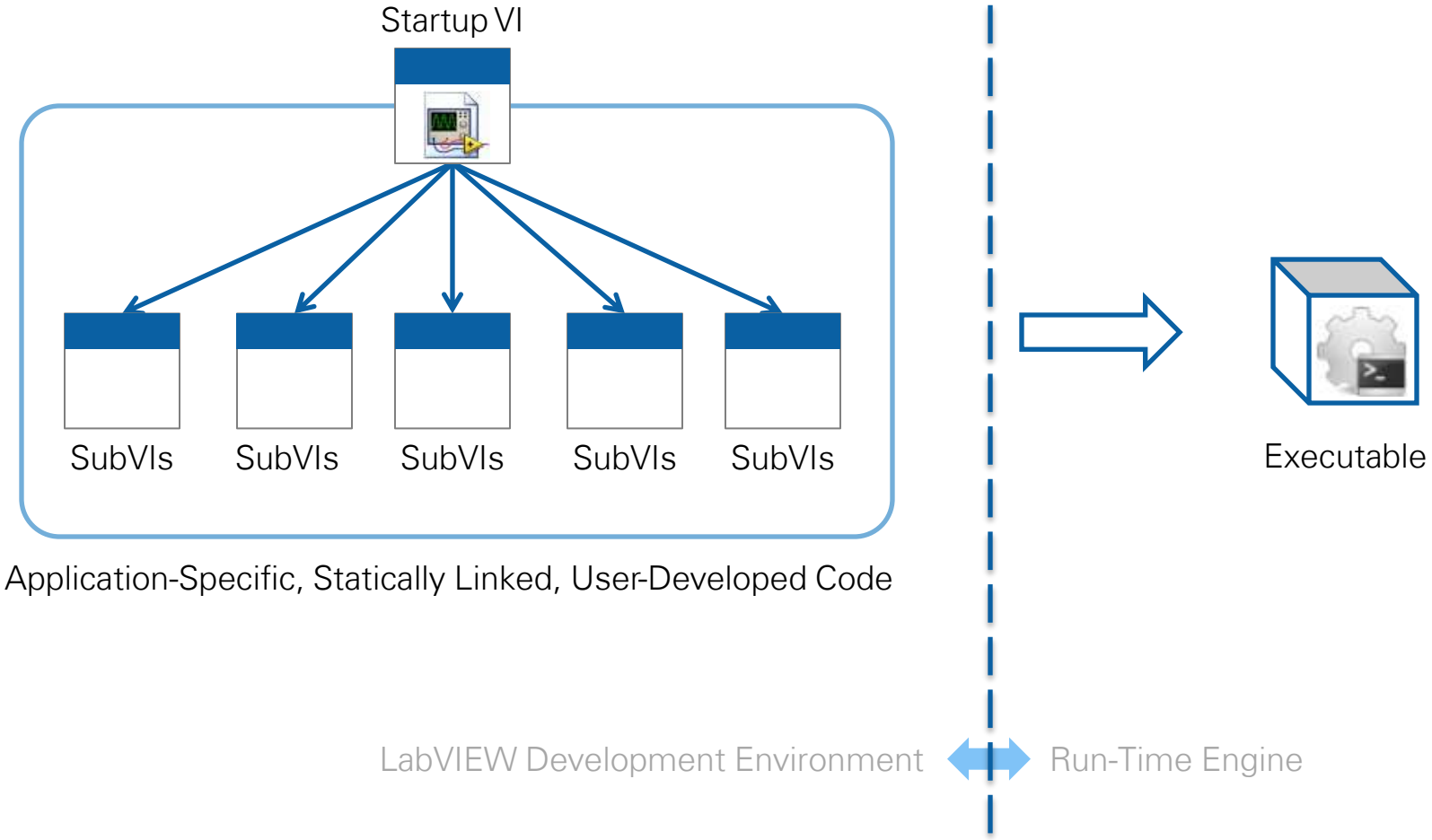
LabVIEW Application Builder:

Tips and Tricks for Deploying Desktop Applications

Agenda

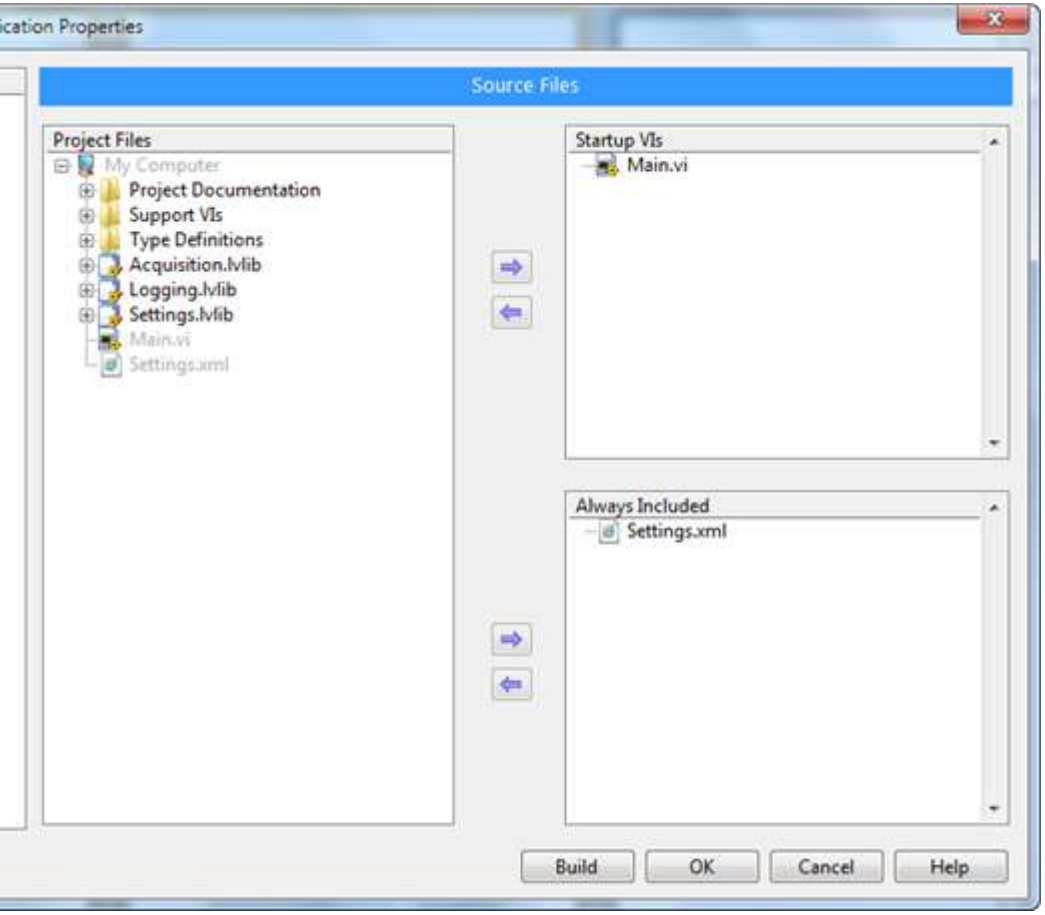
1. Review of LabVIEW executables
   - Configuring and building an EXE
   - VI dependencies – load and build considerations
2. Designing for modularity and upgradeability
   - Source Distribution plug-ins (demo)
   - Packed Project Library plug-ins (demo)

