
NATIONAL
INSTRUMENTS™

Piotr Kruczkowski
Senior Solutions Engineer

SOLID Design Principles

Software Architecture Design

NATIONAL INSTRUMENTS

Rotten software

Rigidity	The design is difficult to change
Fragility	The design is easy to break
Immobility	The design is difficult to reuse
Viscosity	It is difficult to do the right thing
Needless complexity	Over-design
Needless repetition	Mouse abuse
Opacity	Disorganized expression

Goals of software design

1. Software is soft, easy to change when requirements change
2. Software handles the current requirements

How many times did you write the same code in separate projects?

How much of it could be reused?

What can be abstracted and reused?

- State machine flow
- Thread functionality
- Type safe networking
- Standardized messaging
- XML loading and parsing
- Hardware abstraction layer
- Measurement abstraction layer
- Notifications, dialogs for UI
- Optimized global storage
- Application frameworks
- Dynamic UI - MVC
- Utility modules
- Settings files
- Templates

We need to make a choice – how do we implement these?

Object Oriented Design

What I mean

- Scalable
- Modular
- Reusable
- Extensible
- Simple

- **Standardization**
- **Design principles**
- **Design patterns**

What I don't mean

- Cats, dogs and mammals
- Unreadable code
- Optimistic design
- Artificial complexity
- Meaningless busy work
- No procedural data flow
- Using classes for everything
- Brittle complicated inheritance
- Everyone expert in actor framework

NATIONAL INSTRUMENTS

Object Oriented Design / Common pitfalls

- Lack of data flow understanding
- Undocumented API changes
- Bad naming conventions
- Deep, brittle hierarchies
- Optimistic development
- Big design up front
- Over-architecting

Object Oriented Design / Recommendations

- **Architecture creation is dataflow analysis and type design, everything else is an extension**

- Interfaces > Extensible Components > Components > Inheritance
- Use more than one level of inheritance only in special cases
- Design requires different skills than development
- Learn to recognize design patterns
- Follow SOLID design principles

Object Oriented Design / Recommendations

Threads, objects that receive messages, actors

- Static threads over dynamic threads
- Construct threads only for heavy lifting
- Use subscriber-publisher between loops
- Clearly define lifetime and communication
- Use frameworks and do not reinvent the wheel
- Use abstract messages and implement them

Object Oriented Design / Recommendations

Functional programming, functions and type classes, but no objects

- Only arguments and return values – natural dataflow
- No unforeseen consequences or side effects
- Functions can be arguments and can be partially evaluated
- Partial functions are a thing
- Make operations undoable
- No global variables (references are also global variables)
- Classes > Type Definitions > Clusters > Base Types
- Input – Function – Output – Function - …

- Understanding **Channel Wires** helps to think in a functional programming way

Object Oriented Design / Key Takeaways

- Allows to standardize on best practices, modules, patterns and solutions
- Can be limited to things that provide highest return on investment
- Is very similar to the development we already do or should do

Object Oriented Design / Key Takeaways

Paraphrasing from stackoverflow.com

Q: Why include support for Object Orientation (OO) directly in the language when procedural languages can be used to design and write modular, reusable and extensible code?

A: To have a standard for how complexity is expressed in the source code, so you don't end up with 22 different implementations for the same thing.

SOLID Object Oriented Design Principles

- **Single responsibility** – Classes should do a single logical thing
- **Open/Closed** – Open for extension but closed for modification
- **Liskovs substitution** – You should be able to substitute child for parent
- **Interface segregation** – Single responsibility in interface design
- **Dependency inversion** – Put interfaces between classes

SOLID Object Oriented Design Principles

- Single responsibility – Classes should do a single logical thing
- Open/Closed – Open for extension but closed for modification
- Liskovs substitution – You should be able to substitute child for parent
- **Interface segregation – Single responsibility in interface design**
- **Dependency inversion – Put interfaces between classes**

- Two of these points are directly tied to interfaces
- One of them can be done in LV i.e. the dependency inversion

Single responsibility – Classes do a single logical thing

- *Text formatting class should not save to file*
- *Measurement class should not define visualization*

- Classes need to work together through well defined APIs and interfaces, instead of implementing everything they need themselves
- To the end user it can look like one API is doing everything, but that API can be encompassing many classes in the background

Open/Closed – Open for extension, closed for modification

- *Algorithm that defines most steps, but allows new steps to be defined*
- *Sequence the defines the flow of operations without hardcoding the details*

