Qt Developer Days 2013

Wrapping C++ Objects For Property Exposure In QML

Charley Bay

Beckman Coulter, Inc.

Review: C++ From Within QML

Integrating QML and C++:

(1) **Call (Invoke) C++** from QML

- Invoke *(existing / internal)* C++ API for Qt QML or Qt Quick
- Apply `Q_INVOKABLE` to funcs in `QObject`-derived

(2) **Implement C++ Types** for QML

- Derive from `QObject`
- Define “properties” (`Q_PROPERTY()`)
- Register with QML

`qmlRegisterType()`	`qmlRegisterUncreatableType()`
`qmlRegisterInterface()`	`qmlRegisterSingletonType()`

What's The Problem?

The Issue: "Deep / Rich" Types

Are typically:

- **Domain-specific**
- Have **business logic**
- Often **found in legacy systems**
- **Expected to be "reusable"** within that domain
- Are **often "*key abstractions*"** *(but not always)*
- May be **"lightweight" or "heavyweight"**

Deep / Rich Types: Ligthweight

Lightweight: *"Popcorn Types"*

- **Frequently come-and-go** *(are created-and-destroyed)*
- Have **non-trivial business logic**
- Often **imply rules needed throughout** the system *(e.g., point-of-entry data validation, back-end system adaptation across components/revisions of hardware, etc.)*
- Often used to **bridge sub-systems**
- Often **"serialized" to span sessions, setups, configurations**
- Often **comprise significant portions of the domain-specific APIs** for "interface-state" *(e.g., are "ubiquitous")*

These are typically your desired QML "properties"!

Deep / Rich Types: Lightweight Examples

Example "Lightweight-Deep/Rich" types

- `class Wavelength`
- `class Voltage`
- `class Parameter`
- `class Detector`
- `class SpectraFilter`

Deep / Rich Types: Heavyweight

Heavyweight: *Typically "Key Abstractions"*

- Often **created at "system-start"** with known configuration *(e.g., "devices")*
- Typically **expressed through class hierarchies** *(e.g., across device revisions, devices specific to a product line evolution)*
- Typically **consistent across products / product-lines** *(e.g., "one software to control all devices in a family-of-product-lines)*
- **May be "plug-&-play"** *(e.g., subsystems may come-and-go as units during system-run)*
- Is often **what needs to be monitored / controlled**
- Is **part of higher-level system** *(coordination with other key abstractions is typical for data validation, denial of access/operations [e.g., "baton-passing"], etc.)*

You typically want to "hand-wrap" these for QML!

Deep / Rich Types: Heavyweight Examples

Example "Heavyweight-Deep/Rich" types

- `class FluidicsSubsystem`
- `class OpticsSubsystem`
- `class CytometerInstrument`
- `class ExperimentArchive`

`QObject` Implies Alternative Control Flow

`QObject` and `QMetaObject`:

- `signals/slots` introduce coupling to imply **"alternative-control-flow"** *(only relevant to designs intended for that control flow)*
- `QObject`-derived types should typically be **used as "identity" instances** *(not with value semantics, copy `CTOR` and `operator=()` not available)*

`Signals/slots` are "side-effect" triggers used to assemble, communicate among, and "stitch-together" across sub-systems. *While they can be used within a sub-system, they are often* ***not needed where deterministic behavior is required*** *within (even large) key-abstractions that represent functional business-logic state.*

Wrapping Deep / Rich Types: `QObject`

C++ types **sometimes cannot derive from `QObject`:**

- If **cross-platform embedded** *(to "very-small-footprint-deployment")*
- If **application-specific performance requirements**
- If application-specific **design may establish class hierarchies** *(which disallow* `QObject`*)*
- If **used where Qt is not** used
- If **"existing/legacy"** code
- If **3rd Party code** that cannot be modified
- If developed / tested at unit / functional level with **as little coupling as possible**

An Example: `class Wavelength`

- **Lightweight** class *(only data member is "`double`")*
- Has **domain-rules**
  - Range-bounds
  - Label formatting
  - Interpreted as "color"
- Is ***ubiquitous*** throughout **domain-specific APIs**

```
class Wavelength
{
  double value_nm_; //IS ONLY DATA MEMBER!
public:
  Wavelength(double value_nm);
  Wavelength& operator=(double value_nm);
  Wavelength& operator=(Wavelength& other);

  QString getAsLabel(void) const;

  bool isBlue(void) const;
  bool isGreen(void) const;
  bool isOrange(void) const;
  bool isRed(void) const;
  bool isViolet(void) const;
  bool isYellow(void) const;

  bool isVisible(void) const;
};
```

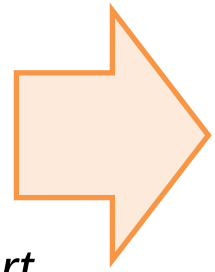
Qt Developer Days 2013

As It Really Exists: `class Wavelength`

Wrapped In QObject: "SdqWavelengthBox"

- We only care about "**properties**"!
- **Some "get / set"** utility functions *(to support properties)*
- ***Plus** (optional!):*

