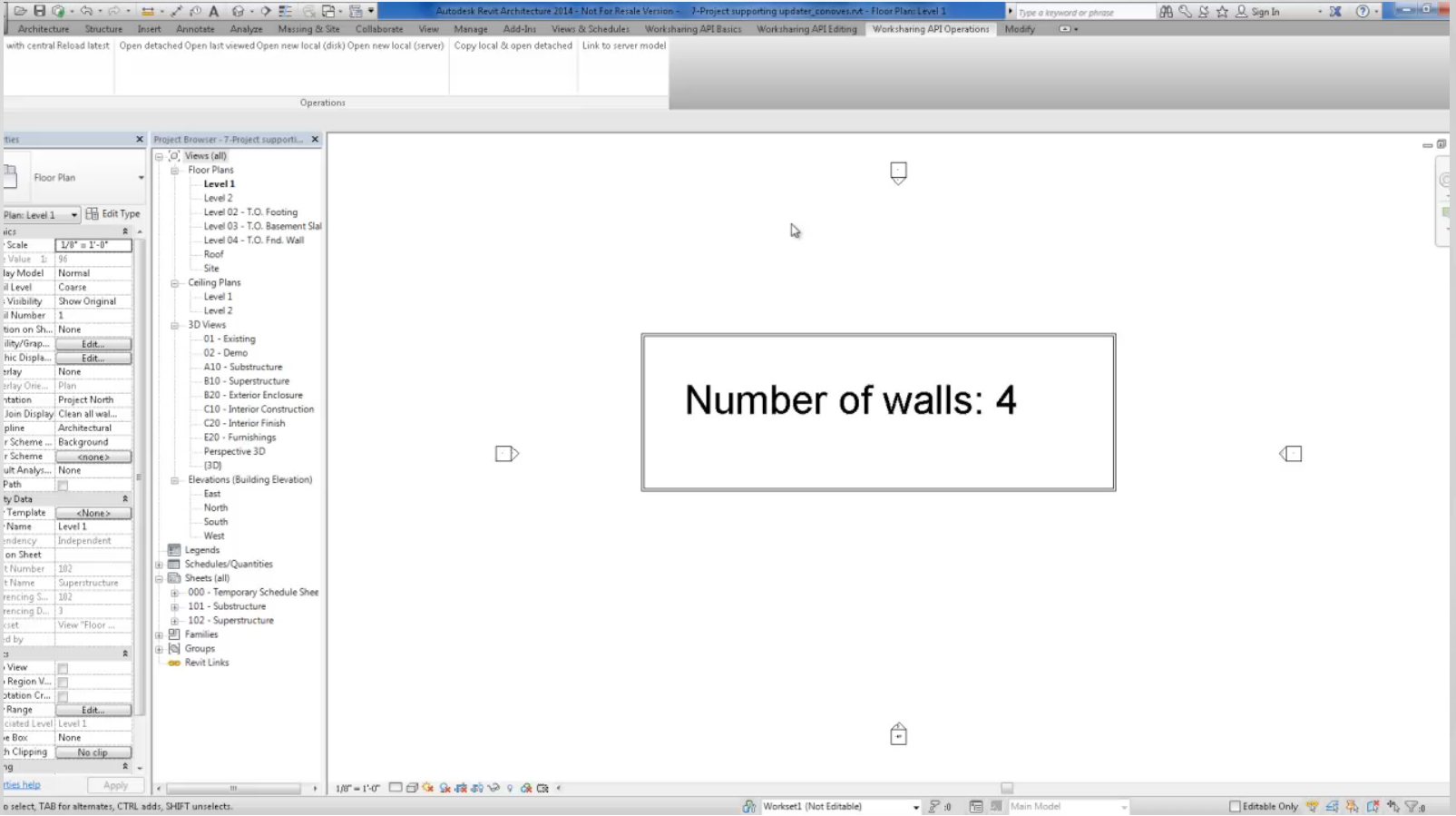
Make system changes instead of user changes

- If you are trying to keep elements in synch with user changes
  - E.g. calculated results derived from element data
  - Verification checks
- *And* you want these changes to be as automatic as possible
- Use (non-optional) updaters to drive the system changes
- Complication: reload latest
  - Does not call updaters, so your calculation may be out of date
    - Ask the user to manually run an update to recalculate…or…
    - Find some clever techniques involving DocumentChangedEvent
- Complication: all users should have updater installed
  - Warnings will result otherwise on file open