- The moment you finished gathering requirements, they are already outdated
- Need to be able to add missing functionality
- Having a design that is closed for modification protects it
- Think of plugins, dynamically loaded when needed
- Can be VI Server based or object oriented
- Require VI connector pane pattern i.e. interface

Liskovs substitution – Allow to substitute child for parent

- *An application written for a parent should not break when a child of that parent is used instead*

- Inheritance is dangerous and difficult to understand, especially when it grows in layers
- Limiting the number of layers to one, or maximum two when interfaces are involved, is the general guidance
- It is safer to build complexity by composition than inheritance, because this is what people understand as adding modules

Interface segregation – Single responsibility in interface design

- *Not every class that is sortable has to be searchable etc.*

- The interfaces for operations need to follow the same single responsibility principle all classes should
- Impossible to implement in pure LV due to single inheritance
- **Class mutation mechanism has to be used**

Dependency inversion – Put interfaces between classes

- *Measurement application should not depend on the driver details, instead the driver details should depend on a HAL. The application should be using that HAL instead, with the driver details hidden behind the abstraction.*

- This can be done in LV, as long as there is only one interface for a class
- The moment you need two or more independent functionalities and logically two interfaces, **you need to use class mutation mechanism**

Limitations of LV OO

- The only limitation that doesn't allow to use two of the five principles is the lack of multiple interfaces
- For this reason implementation of class translation mechanism is used as a solution
- By extending pure LV OOP with class translation you can achieve and follow all design principles

Class Translation - Interfaces in LabVIEW

SOLID Design Principles

- Single responsibility – Classes should do a single logical thing
- Open/Closed – Open for extension and closed for modification
- Liskovs substitution – You should be able to substitute child for parent
- **Interface segregation – Single responsibility in interface design**
- **Dependency inversion – Put interfaces between classes**

- Two of these points are directly tied to interfaces / abstract classes / multiple inheritance of interfaces
- Only one can be done in LV i.e. the dependency inversion

Dependency inversion – Put interfaces between classes

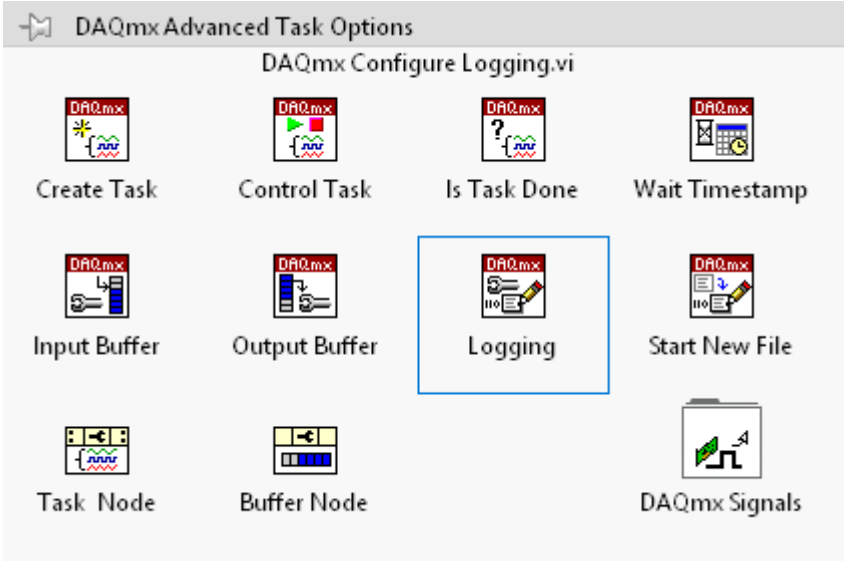
- *Measurement application should not depend on the driver details, instead the driver details should depend on a HAL. The application should be using that HAL instead, with the driver details hidden behind the abstraction.*

- This can be done in LV, if there is only one interface for a class
- The moment you need two or more independent functionalities and two interfaces, **you need to use something custom**

Interface segregation – Single responsibility in interface design

- *Not every class that is sortable must be searchable etc.*
- *Logging measurements to file should not be done by measurement class.. as it is now in DAQmx*

- The interfaces for operations need to follow the same single responsibility principle all classes should
- Impossible to implement in pure LV due to single inheritance
- **Class translation can be used to solve the problem**