Goal: Package Compiled Code for Distribution



NATIONAL INSTRUMENTS

Basic Configuration: Source Files Category



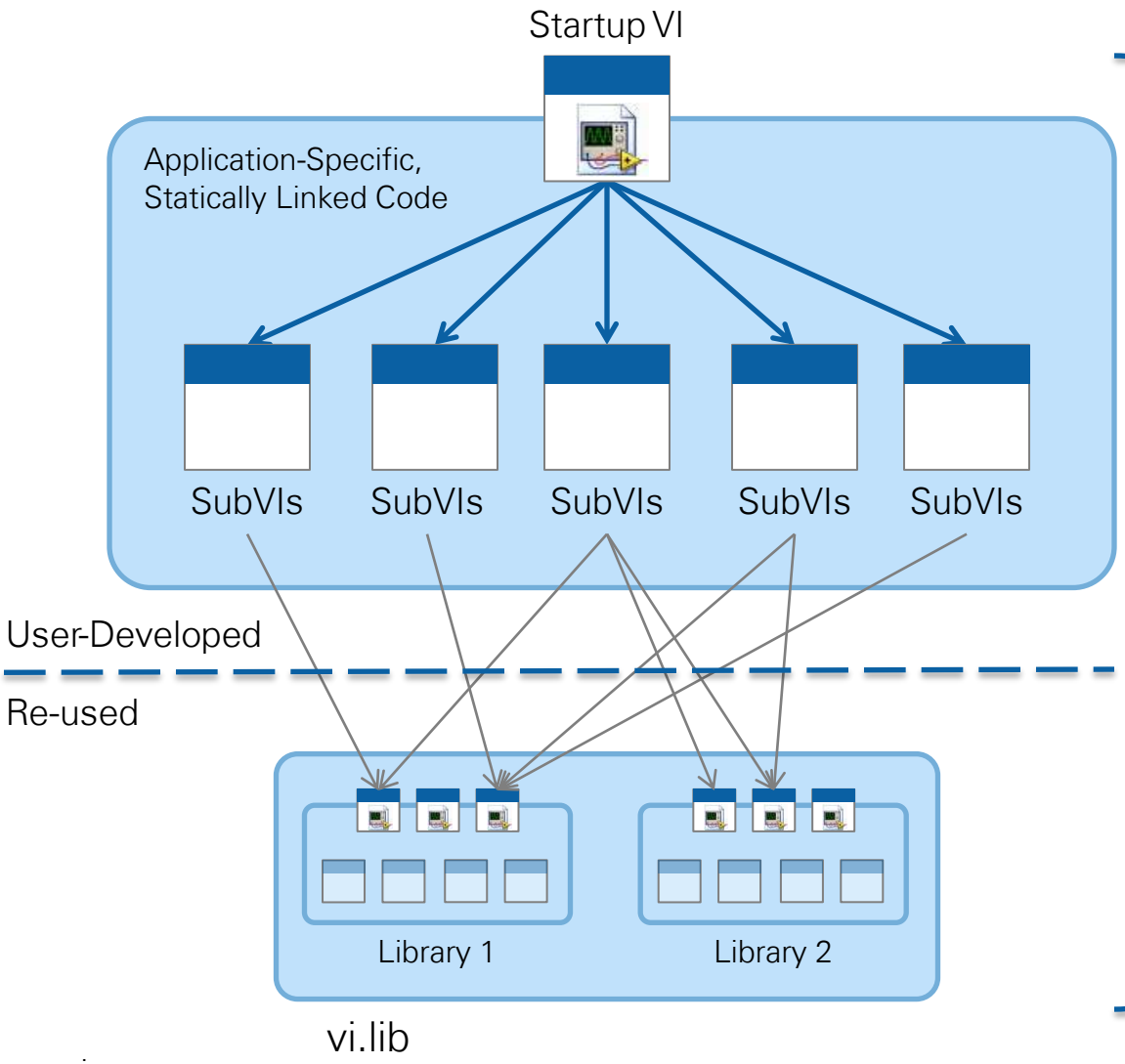
These VIs will be run when the executable is launched

These components will be built in to, or next to the executable

This is the only dialog that absolutely needs to be defined by a user in order to build a very basic LabVIEW system

NATIONAL INSTRUMENTS

What App Builder Does When You Build



1. Create new application instance for build
2. Open top-level VI
   (and any always-included VIs)
3. Identify statically linked dependencies
4. Remove unused VIs
5. Re-compile in new application instance
6. Save files to new location
7. Build into executable