Use updaters – bulk update



Use updaters – synchronize calculations



Document operations

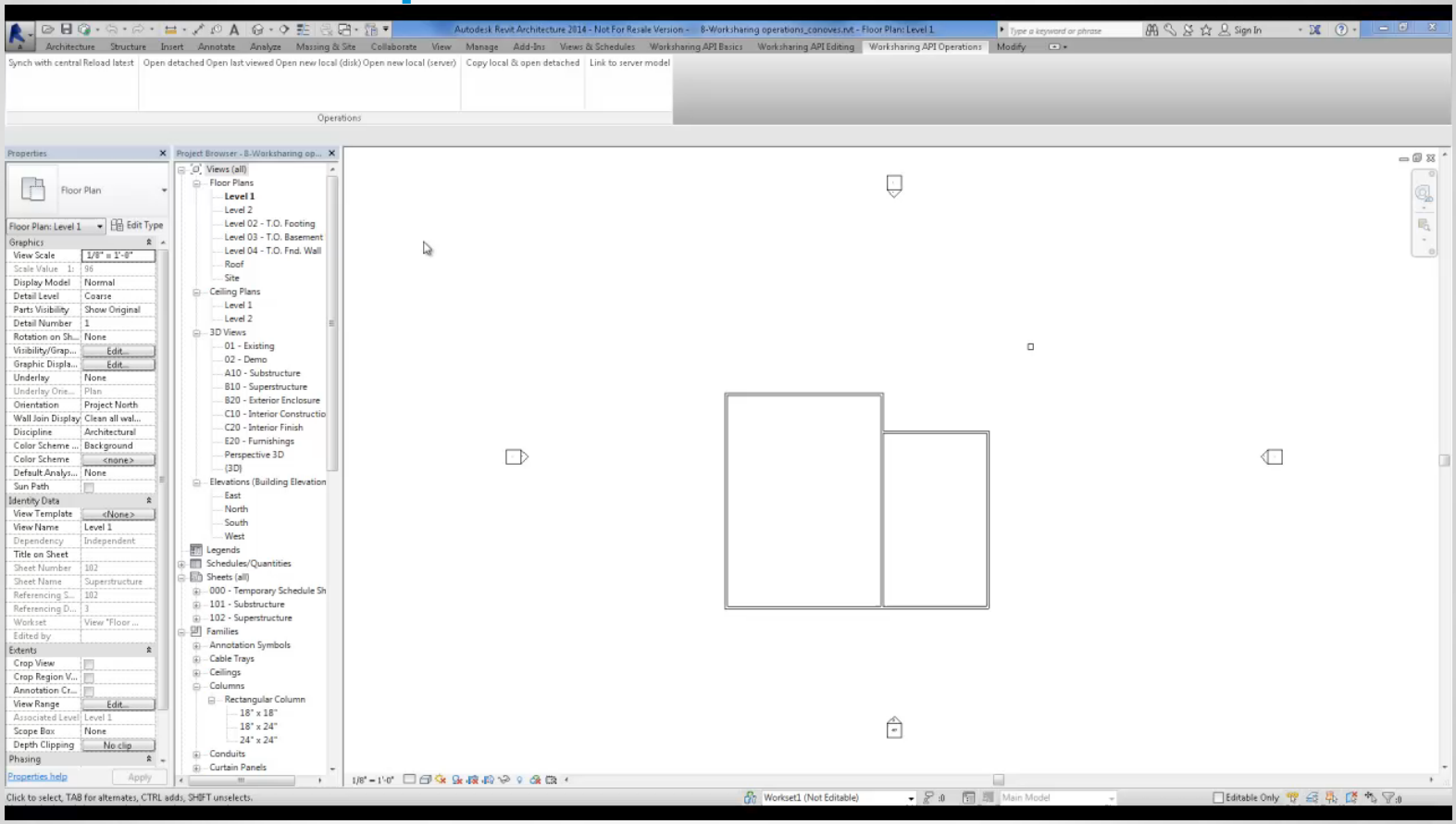
Synchronize with central

- Document.SynchronizeWithCentral()
  - Can be very performance intensive
    - If doing this as a “side effect” to your command, users should know about it up front
    - Remember: all user changes will be committed, not just those made by your add-in
  - Many documented exceptions for this command, related to central model interaction and failure points