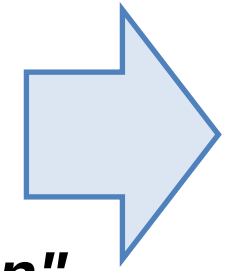
  Utility "enums" and functions to support QAbstractItemModel *(for a **"model / collection"-of-Wavelength" instances**)*



```
class SdqWavelengthBox : public QObject
{
   Q_OBJECT
   Q_PROPERTY(const bool hasState READ hasState NOTIFY wavelengthChanged)
   Q_PROPERTY(const qreal wavelengthNm READ wavelengthNm WRITE
setWavelengthNm NOTIFY wavelengthChanged)
   Q_PROPERTY(const QString asText READ asText NOTIFY wavelengthChanged)
   Q_PROPERTY(const QColor color READ color NOTIFY wavelengthChanged)

 private:
   Wavelength my_wavelength_;  // WRAPPED DATA MEMBER!

 signals:
   void wavelengthChanged(void) const;
 public:
   // ...CTORs, DTOR…
   const bool hasState(void) const;
   const qreal wavelengthNm(void) const;
   void setWavelengthNm(qreal value_new);
   const QString asText(void) const;
   const QColor color(void) const;
```



```
   public:
      enum EnumRole {
        ENUM_ROLE_FIRST      ,
          ROLE_AS_TEXT       = ENUM_ROLE_FIRST,
          ROLE_COLOR         ,
          ROLE_HAS_STATE     ,
          ROLE_WAVELENGTH_NM ,
        ENUM_ROLE_LAST       = ROLE_WAVELENGTH_NM,
      };
      enum {
        NUM_ENUM_ROLES = ENUM_ROLE_LAST - ENUM_ROLE_FIRST + 1,
      };

      static QVariant GetQVariantForEnumRoleEnum(
        const SdqWavelengthBox& object_box_value,
        SdqWavelengthBox::EnumRole enum_role);

      static const char** GetStrArrayEnumRoleNames(void);
};
```

```
//-----------------
// RECALL: To convert our type easily "to/from" "QVariant",
// we must EXPLICITLY declare the meta-type.
//
Q_DECLARE_METATYPE(SdqWavelengthBox)
//-----------------
#endif // SdqWavelengthBox_hpp
```

Registering Our "SdqWavelengthBox"

In application, or QML-plugin derived from `QQmlExtensionPlugin`, **register the type** into the QML type-system.

This **"translates"** the "type-wrapper" of "`SdqWavelengthBox`" so that **QML only sees the type as "`Wavelength`"** *(which is ALWAYS what you want!)*

```
#include <QtQml/QQmlExtensionPlugin>
#include <QtQml/qqml.h>
// ...
#include "SdqWavelengthBox.hpp"

class CytoQmlPlugin : public QQmlExtensionPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "com.bec.cyto")

  public:
    void registerTypes(const char *uri)
    {
      qmlRegisterType<SdqWavelengthBox>(
        uri,              //const char * uri,
        1,                //int versionMajor,
        0,                //int versionMinor,
        "Wavelength"); //const char * qmlName)
    }
};
```

Using Our "SdqWavelengthBox"

In QML file:

- **Import your plugin** (that registered/exported "SdqWavelengthBox" to the QML type system)
- **Use it** (with the name "**Wavelength**")

*(This example imports to the namespace "**Bec**", which is optional.)*

Launch HelloWavelength.qml

```
// FILE: HelloWavelength.qml
import QtQuick 2.1
import QtQuick.Controls 1.0
import QtQuick.Window 2.0

import com.bec.cyto 1.0 as Bec

ApplicationWindow {
  title: qsTr("TestHelloWavelength")
  width: 360
  height: 360
  Bec.Wavelength {
    id: myWavelength
    wavelengthNm: mySlider.value
  }
  Rectangle {
    id: myRect
    anchors.fill: parent
    color: myWavelength.color
    Label {
      id: myLabel
      anchors.centerIn: myRect
      text: myWavelength.asText
      font.pixelSize: 48
      font.italic: true
      color: "white"
    }
    Slider {
      id: mySlider
      anchors.top: myLabel.bottom
      width: myRect.width
      maximumValue: 800
      minimumValue: 300
      stepSize: 1.0
      value: myWavelength.wavelengthNm
    }
  }
}
```

REVIEW: What Just Happened?

What we did:

1. **Existing C++ `class`** *(did not derive from* `QObject`*, we did not "touch" it)*
2. **Defined "`QObject`-derived-wrapper"** `class`
3. **Exported "`QObject`-derived-wrapper"** *(to the QML type system)*
4. **Used type "natively"** within QML

…There are implications from "(1)"…

REVIEW: What Just Happened? (continued)

What we got:

1. **Made available** an existing C++ class **to QML (*without touching it!)***
    1. **Do not need to re-compile** existing (legacy) code
    2. **Do not need to re-validate / re-verify existing systems** *(BIG concern for regulated industries)*
2. **Abstracted to a "higher-level"** a "new-layer" of properties **for declarative-binding**.
    1. Interface for **"declarative-QML-`Wavelength`" is DIFFERENT** from "imperative-C++-`Wavelength`". *THIS IS A GOOD THING*.
    2. **New declarative-abstractions**, with internal "bridging / binding", **enables better interfaces and separation of subsystems** *(across modules, devices, and between GUI / UI and the "back-end")*

Some Issues With This Approach

- Not difficult work, but **is tedious** if must wrap many existing C++ `class`es *(many systems have dozens, or hundreds of these domain-specific types)*
- We achieved **"value semantics":**
  - **What if we want "reference" semantics** to reference a "`Wavelength`" instance in the "back-end" system? *(Could change our implementation to a "`smart_ptr<Wavelength>`"…)*
- Need for **updates:**
  - If many QML-`Wavelength` instances "reference" a "bound-property-instance" in the "back-end", **how to ensure all QML-`Wavelength` instances are "updated/notified"** when back-end instance changes?
- **Must manage the "two-interfaces"** for "QML-`Wavelength`" and "C++-`Wavelength`"

Proposed: Code Generator To Expose C++ to QML

1. **Existing C++ `class`** *(not derived from `QObject`)*
2. **Define "interface-file"** *(for each C++ `class`)*
3. **Run Code Generator** *(creates C++ "wrapper-class(es)")*
4. **Build, Link, Run**
5. ***Profit!***

Due to additional features provide through the code-generator, ***we now refer to this as "Boxing"*** *the C++ class – not "wrapping".*

This relates to a tongue-in-cheek reference to the C++/CLI topic of "boxing/unboxing" in the .NET Common Language Runtime (CLR) that (implicitly) "boxes" a value-type to the type-object, where later (explicit) "unboxing" extracts the value-type from the object.