Executable

NATIONAL INSTRUMENTS

EXE and the Run-Time Engine

Executable

Loader

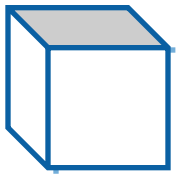
loads VIs and talks to run-time engine

VIs

+ compiled code
+ VI data
± diagram
± panels

Resources

+ icon



Run-time Engine

Core LV RTE

Core LabVIEW libraries and resource files required for VIs to run

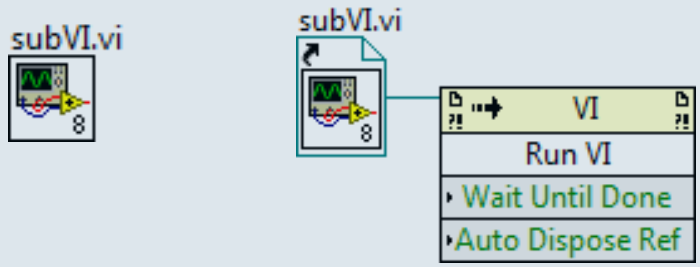
MSVC Run-time Files

Libraries required for C code to run

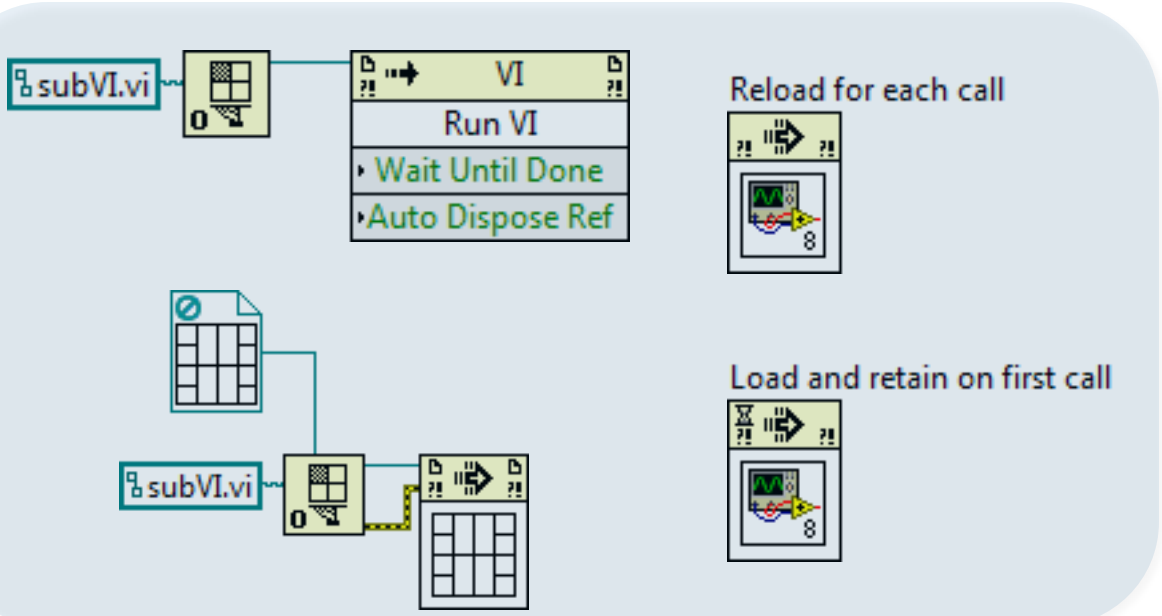
Additional Components

± Datasocket
± Variable Engine
± NI-USI

When Are Dependencies Loaded?



Loads when caller **loads**
- "Static" loading
- Load cost incurred when EXE loads even if code is not executed

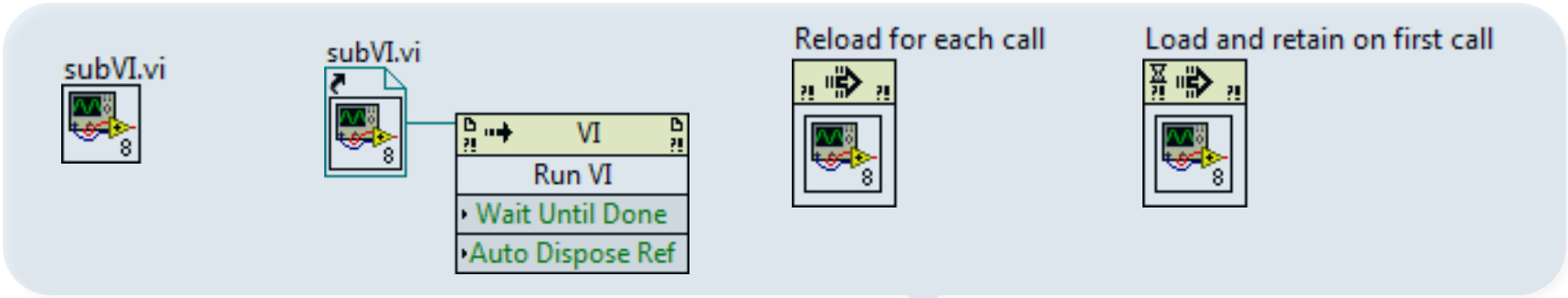


Loads when caller **runs**
- "Dynamic" loading
- Load cost incurred when EXE runs, but only if this code is executed

NATIONAL INSTRUMENTS

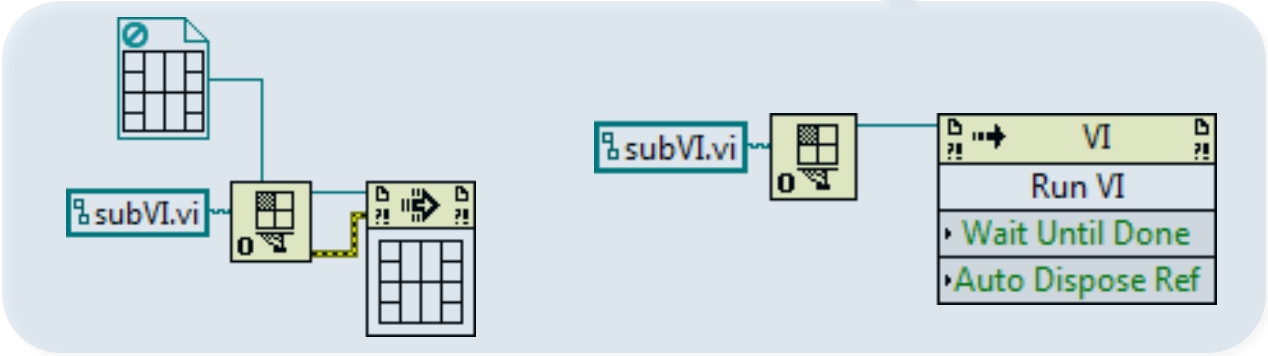
How Are Dependencies Included in EXE?

Added to EXE automatically



Executable

Not added to EXE automatically



NATIONAL INSTRUMENTS

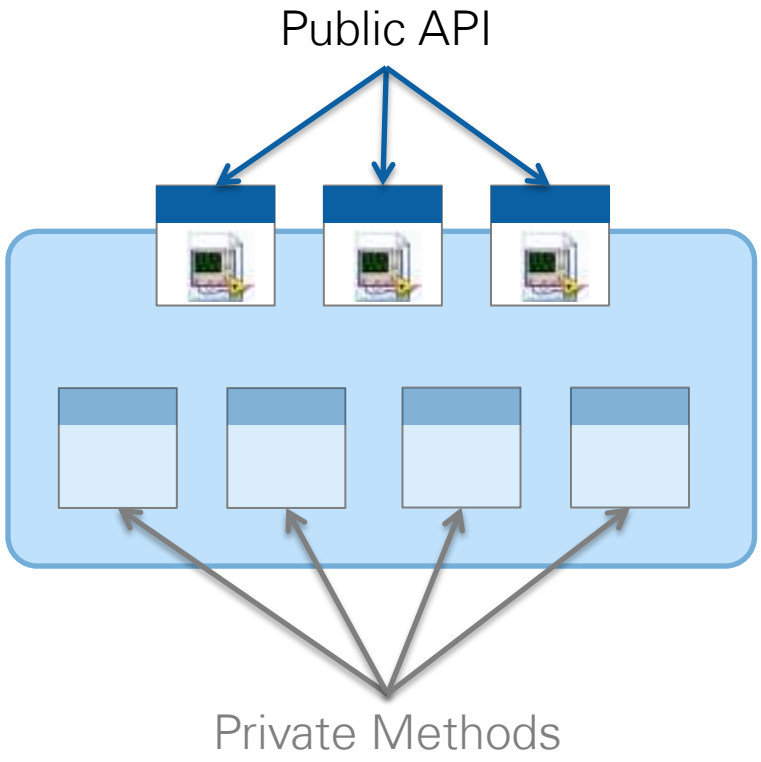
"Field Upgradable"