NATIONAL INSTRUMENTS

- The only limitation that doesn’t allow to use two of the five principles is the lack of multiple interface inheritance
- Class translation can be used as a solution
- By extending pure LV OOP with class translation you can achieve and follow all design principles

Twin pattern

Connected to: Portland Pattern Repository | The Hillside Group | Linda Rising

From Wikipedia, the free encyclopedia

In software engineering, the **Twin pattern** is a software design pattern that allows developers to model multiple inheritance in programming languages that do not support multiple inheritance. This pattern avoids many of the problems with multiple inheritance.[1]

Definition

Instead of having a single class which is derived from two super-classes, have two separate sub-classes each derived from one of the two super-classes. These two sub-classes are closely coupled, so, both can be viewed as a Twin object having two ends.[1]

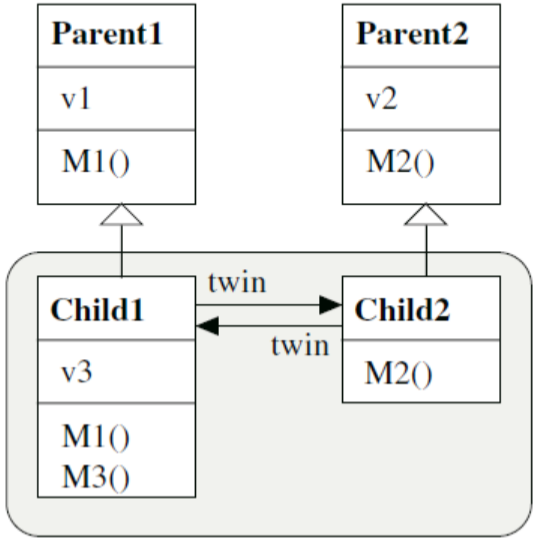
Applicability

The twin pattern can be used:

M2()
M3()

[1]

The following diagram shows the Twin pattern structure after replacing the previous multiple inheritance structure:

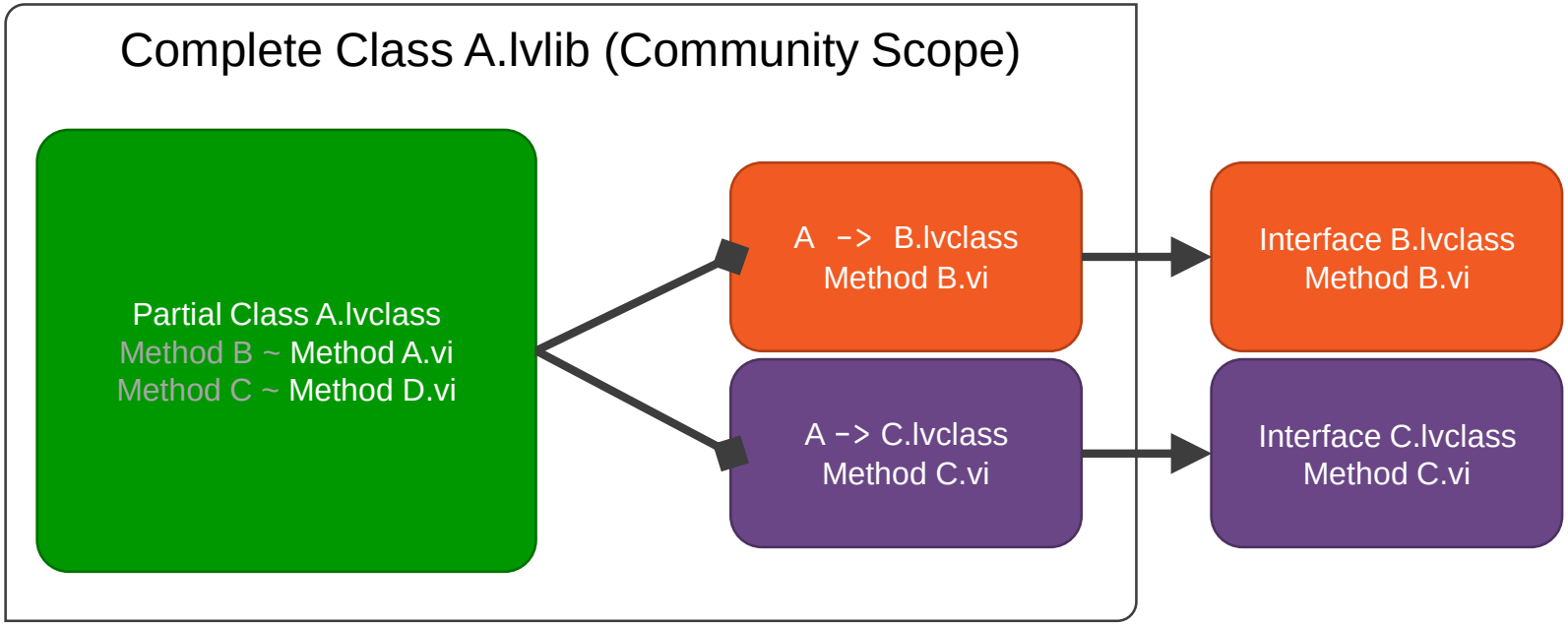


[1]

Collaborations

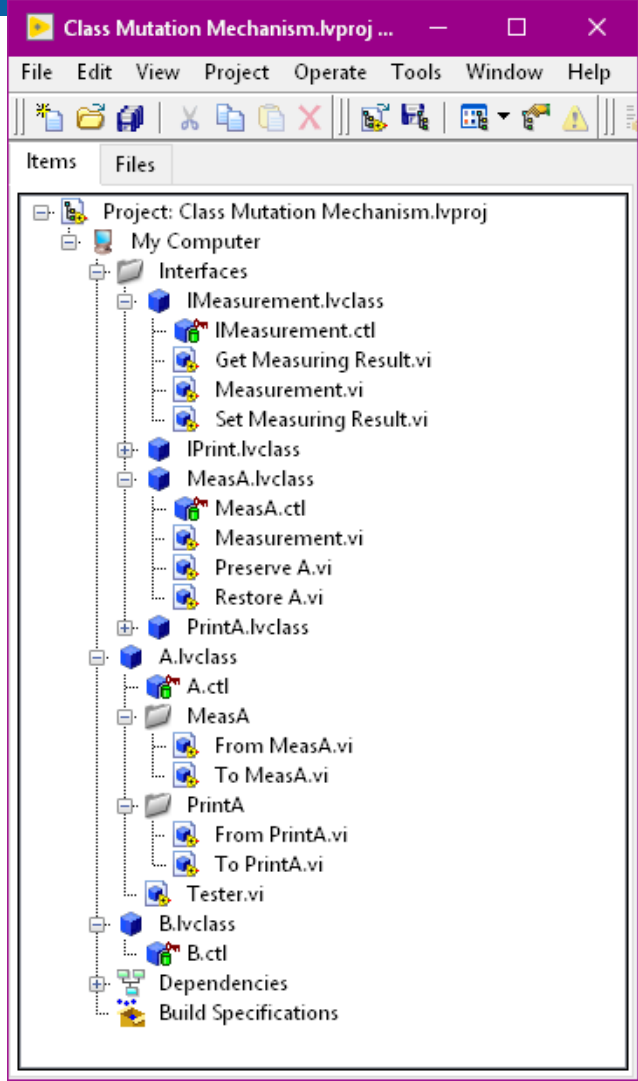
Class Translation / Twin Pattern / State Actors / Class Composition

“Composing” a class from smaller classes