Case Study: An Example Code Generator

1. **Assume existing** C++ "`class Wavelength`"
2. **Define a "declarative-property-interface"** in a new file, "`Wavelength.sdgen_sdqbox`"

```
// FILE: Wavelength.sdgen_sdqbox

bool hasState {
  token            : HAS_STATE
  cppfuncget       : hasState
  qmlnotify        : wavelengthChanged
}
qreal wavelengthNm {
  token            : WAVELENGTH_NM
  cppfuncget       : getWavelengthInNanometers
  qmlwrite         : setWavelengthNm
  cppfuncset       : setFromWavelengthInNanometers
  cppfunchasvalue  : hasWavelength
  qmlnotify        : wavelengthChanged
}
QString asText {
  token            : AS_TEXT
  cppfuncget       : QtUtil::GetQStringFromSdString(getItemToBox()->getAsStringUserInterface())
  qmlnotify        : wavelengthChanged
  headersextra     : QtUtil SdString
}
QColor color {
  token            : COLOR
  cppfuncget       : GuiCytoUtil::GetQColorBackgroundForWavelength(*getItemToBox())
  qmlnotify        : wavelengthChanged
  headersextra     : GuiCytoUtil
}
```

Side Issue: Data Ontology

Names Matter!

- C++: `ReallyLongFunctionNamesAreFine`
- QML: ***short*** property names! (e.g., "`text`", "`color`")

Create an Ontology: *"An ontology provides a **shared vocabulary**, which can be used to model a domain, that is, the type of objects and/or concepts that exist, and their properties and relations."* ***--Ontology (information science)***
http://en.wikipedia.org/wiki/Ontology_(information_science)

- **Universal / ubiquitous:**
  `id`, `text`, `name`, `value`
- **Domain-specific:**
  `voltage`, `gain`, `filter`, `detector`
- Different domain **ontologies separated by namespaces:**
  `cyto`, `inst`

Case Study (continued): Organizing "Declarative Interface Files"

3. The **"file-name-root"** is ASSUMED to be **the same as the C++ `class`** that is to be "boxed".

4. **Place all "`*.sdgen_sdqbox`" files into a common directory** for the same "module" *(same shared-library or plugin)*

```
./MyWorkspace/.
              MyPlugin1/.
                        Detector.sdgen_sdqbox
                        SpectraFilter.sdgen_sdqbox
                        Wavelength.sdgen_sdqbox
```

5. From that directory, **run the "`sdgen_sdqbox.exe`"** utility *(code generator executable).* It **globs all "`*.sdgen_sdqbox`"** files and **generates (C++) source code** to that directory for each "boxed-class".

```
C:\MyWorkspace\MyPlugin1> sdgen_sdqbox.exe
...generating...
C:\MyWorkspace\MyPlugin1> dir /b

Detector.sdgen_sdqbox
SpectraFilter.sdgen_sdqbox
Wavelength.sdgen_sdqbox

SdqModelListWavelength.cpp   <==(generated!)
SdqModelListWavelength.hpp   <==(generated!)
SdqWavelengthBox.cpp         <==(generated!)
SdqWavelengthBox.hpp         <==(generated!)
SdqWavelengthBoxBack.cpp     <==(generated!)
SdqWavelengthBoxBack.hpp     <==(generated!)
SdqWavelengthBoxSet.cpp      <==(generated!)
SdqWavelengthBoxSet.hpp      <==(generated!)

...+generated for "SpectraFilter"...

...+generated for "Detector"...

C:\MyWorkspace\MyPlugin1>
```

Case Study (continued): Build & Link Declarative Box-Types

```
C:\MyWorkspace\MyPlugin1> dir /b

Detector.sdgen_sdqbox
SpectraFilter.sdgen_sdqbox
Wavelength.sdgen_sdqbox

SdqModelListDetector.cpp          <==(generated!)
SdqModelListDetector.hpp          <==(generated!)
SdqDetectorBox.cpp                <==(generated!)
SdqDetectorBox.hpp                <==(generated!)
SdqDetectorBoxBack.cpp            <==(generated!)
SdqDetectorBoxBack.hpp            <==(generated!)
SdqDetectorBoxSet.cpp             <==(generated!)
SdqDetectorBoxSet.hpp             <==(generated!)

SdqModelListSpectraFilter.cpp     <==(generated!)
SdqModelListSpectraFilter.hpp     <==(generated!)
SdqSpectraFilterBox.cpp           <==(generated!)
SdqSpectraFilterBox.hpp           <==(generated!)
SdqSpectraFilterBoxBack.cpp       <==(generated!)
SdqSpectraFilterBoxBack.hpp       <==(generated!)
SdqSpectraFilterBoxSet.cpp        <==(generated!)
SdqSpectraFilterBoxSet.hpp        <==(generated!)

SdqModelListWavelength.cpp        <==(generated!)
SdqModelListWavelength.hpp        <==(generated!)
SdqWavelengthBox.cpp              <==(generated!)
SdqWavelengthBox.hpp              <==(generated!)
SdqWavelengthBoxBack.cpp          <==(generated!)
SdqWavelengthBoxBack.hpp          <==(generated!)
SdqWavelengthBoxSet.cpp           <==(generated!)
SdqWavelengthBoxSet.hpp           <==(generated!)

C:\MyWorkspace\MyPlugin1>
```