Add to or change an application's functionality after it has been deployed

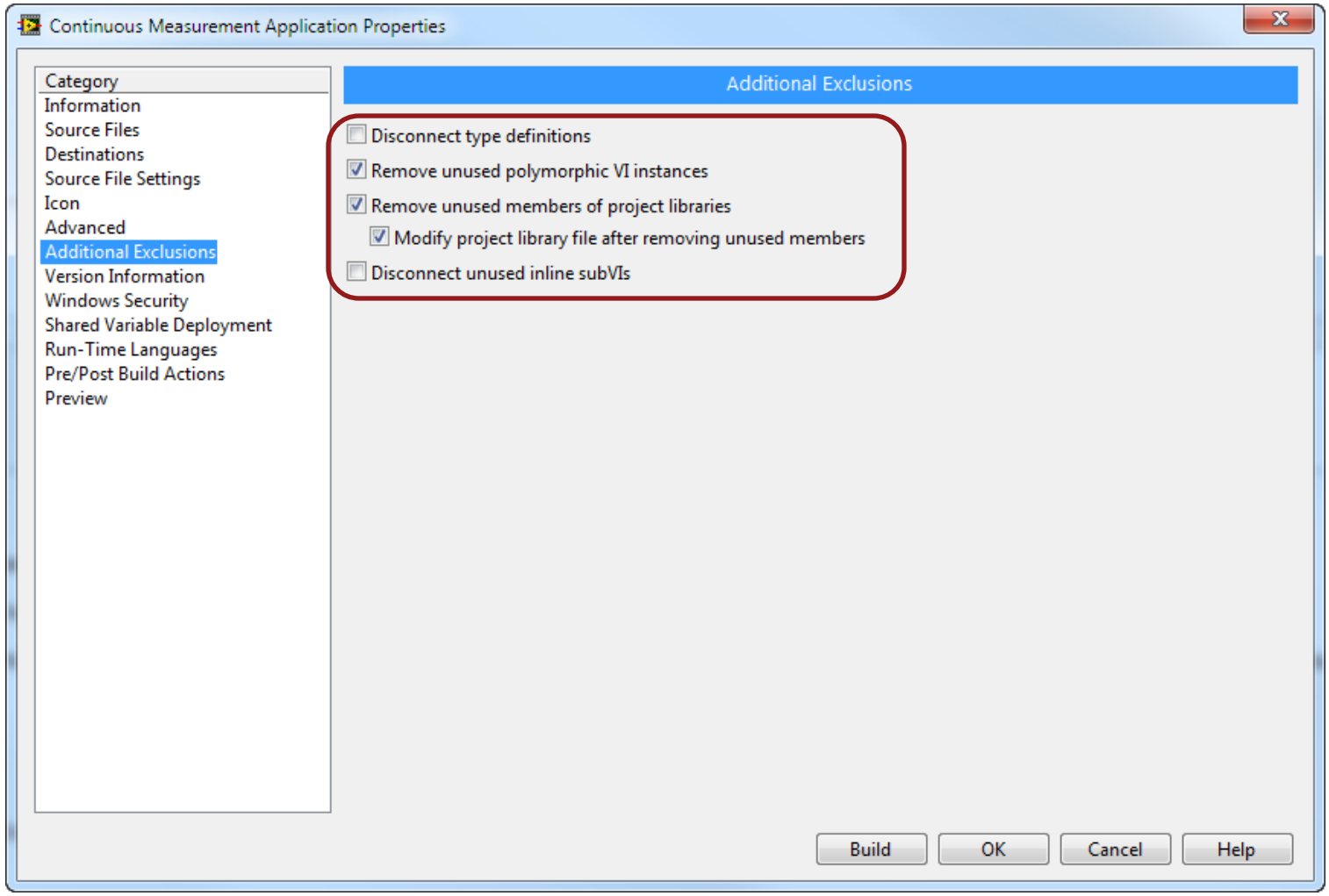
- Could be an update to existing code
  - Updating statically linked code
  - Localized bug fix
  - Swap one implementation for another
- Could be new functionality via a plugin
  - Dynamically loaded
  - Additional features, HW, etc.

Organizing Code into Project Libraries

- Identify atomic modules of code that are de-coupled
- As a general guideline, atomic modules of code should be organized into libraries
- Libraries should have a public API, which are the only VIs that can be called externally
- This is the first step