Synchronize with central options

Option	Details
Comment	User comment for the synchronization
Compact	True to compact the central file, false otherwise (default)
RelinquishOptions	Options to relinquish ownership of different classes of elements and worksets, including checked out elements, family worksets, standard worksets, user worksets and view worksets. By default, everything is relinquished.
SaveLocalBefore	True to save the local before synchronizing (true is default)
SaveLocalAfter	True to save the local after synchronizing (true is default)
TransactWithCentral Options	Allows for a callback in case the central is locked by another user, to change the default behavior of waiting for release

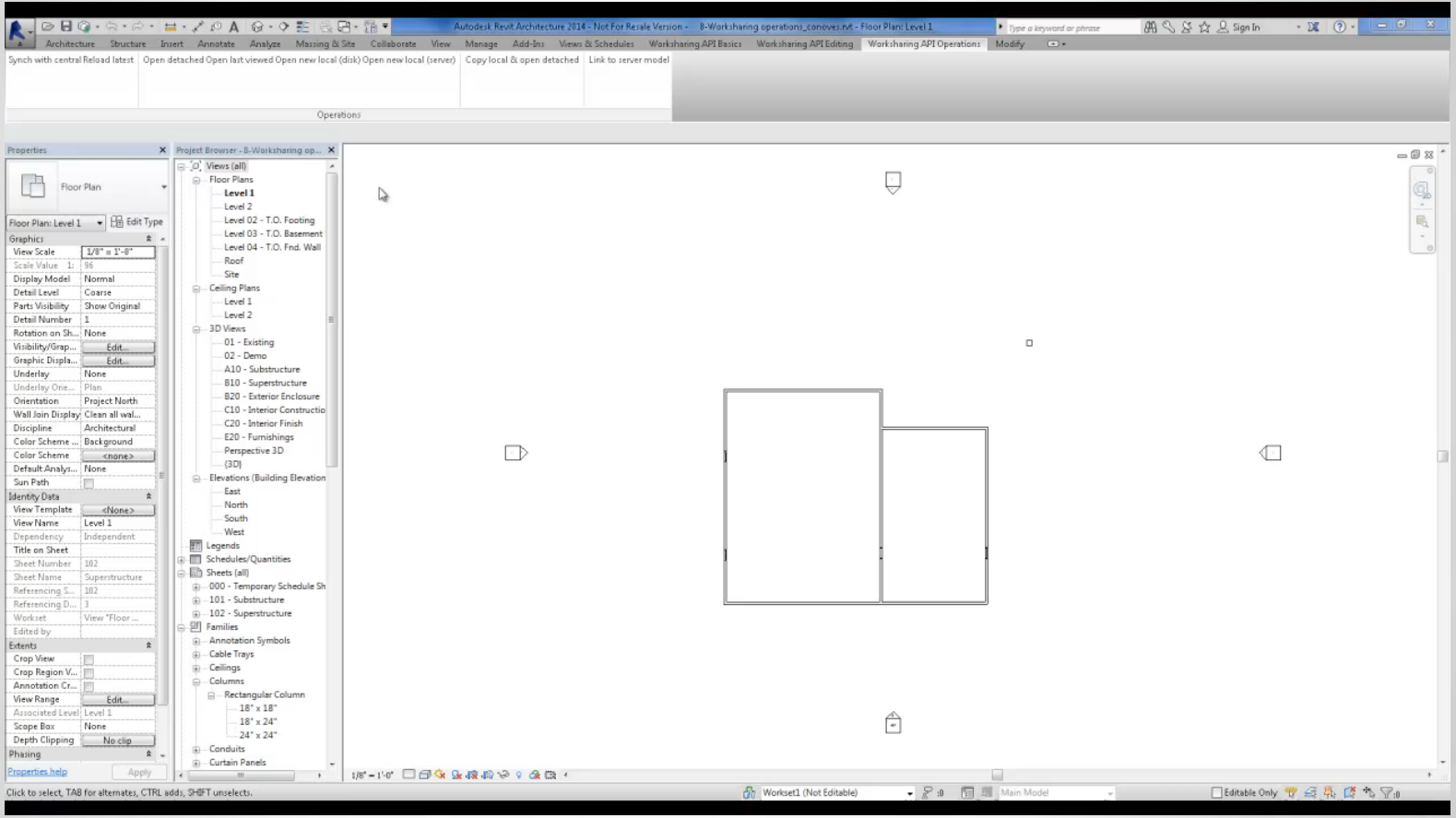
Synchronize example



Reload Latest

- Document.ReloadLatest()
  - Can be very performance intensive
    - If doing this as a “side effect” to your command, users should know about it up front
    - Remember: all updated elements will be brought locally, not just those needed by your add-in
  - Need to supply a default ReloadLatestOptions object
  - Many documented exceptions for this command, related to central model interaction and failure points