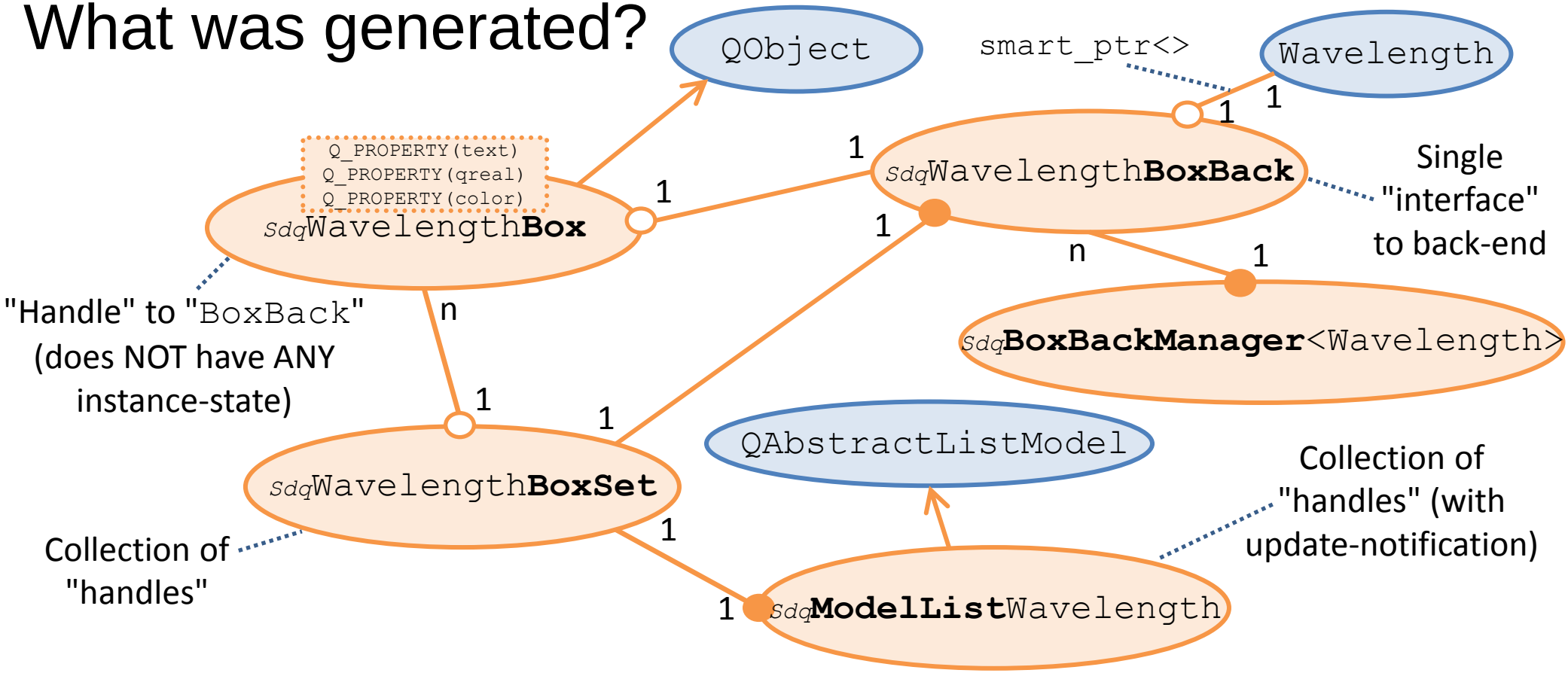
6. **Build/Link all the files in that directory** as a "module" *(e.g., "shared-library").*

1. A build setup that "globs" all "`*.hpp/*.cpp`" files in that directory makes this simple
2. Best Practice:
   1. Only "generated" files are found in this directory (no "hand-maintained" files)
   2. Both "`*.sdgen_sdqbox`" and generated "`*.hpp/*.cpp`" files are checked into the Version Control System

```
./MyWorkspace/.
              MyBin/.
                     CytoPlugin1.dll
                     CytoPlugin1.lib
```

Case Study (continued): What Did The Generator Do?

What was generated?



Class Diagram: A **single "BoxBack" instance represents "state"**. Many "Box" instances are "handles-to" a single "BoxBack" instance (**each "Box" reflects the state represented in the "BoxBack"**).