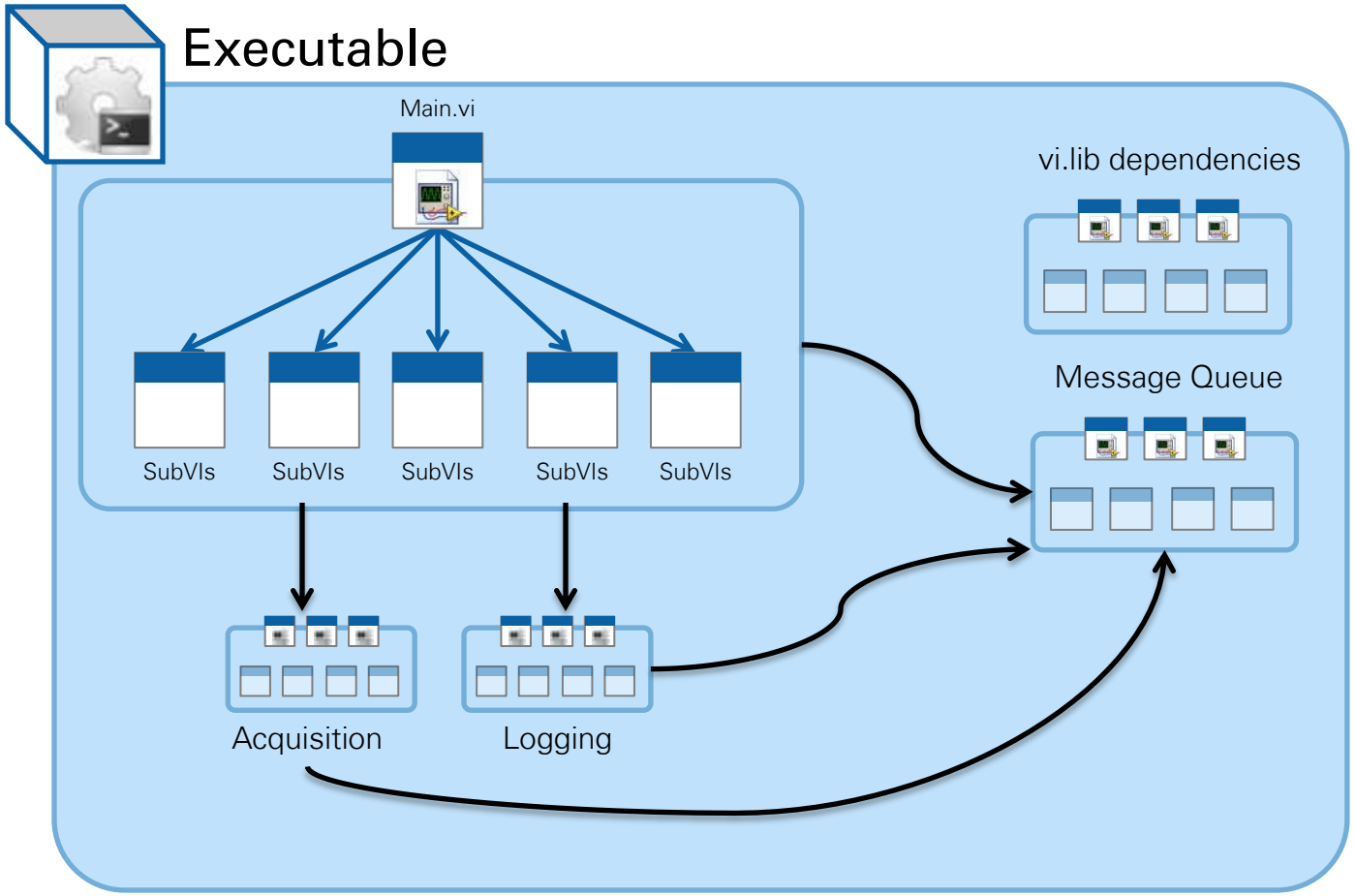


Reducing Footprint of Large Libraries (like AAL)