Class translation - Project

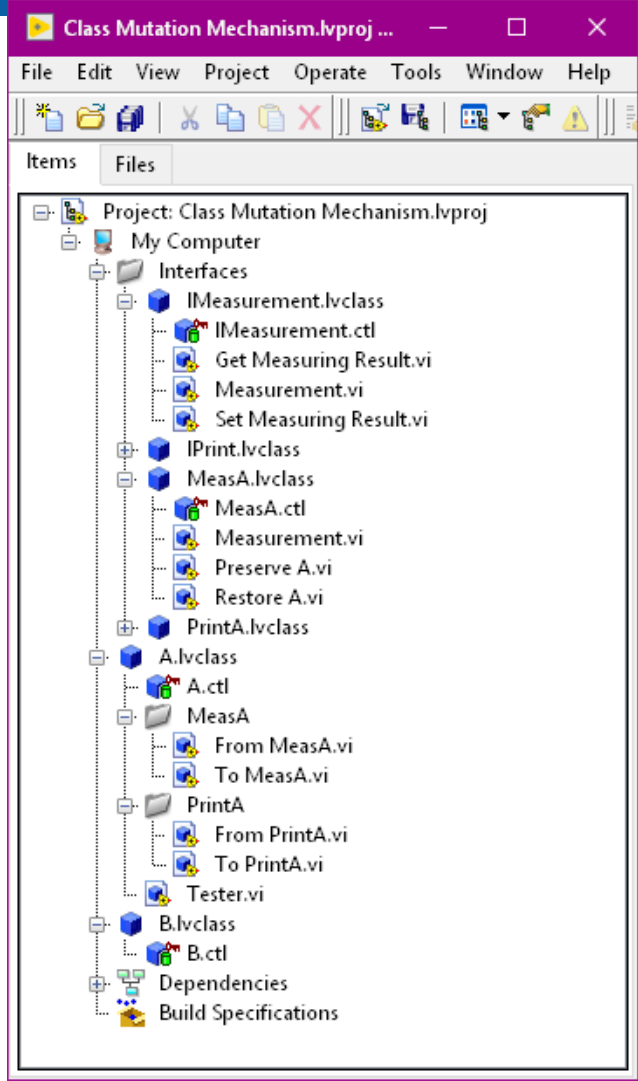
- Interfaces separated
- Interfaces do not need to be pure virtual
- Complete class A = class A + interface implementations
- Two translation methods for each interface