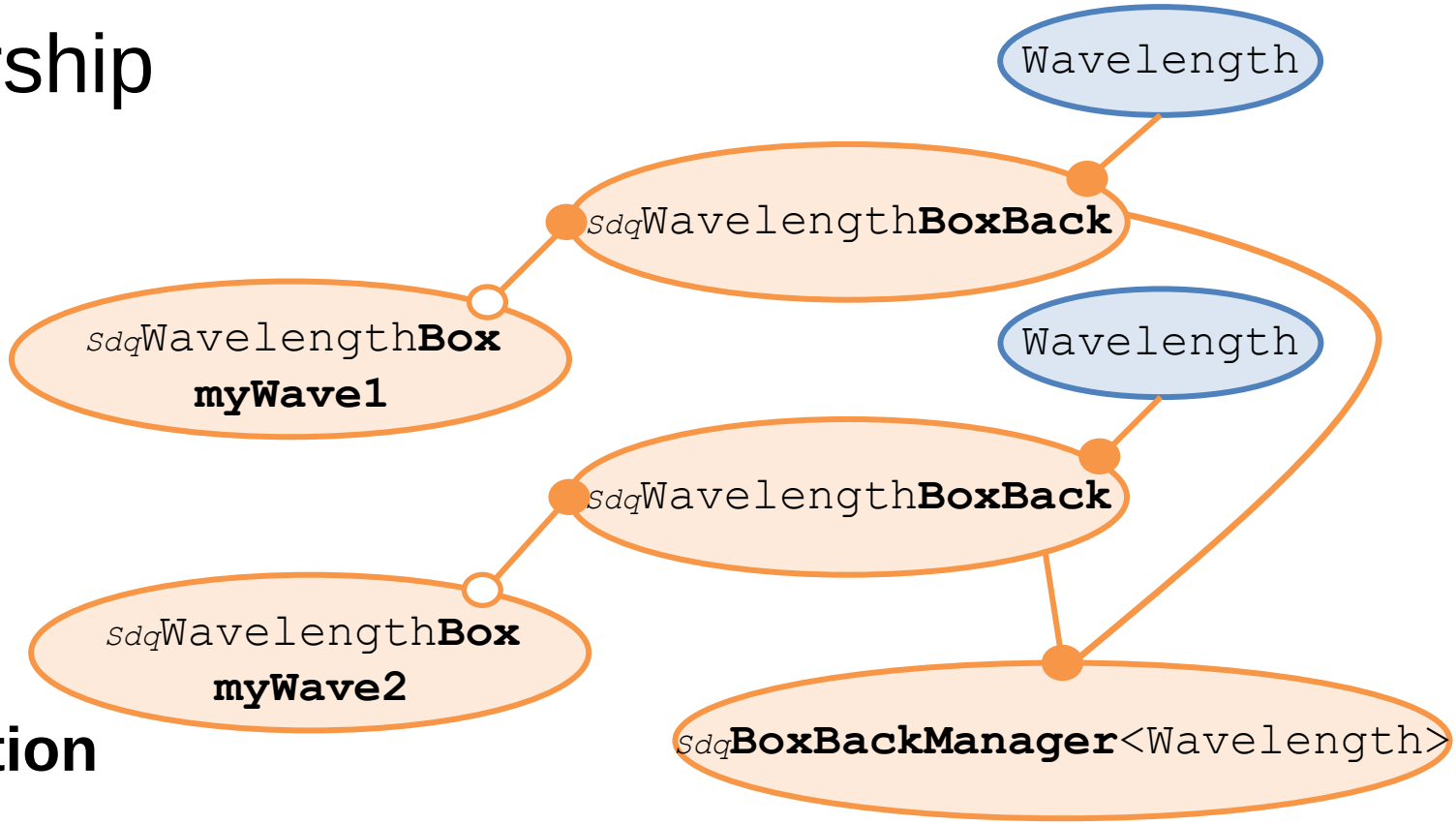
Case Study (continued): Memory Management: Who Owns What?

Object Ownership

```
// FILE: HelloWavelength.qml

import com.bec.cyto 1.0

Item {
  Wavelength {
    id: myWave1
    wavelengthNm: 488
  }
  Wavelength {
    id: myWave2
    wavelengthNm: 532
  }
}
```

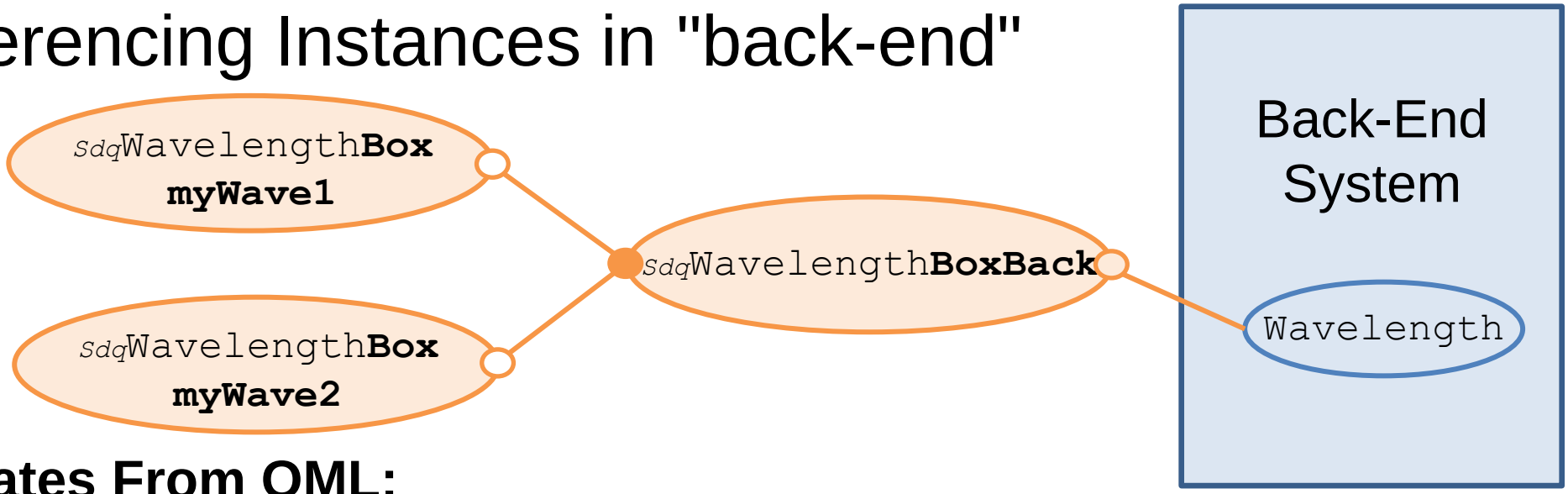


Object Instantiation

- Creating "Box" triggers creation of "BoxBack" which triggers creation of "Wavelength"
- All "BoxBack" instances are "owned" by the "Manager<Wavelength>"

Case Study (continued): Updates From QML: How Does It Work?