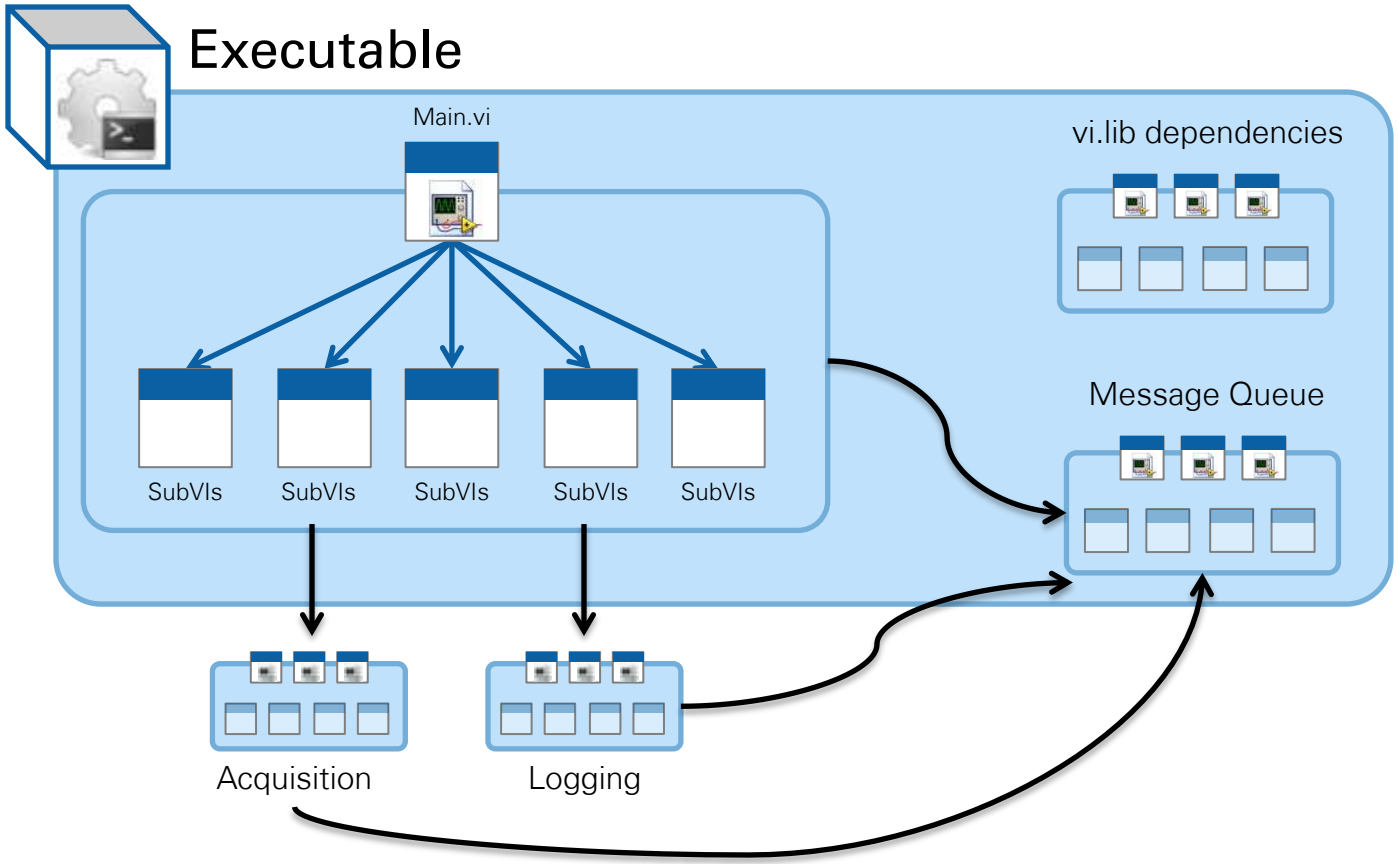
NATIONAL INSTRUMENTS

Single EXE



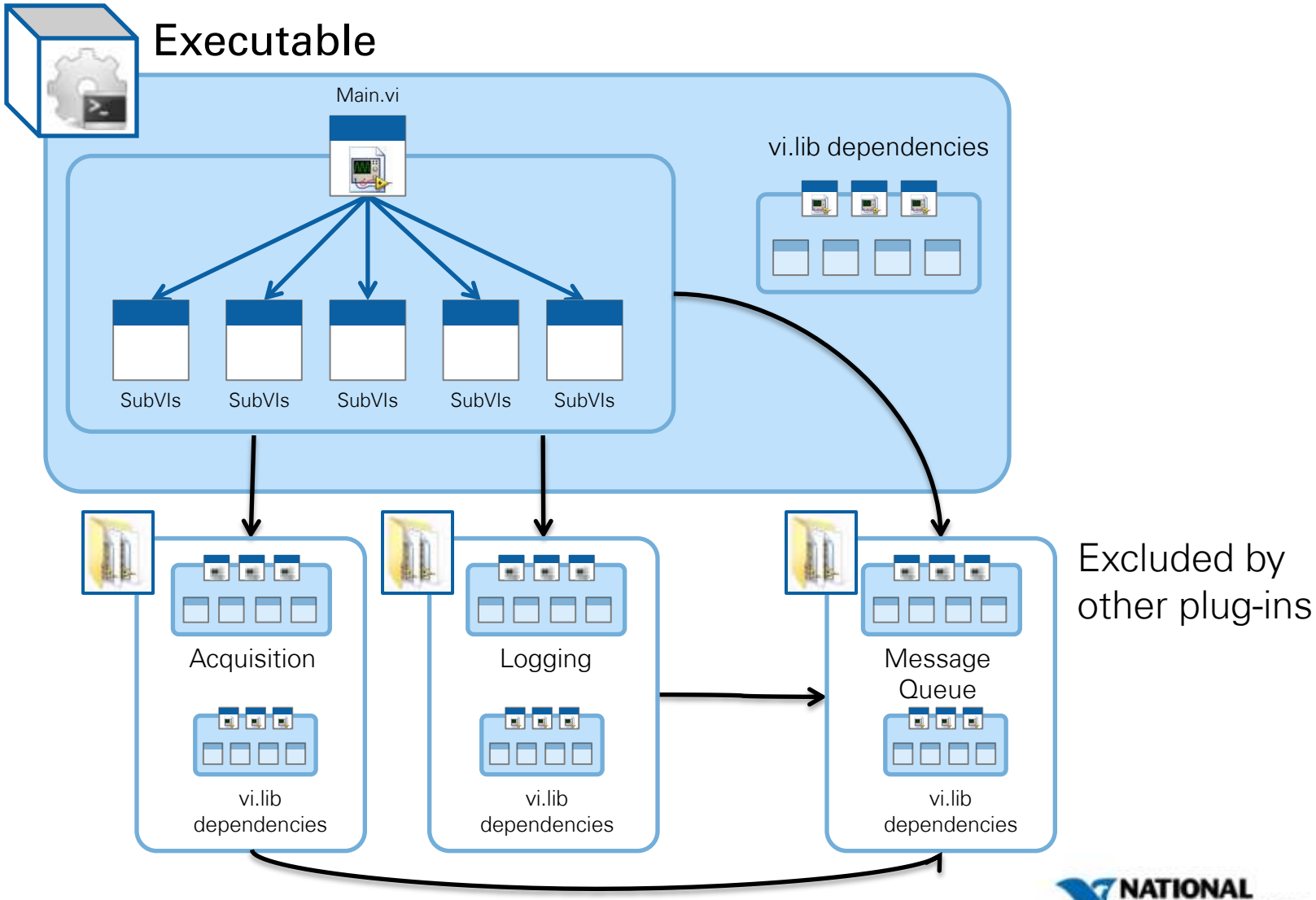
"External" libraries statically linked, automatically included in EXE

EXE + External VIs



External libraries statically linked, kept external to EXE via build settings

Separate Source Distributions



Separate Source Distributions

- Create an EXE with core functionality
- Distribute source distributions for plugins

Process:

1. Create a new project for lowest level plugin library
2. Place library in project
3. Add other plugins libraries from built source distributions
4. Build source distribution from project
    - include vi.lib VIs, exclude other plugins
5. Repeat 1-4 for all libraries, working up dependency chain
6. Build EXE with each library in it's own folder

Gotchas:

- If similarly named VIs may be revved independently then add a prefix or library associate
- At each stage resolve all conflicts to point to the exported source distribution before building

Separate Source Distributions (continued)

- Create an EXE with core functionality
- Distribute source distributions for plugins

To upgrade:

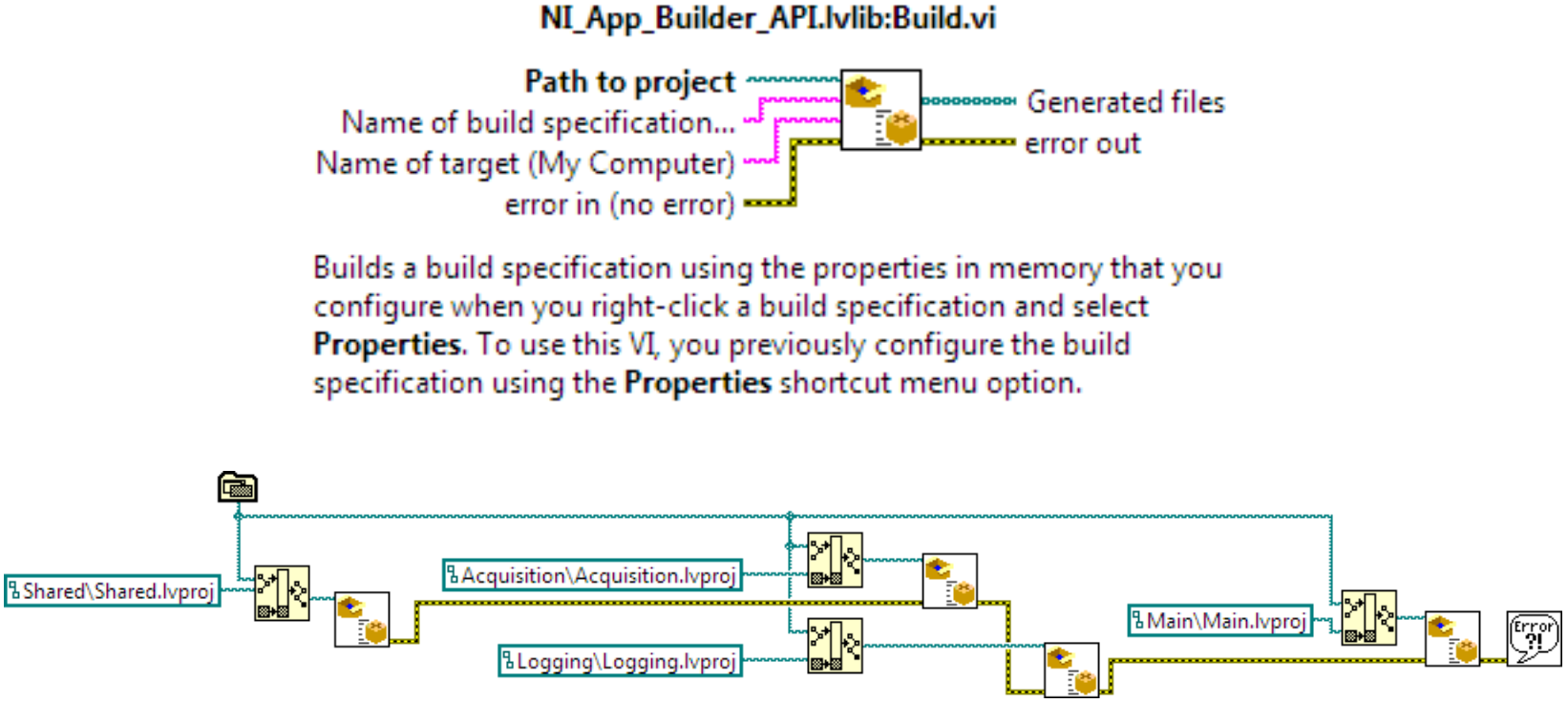
1. Modify VIs in the appropriate project
2. Rebuild source distribution
3. Replace folder in EXEs build location