Reload latest example



Relinquish ownership

- WorksharingUtils.RelinquishOwnership()
  - Relinquish ownership of
    - Checked out elements
    - Specific types of worksets (user, family, view, standard)
  - Only unmodified elements are relinquished
  - Grants other users access to these elements and worksets

Before you open projects

- User name
  - Identifies the “user” who is editing elements and worksets
  - Defaults to the Windows login for the current user
  - Get it via Application.Username
  - No set API available in-session, but you can set it in Revit.ini before you start Revit

Before you open projects

- Determining options
  - BasicFileInfo.Extract()
    - Read information from the project before it opens
      - IsWorkshared
      - IsCentral
      - IsLocal
  - WorksharingUtils.GetUserWorksetInfo()
    - WorksetPreview
      - Reads names and ids of user worksets
      - Useful for setting worksets to open

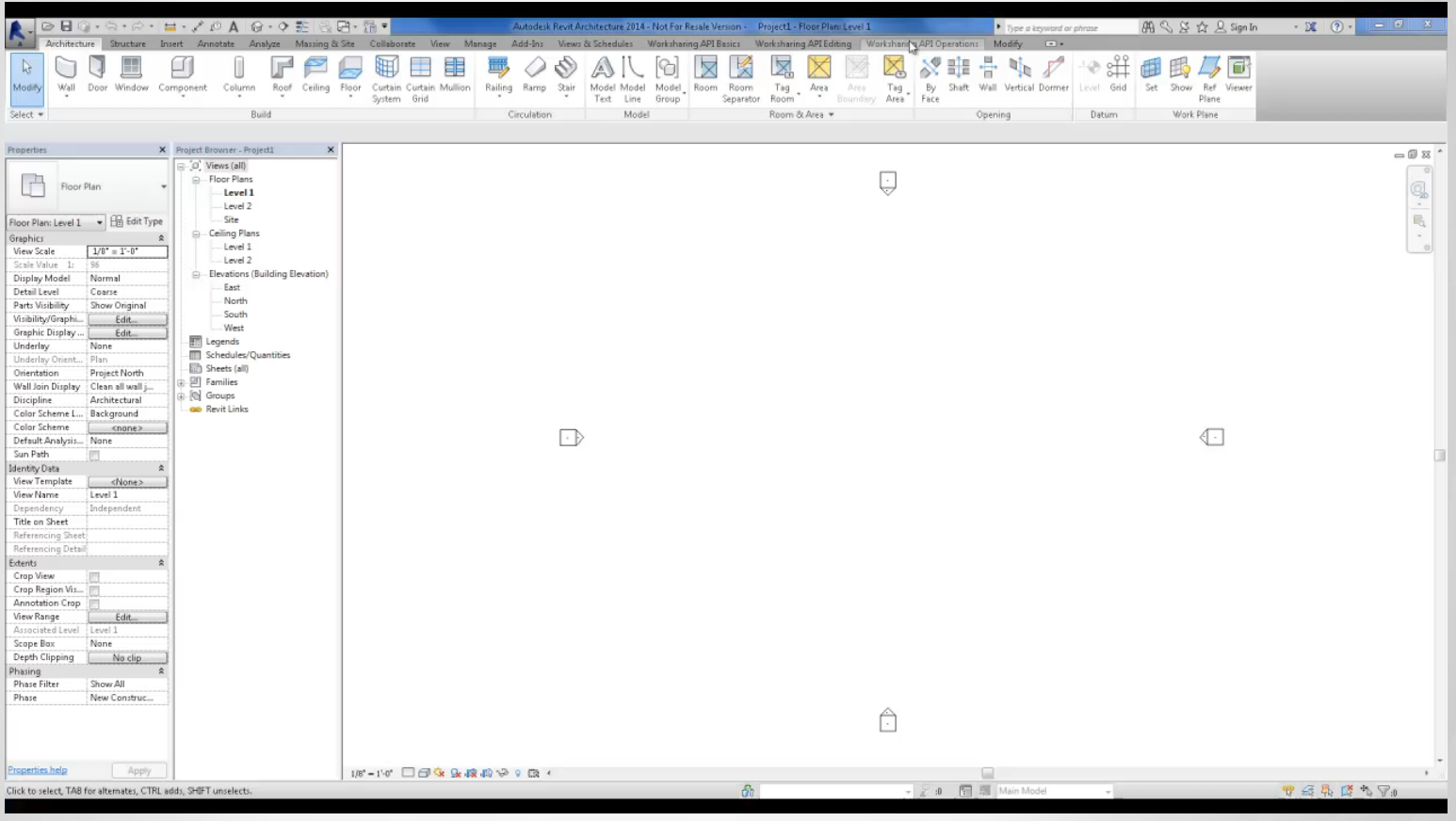
Before you open projects

- Create a new local file
  - WorksharingUtils.CreateNewLocal()
    - Input: ModelPath (central model on disk or server)
    - Input: ModelPath (file path to the new local)
  - Local is assigned to the user associated with session
  - Note exceptions related to inability to find or access the central model, attempting to use a non-workshared central model, or attempting to overwrite existing files.

Open projects

- Application.OpenDocumentFile()
- UIApplication.OpenAndActivateDocument()
  - Open options related to worksharing