Referencing Instances in "back-end"

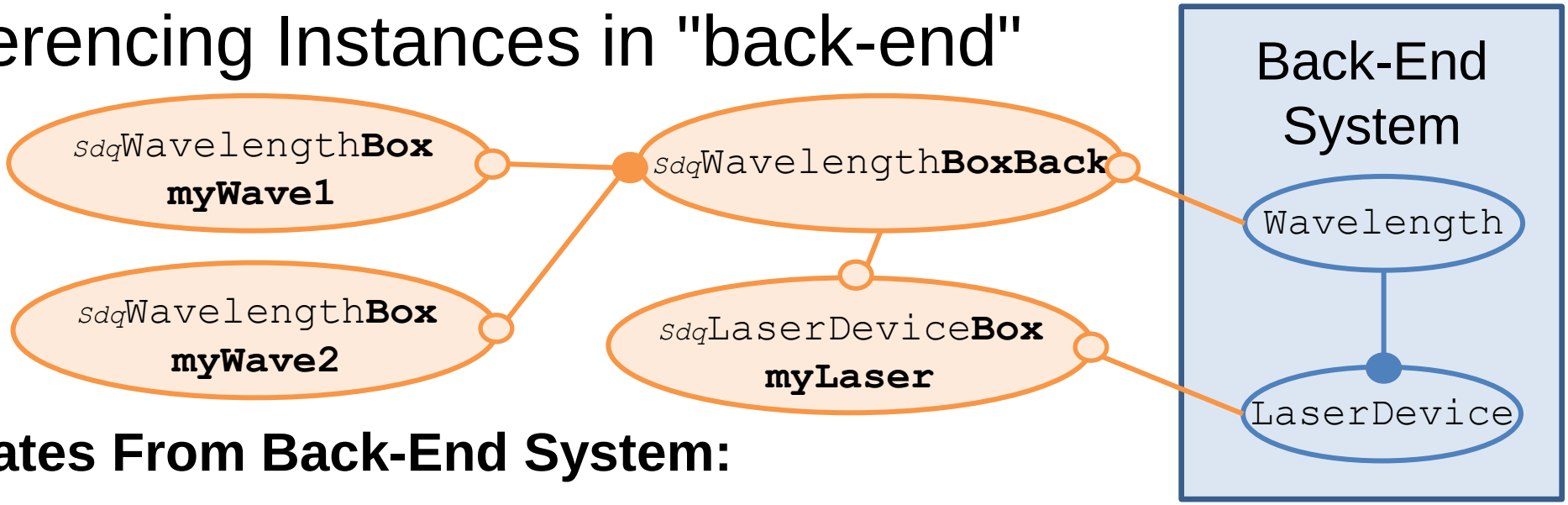


Updates From QML:

1. **"myWave1" ==> "write" property** attempted
2. Call **"forwards" to "BoxBack"**
3. **Value is changed** in "Wavelength" instance
4. **"BoxBack" notifies all "Box" instances** of property change
    1. myWave1 "wavelengthChanged"
    2. myWave2 "wavelengthChanged"

Case Study (continued): Updates From Back-End: How Does It Work?

Referencing Instances in "back-end"



Updates From Back-End System:

1. **"myLaser" ==> notified of change** by back-end system *(can "poll", monitor network traffic, catch a "system-refresh" event, or otherwise be notified explicitly)*
2. **"myLaser" notifies all listeners** its "Wavelength" property changed
3. **"BoxBack" notifies all "Box" instances** of property change
    1. myWave1 "wavelengthChanged"
    2. myWave2 "wavelengthChanged"

Side Note: QML Plugins Are Awesome

(Side Note) ***"Soap-Box": Use QML Plugins!***

- QML Plugins *are Awesome*
- QML Plugins *make your life easier*
- QML Plugins *make you better-looking*
- QML Plugins *make you more appealing for romantic encounters*
- QML Plugins *make you taller*
- QML Plugins *are Awesome*
- QML Plugins *are Really Awesome*
- QML Plugins *are Really Really Awesome*

Summary: USE QML Plugins!!!

…or I will find you…

Side Note #2: Organize Your Projects! *(Or you will Die Die Die!!!)*

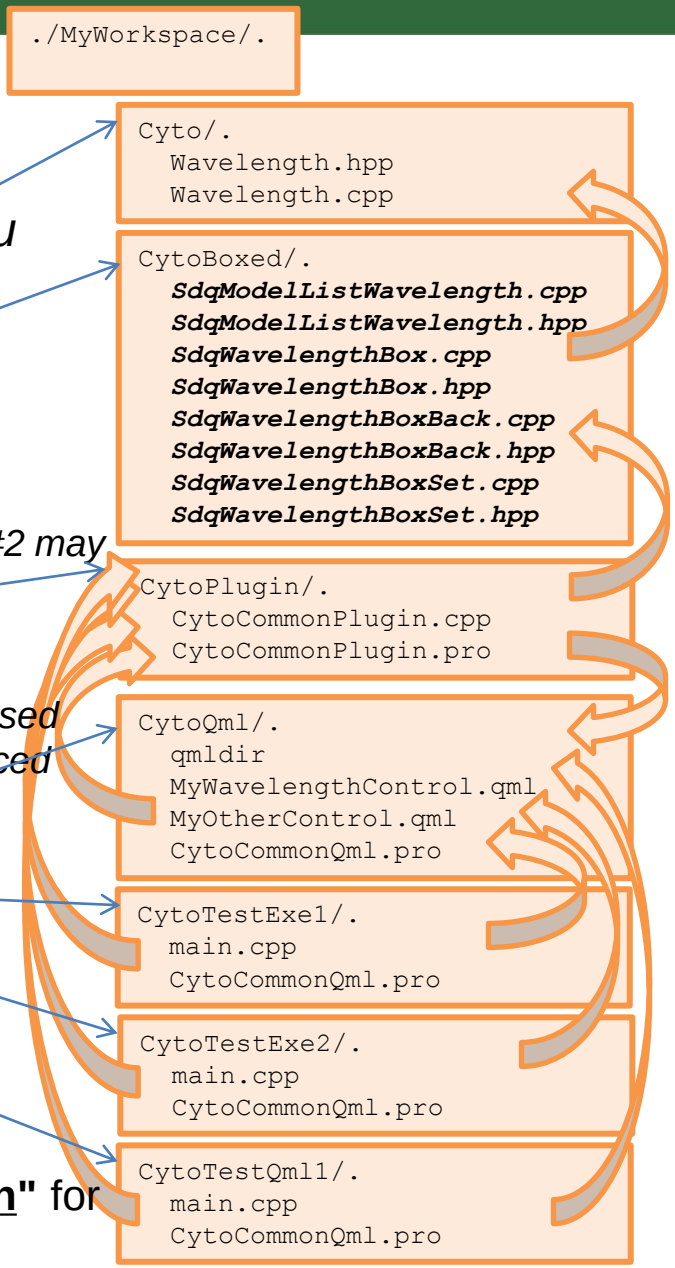
(Side Note #2) **Project Organization**: *Do whatever you want, but this is a convention that works.*