Class translation - Project

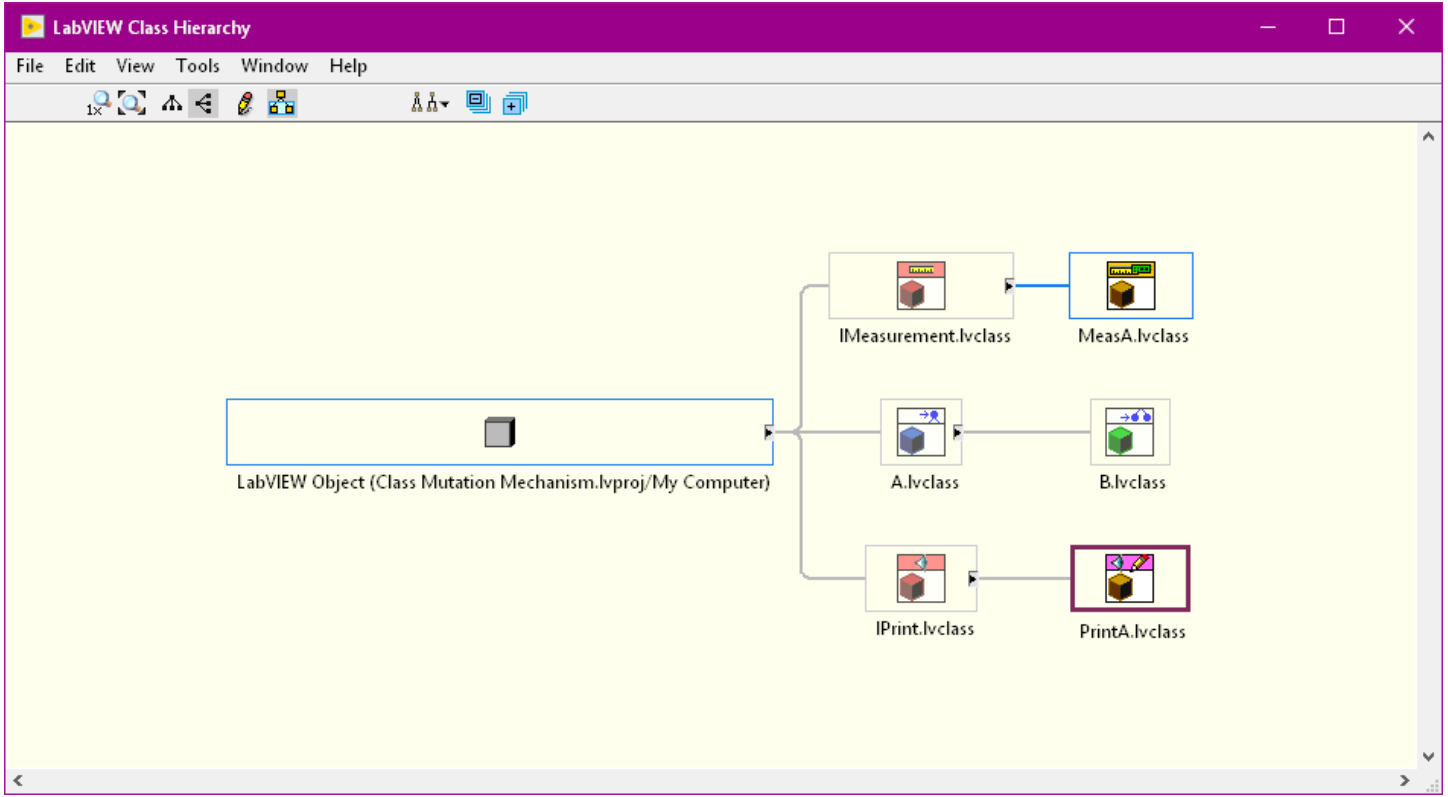
- Interfaces provide only the function prototypes
- Interface implementations are friends of class A
- Data accessors with community scope, or wrapped target methods