    - DetachFromCentralOption – use when your add-in will operate on a workshared file but doesn't need to make permanent changes (e.g. for export/print or data extraction)
      - *DetachAndDiscardWorksets broken in 2014 FCS, fixed in UR1*
    - AllowOpeningLocalByWrongUser
    - OpenWorksetsConfiguration – configure the worksets to be opened (per information read by WorksharingUtils.GetUserWorksetsInfo())
  - Note new exceptions related to central model access problems

Open projects examples – detached, last viewed, open worksets



Save projects

- Document.Save()
- Document.SaveAs()
  - WorksharingSaveAsOptions
    - OpenWorksetsDefault – all, all editable, last viewed, ask user
    - SaveAsCentral
    - ClearTransmitted
      - Workshared transmitted models are always detached from central
  - Note new exceptions related to central model

Links

- RevitLinkType.Create()
  - RevitLinkOptions includes WorksetConfiguration
  - Specify worksets to be open or closed within the link

Events

- Synchronize with central
  - Application.DocumentSynchronizingWithCentral
  - Application.DocumentSynchronizedWithCentral
- Document events (open, close, save)
  - Sometimes these will trigger as a result of worksharing operations, e.g. Save during a Synchronize

Revit Server considerations

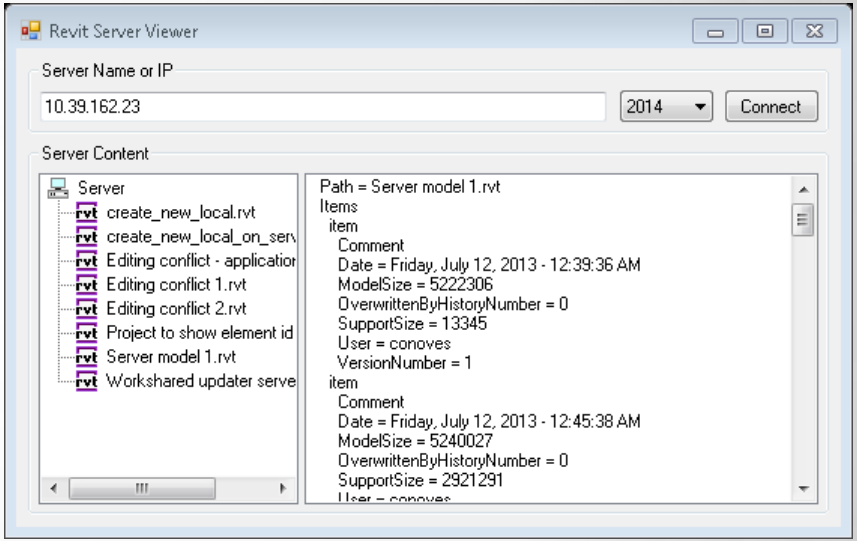
- Add-ins should consider
  - Model paths vs. server paths for document operations
  - Performance and latency related to
    - Synch with central
    - Reload latest
  - CopyModel() – can copy from server to local file for detached operations (export, data extraction)
  - Revit Server REST API – to get information on managed files