Peers within your workspace *(does not matter if you maintain "one" or "more-than-one" cooperating "workspaces"):*

1. **C++ "Reference/Legacy/Hand-maintained" library/module**
2. **Generated-C++** Declarative-Interface code
3. **QML-Plugin** (shared-library) code *(may be the same as #2, or several libs in #2 may be combined into a single QML-plugin; does not contain QML files, those are developed/tested in (4))*
4. **QML resource files** *(will be deployed with your plugin shared-lib, commonly used across your test-QML applications and test-compiled-applications, are referenced directly by the test applications in the workspace, and (3) for deployment)*
5. **TestApplication code** *(for each compiled "*.exe" binary)*
6. **Test QML Application Code** *(for each QML application)*

NOTE for (3) and (4), that even NESTED plugins are "peers" at the workspace-level

NOTE that each project "builds-and-deploys" to the "install-location" for the machine when developing across modules/plugins



Qt Developer Days 2013

Case Study (continued): Deployment (What Do You Need?)

When Deploying...

- **Executable Binary** – *(e.g., "`MyProgram.exe`"),*
  Example: Users will launch your executable, your system is "locked-down"

  `MyApp.exe` `Cyto.dll` `CytoBoxed.dll`

- **Shared Library** – *(e.g., "`CytoBoxed.dll`"),*
  *Example: Other developers will **link** your DLL into their executable*

  `Cyto.dll` `CytoBoxed.dll` `Cyto.lib` `CytoBoxed.lib`

  ```
  CytoBoxed/.
    SdqModelListWavelength.hpp
    SdqWavelengthBox.hpp
    SdqWavelengthBoxBack.hpp
    SdqWavelengthBoxSet.hpp
  ```

- **QML Plugin** – *(e.g., "`CytoPlugin.dll`"),*
  Example: Other developers will load your plugin; Users will run your plugin

  ```
  CytoPlugin.dll
  qmldir
  MyControl.qml
  ```
  `Cyto.dll` `CytoBoxed.dll` `Cyto.lib` `CytoBoxed.lib`

No Headers are needed to deploy plugins!

Build Products

- Business-logic shared library
  - `Cyto.dll`
  - `Cyto.lib`
- Boxed-shared-library
  - `CytoBoxed.dll`
  - `CytoBoxed.lib`
- Boxed-headers

  ```
  CytoBoxed/.
    SdqModelListWavelength.hpp
    SdqWavelengthBox.hpp
    SdqWavelengthBoxBack.hpp
    SdqWavelengthBoxSet.hpp
  ```
- Plugin-Package

  ```
  CytoPlugin.dll
  qmldir
  MyControl.qml
  ```

Proprietary Headers Are Never Deployed!

Shipping C++ Headers (Expensive!)

Shipping C++ Headers (`*.hpp`) is expensive:

- Are often **volatile, especially early-in-development**
- Can be **volatile late-in-development, causing serious configuration problems** for distributed teams
- **Must be versioned** *(version must be considered in build-use configurations)*
- **May be extensive** *(many headers needed)*
- **May require significant build infrastructure** *(coupling to other headers/libraries; language-specific or technology-specific tooling; special build requirements [e.g., "code coverage", "memory/bounds-checking", "compile-enabled" logging or other diagnostics]; compiler versions likely significant)*
- **Must be managed for use** *(`INCLUDE_PATH` from "origin-workspace" may not integrate well with `INCLUDE_PATH` in "use-workspace")*
- **May not be able to ship due to distribution restrictions** *(3rd Party licensing, Corporate IP / Legal / Regulatory standards)*

Never Ship Domain-Specific Headers!

Ship Only What You Need!

- If shipping **"linked-executable"**, application is "locked-down" for use *(not for further development)*, **no headers are shipped**.
- If using **code-generator, only generated-headers are shipped** *(never need to ship "`Wavelength.hpp`" !!)*
  (Frees team to make changes as-needed!)
- If deploying **QML plugins, no headers are shipped**.
  (QML Plugins Are Awesome!)

Implications: Modules Are Cheap

Implication: Design through *"modules-of-declarative-types"*

- **The "act" of creating a "plugin"** is now:
  1. Create a new "`dir`"
  2. Define "property-definition-files" for the classes to expose
  3. *Generate / build / link / install* the plugin

What would you do **if you could "slice" your domain code base** *(for "free")*, where the "slices" **(plugin-modules) can be arbitrarily combined**?

Leveraging Modules During Development

Time / Function-separation of development teams:

- **HW / Back-End developers "more-active-earlier"**, can stabilize module interface files for GUI team
- **GUI / Application developers "more-active-later"**, only need to pay attention when "declarative-interface-definition" files change. *(HW designers merely re-compile and re-deploy QML-plugin files, GUI developers are completely unaffected unless properties are ****removed****.)*

Separate HW / System verification from UI verification (HUGE BENEFIT!)