Demonstration

Separate Source Distributions

Automating Builds

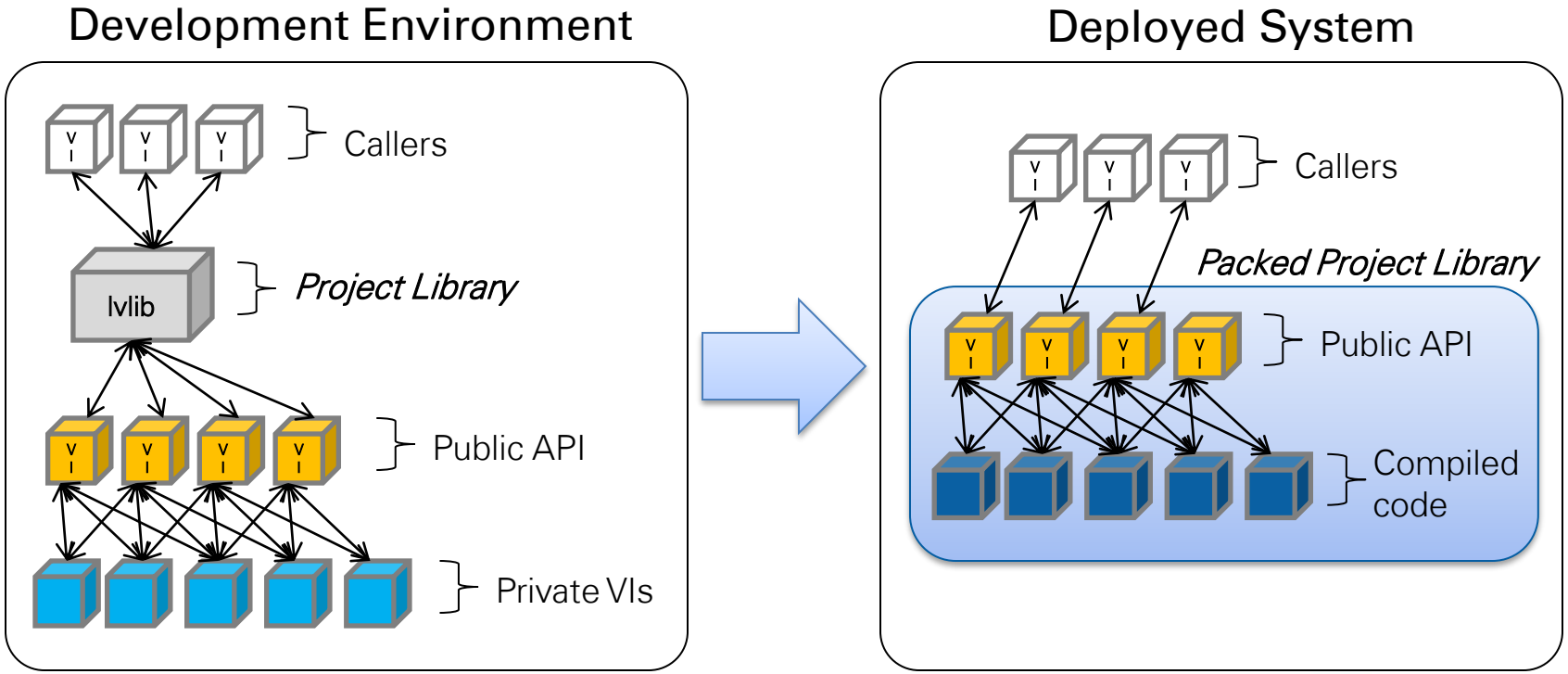
Building multiple projects in the correct order doesn't need to be tedious – this is LabVIEW, you can automate it