Revit Server REST API

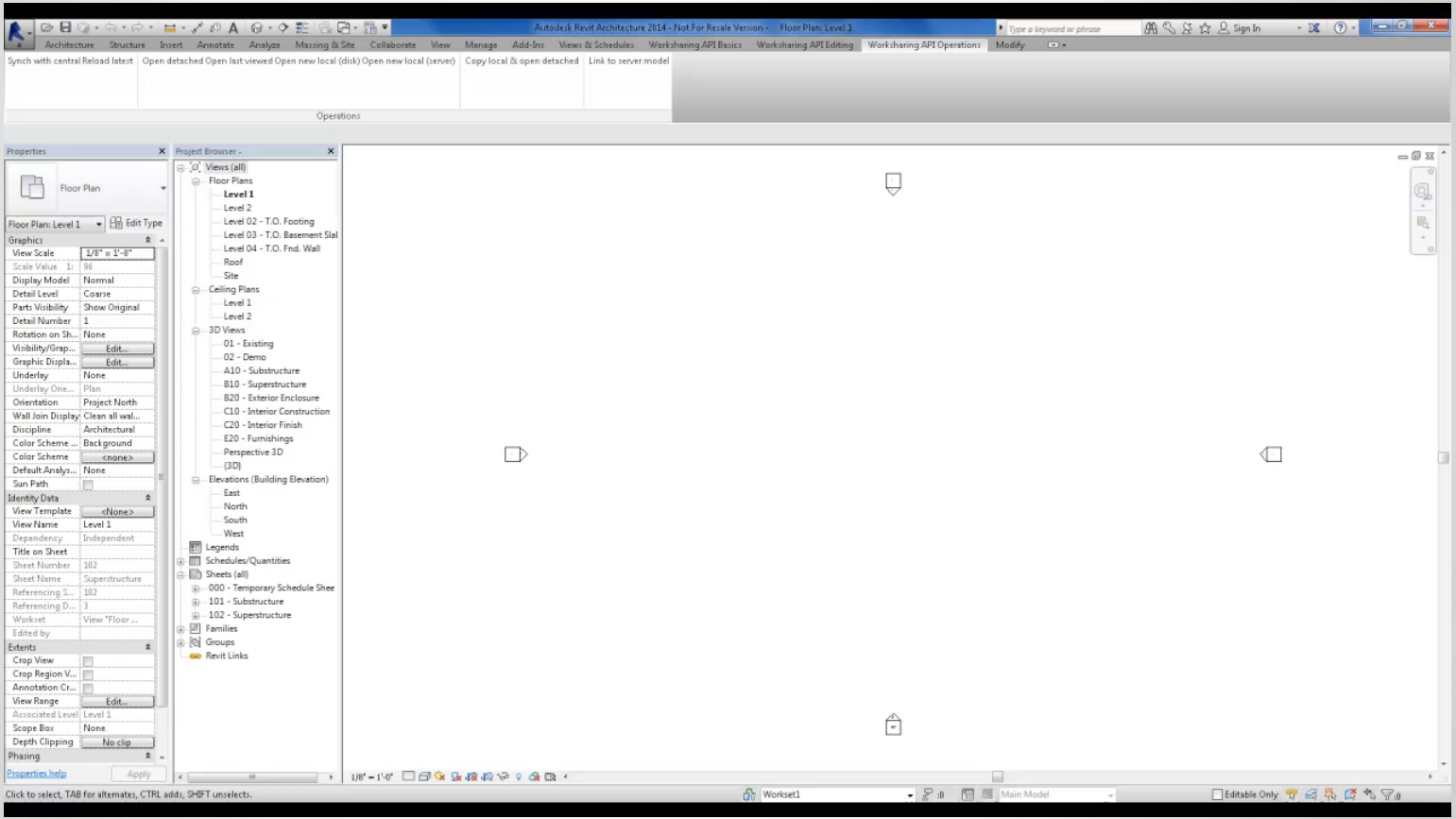
- Query
  - Server properties
  - Contents
  - Folder info
  - Model history
  - File info
  - Project info
  - Thumbnail
- Server management
  - Lock/unlock
    - Server
    - Folder
    - Model
  - Copy/move models on the server
  - Create folders
  - Rename models or folders
  - Delete folders or models

Revit Server REST API

- Use standard HTTP request/response protocol
- Responses are encoded in JSON
- Folder separator difference
  - In HTTP requests, “|”
  - In Revit API operations, “/”



Revit server document example



Course summary

- Learned the basics of Revit worksharing techniques and terminology
- Discovered the basic classes and methods related to worksharing in the API
- Enumerated suitable techniques to make a Revit add-in to work well in a workshared environment
- Identified the techniques related to management of projects, files and servers

Reminder

- Consider attending DV3464-R “Making Autodesk® Revit® Add-ins That Cooperate with Worksharing: A Roundtable Session” at 1:00 pm today for more discussion