  - Easier to **"parallelize" work**
  - Easier to **respond to requirements / design changes**
  - Easier to **fix / repair / modify after deployment**

Module Interface Documentation

"Declarative-interface-definition" files are clean-and-**DOCUMENTED module interface**

- **Represent an "interface-contract"** *(through composition of declarative-properties)*
- Are **checked into version control**
- Can be **specification for external development** teams
- Can be **re-configured** for different types of users

"Slicing" Module Interfaces

Modules are "sliced / re-defined" to **serve a function or purpose** *(by add/remove properties, add/remove def-files)*

- **Different "applications/end-users"** *(only accesses properties for "verified" product)*
  - Different user-products
  - Customer extensions
  - Customer workflow-customization
  - 3rd-Party extensions
- **Manufacturing Personnel** *(accesses factory-configured properties/state)*
- **Service Personnel** *(accesses field-diagnostic properties)*
- **Developers** *(can create "non-supported" and "experimental" configurations, access restricted properties that may damage hardware if used improperly)*

"Sliced" Module Benefit: Configuration

"Sliced" modules enable **better configuration**:

- **Better device sub-system emulation** *(easier during development)*
- Better **verification testing**, better **testing interfaces**
- Better **field-diagnostics**
- Better **product customization** *(e.g., gives Sales staff more options to configure product)*
- Better **"forward-compatibility"** interfacing with future devices, or **3rd-party devices**

"Design By Modules" Summary

"Modules-are-cheap" and **"Modules-are-defined-through-declative-properties"** results in:

- **The act of "System Design" is the Definition of "Modules"** and their relationships *(as it should be!)*
- **Subsystems are better designed and documented** *(are more consistent through property conventions and ontologies!)*
- **Modules are easier to integrate** *(with better backward / forward-compatibility!)*
- Through Modules, **the "synchronous-`get`/`set`" operations** implicitly **become** (somewhat) **asynchronous-bound-property interfaces** *(this solves many system problems, including "refresh/update" issues!)*

Case Study Summary

With this example generator:

- **Does not "touch" domain-specific C++ headers**
- **Exposes to QML *(for cheap!)*** by "hooking-into" `QWidget` apps, MFC apps, etc. *(any system with available C++ headers)*
- These **external QML apps can monitor, externally control, or otherwise "interface" with existing systems** *(without recompiling existing systems!)*
- We **NEVER ship legacy/proprietary headers** *(only "generated" headers if shipping DLLs, no headers if shipping QML plugins)*

Tricky Things For Your Generator

Generator considerations:

- Must **handle “value” and “identity”** semantics
- Must allow **multiple “instances” in QML to represent the same “single-object”** in the underlying system
- **State change in underlying object must notify all QML “instances”** referencing that underlying object
- Must **read / write properties from QML, and from within underlying system**

About: The Case-Study Generator

- **Is implemented in C++** *(would not be difficult to write it in Python / Perl / Ruby)*
- **Uses C++ templates extensively,** *(both composition and inheritence)*
- **Is mostly header-only** *(next revision will likely make it header-only)*
- **Uses C++ "type-traits",** *Example:*

```
template<class ITEM_TYPE_TO_BOX>
struct SdqObjectBoxTraits
{
   typedef ITEM_TYPE_TO_BOX  TypeItemToBox;
   typedef SdqObjectBoxBack  TypeSdqObjectBoxBack;
   typedef SdqObjectBox      TypeSdqObjectBox;
};
```

Future: Generator Case-Study

Possible Improvements to Generator:

- **Adapt to other languages** *(e.g., generate "Python bindings")*
- **Generate "Test Cases"** that exercise underlying system
- **Integration with "build-system"** *(one-step from "dir-of-def-files" to installed-QML Plugin)*
- **Make generated "boxing" network-transparent** *(e.g., QML "boxed-`Wavelength`" in QML process directly reflects state from change in instance in another process [e.g., over TCP/IP])*
- **"Normalize" Data Ontologies, generate specification documentation** *(e.g., "Pretty-PDF" by module, across-modules)*
- **Make wrapper-lib "header-only"** *(is currently "mostly" header-only)*

Who Cares?

What systems might benefit?

- **Legacy systems, code bases** *(does not touch code)*
- **Domain-specific libraries** *(does not touch code)*
- **3rd party libraries** you cannot modify
- **Embedded systems** *(external monitoring)*
- **Hardware control systems** *(external monitoring)*
- **Long-running processes** *(external monitoring)*
- **Time-sensitive components** *(does not interfere with internal timing)*
- **"Deployment" of system for different types of users** *(different interfaces exposed for "users", "manufacturing-personnel", "service-personnel", "developers")*
- **Can "re-wrap" GUI in QML on top of EXISTING SYSTEM** implemented with MFC, `QWidget`s, or other any other GUI library for which you have (or can create) C++ headers!

Case Study Assertion

QML is NOT JUST for GUI. *Exposing to QML is re-wrapping your (imperative) C++ abstractions to declarative-properties that may be bound. It enables "separation" on a scale that implicitly moves your "synchronous" API to a (somewhat) "asynchronous" API. THIS IS A BIG DEAL.*

Describing modules through QML enable modules to "hook-together" through BOUND PROPERTIES. *This is like two modules with "super-magnets" for interfaces that "hook-together" seamlessly, with little effort, correctly, handling all the corner cases. THIS IS AMAZING.*

Example: A "`PeerNetwork`" module (no GUI), and a "`Device`" module (no GUI), could "bind" to each other through "properties" *(because they share the same Data Ontology)*, even though they were developed independently; They bind to each other *(and do so correctly!)* for the first time at deployment.

Summary

In Conclusion:

- Use **QML Plugins**
- A **code-generator can be useful**
- **"Modules" should be "cheap"**, at which point you can **"Design By Modules"**, and you get:
  - **Better (Declarative) Interfaces**,
  - Suited **to the purpose / function**,
  - **With the ability to SCALE** YOUR SYSTEMS.

Thank You!

Special Thanks:

David Van Maren

Beckman Coulter, Inc.

The Qt Community (interest@qt-project.org, development@qt-project.org)

Alan Alpert