NATIONAL INSTRUMENTS

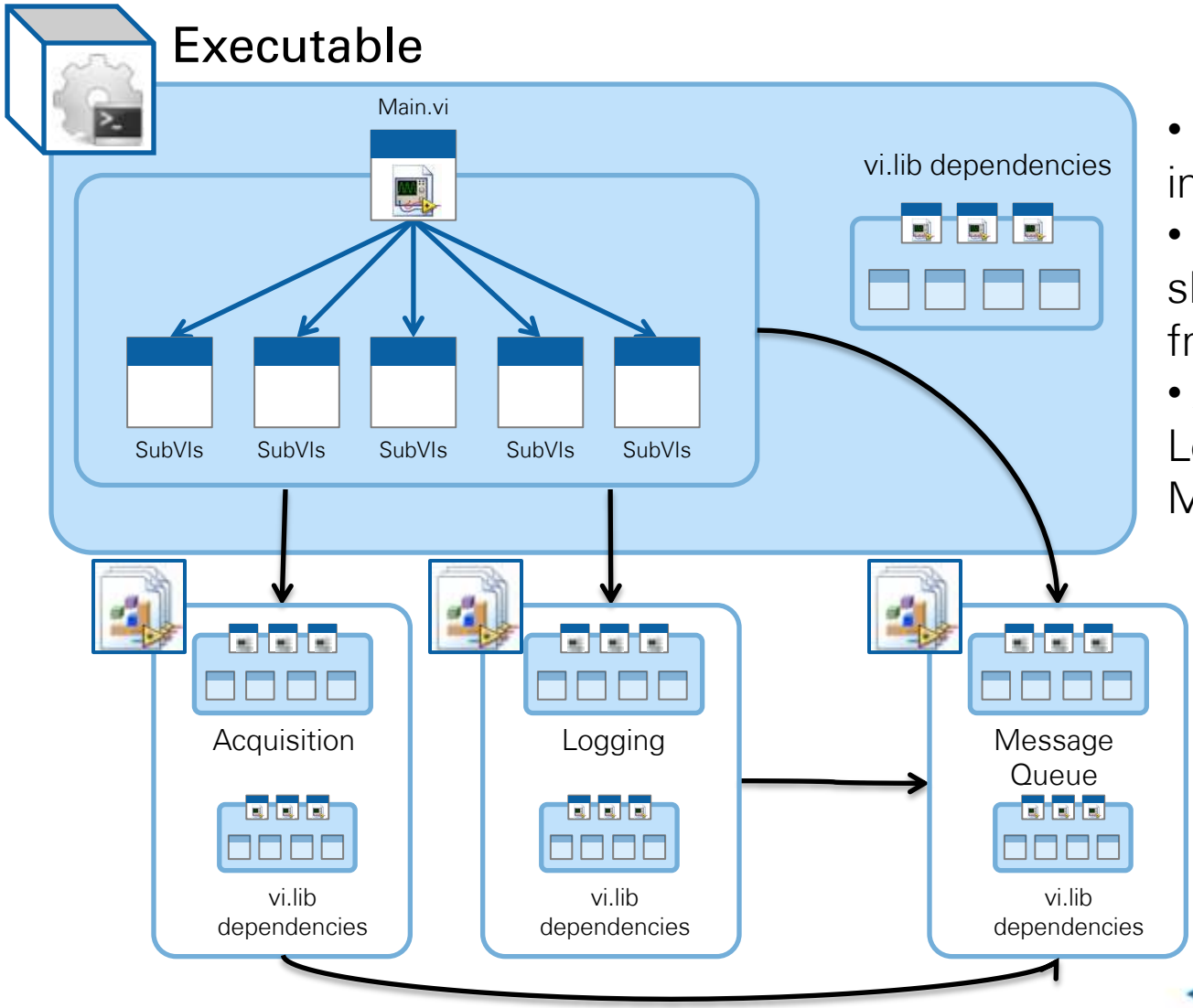
Packed Project Libraries

- Distribute and deploy LabVIEW libraries as a single file
- Think of them like LabVIEW-native DLLs
- PPL still a young feature (especially w.r.t. classes)



NATIONAL INSTRUMENTS

Separate Packed Project Libraries



- Build order important
- Updating slightly different from SDIST case
- Acq and Logging include Message lvlibp

NATIONAL INSTRUMENTS

Separate Packed Project Libraries

- Create an EXE with core functionality
- Distribute packed project libraries for plugins

Process:

1. Create a new project for plugin library
2. Place library in project
3. Add other PPL dependencies as necessary
4. Build PPL from project
5. Repeat 1-4 for all libraries, starting with lowest level
6. Add PPLs to main project, resolve all dependencies
7. Build EXE

Gotchas:

- At each stage resolve all conflicts to point to the exported PPL before building
- Only need keep one copy of each PPL (common dependencies may get duplicated)
- Some issues with PPL and class name resolving

Separate Packed Project Libraries (continued)

- Create an EXE with core functionality
- Distribute packed project libraries for plugins

To upgrade:

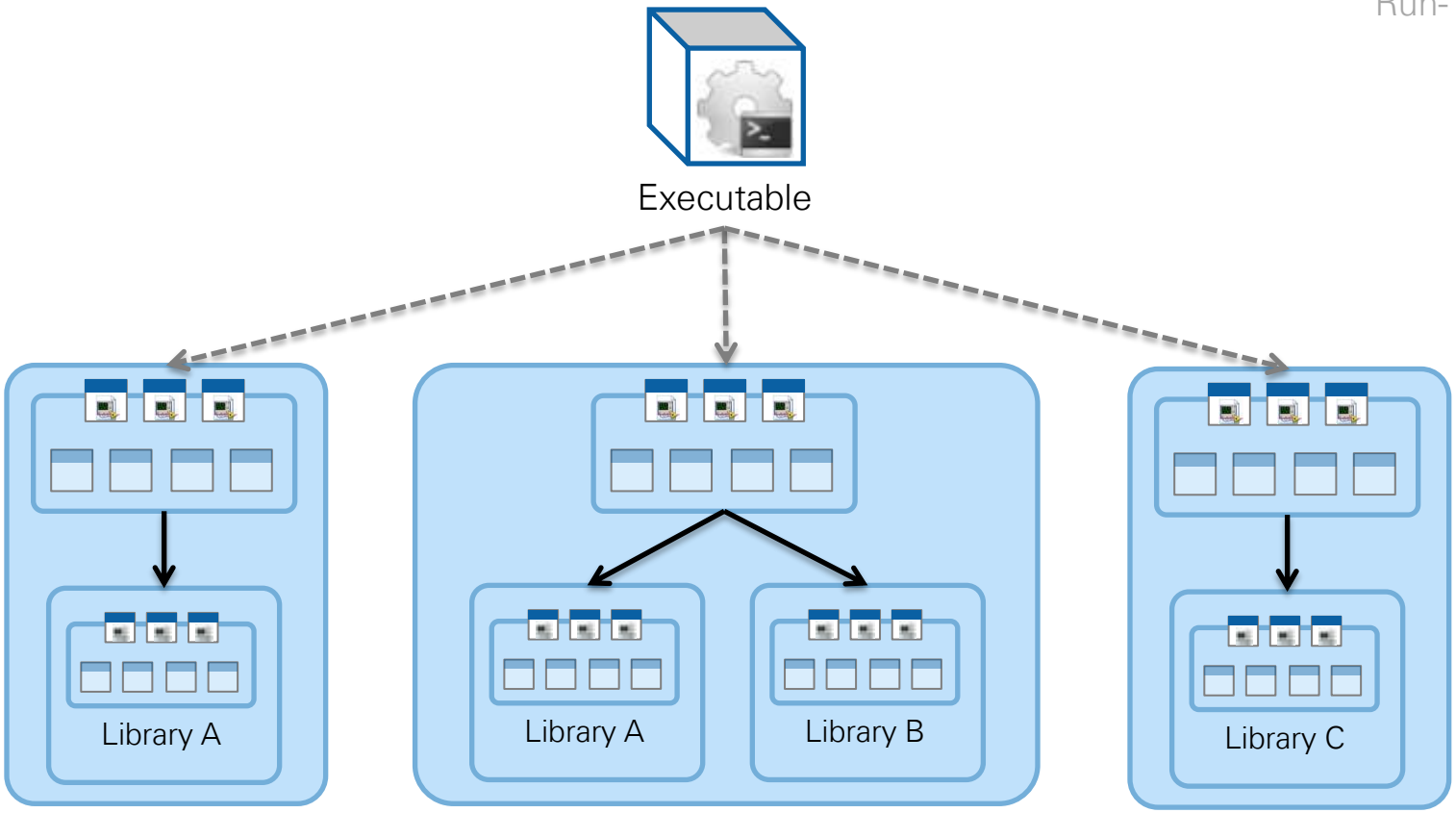
1. Modify VIs in the appropriate project
2. Rebuild PPL
3. Replace PPL in EXEs build location

Demonstration

Separate Packed Project Libraries

Managing Duplicate Dependencies

Run-Time Engine



NATIONAL INSTRUMENTS

Summary

- App Builder is a powerful tool
- Different build strategies optimized for different scenarios
  - EXEs
  - Deployed VIs
  - PPLs
- PPLs are an evolving technology

- http://www.ni.com/labview/skills-guide/en/