- Write your big design with abstract interfaces
  - Specify details for the specific domain

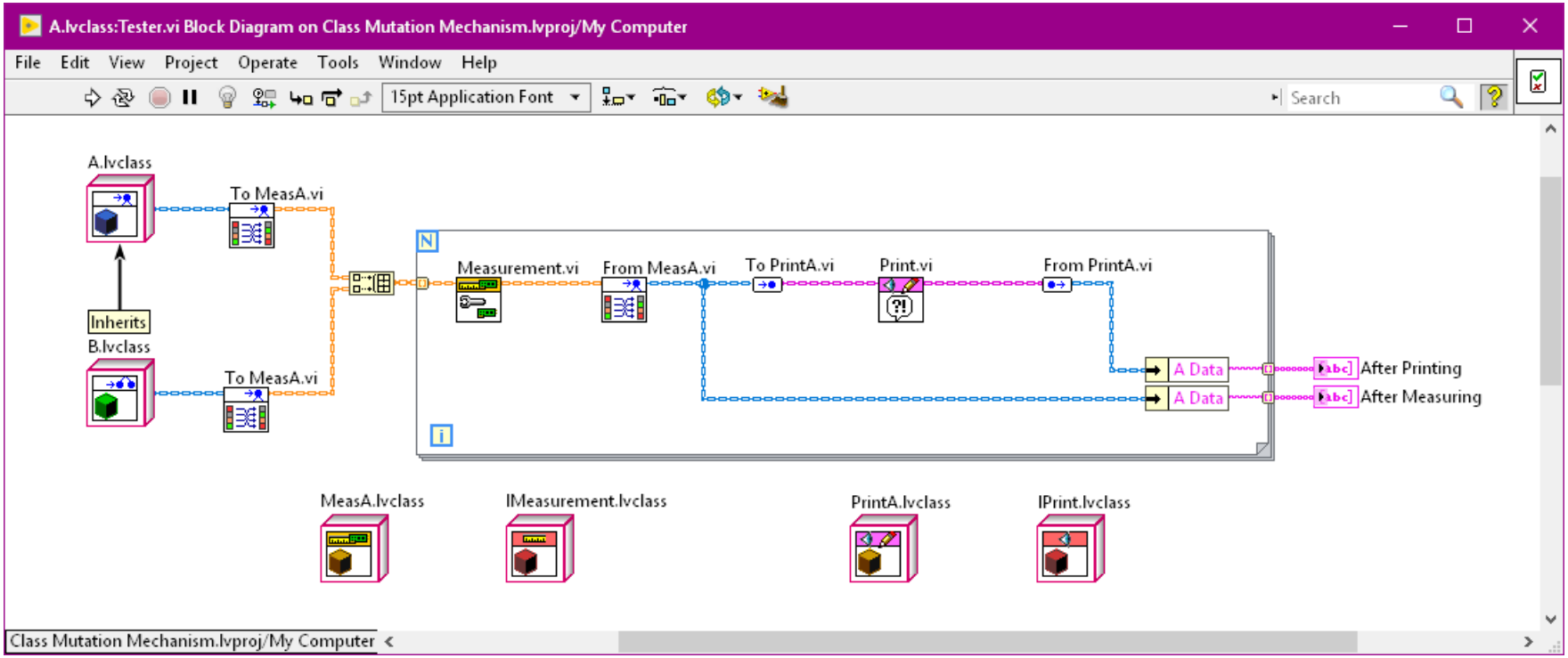


NATIONAL INSTRUMENTS

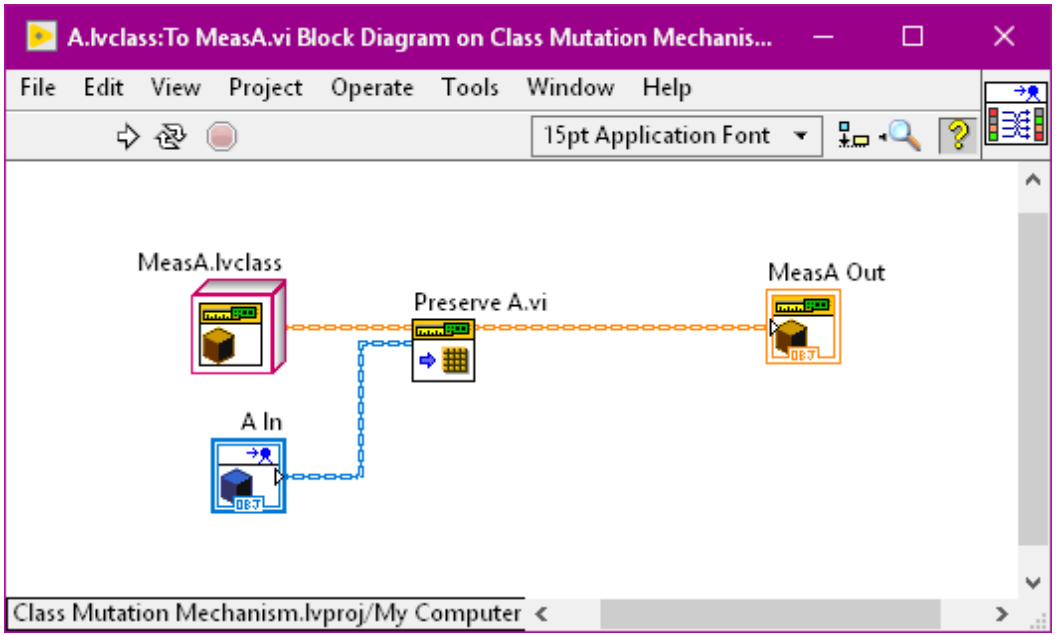
Class translation - Inheritance



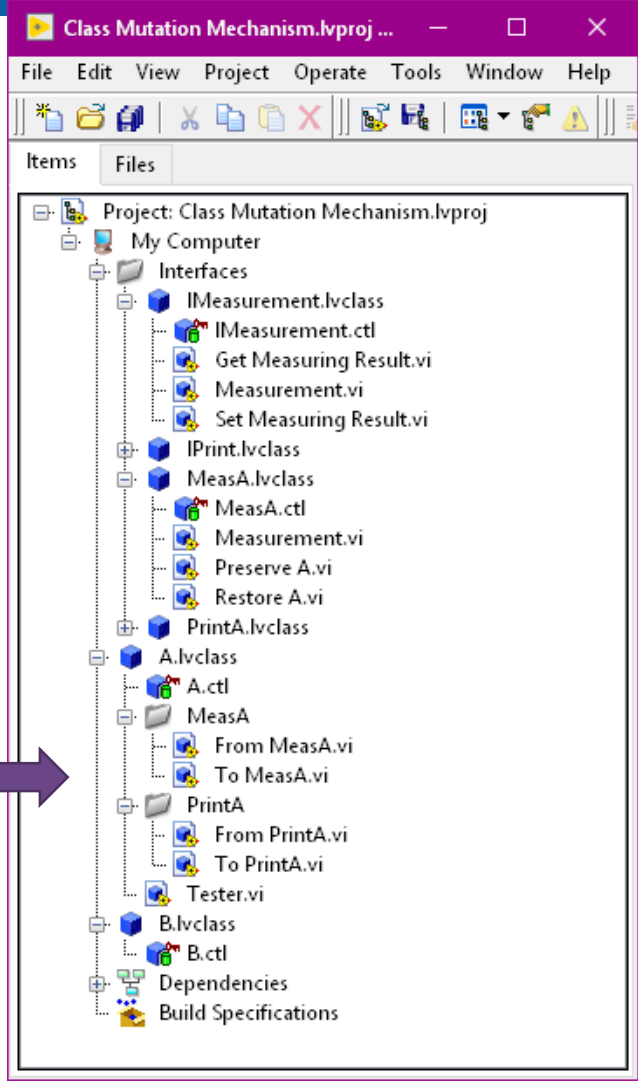
Class translation – Main Tester



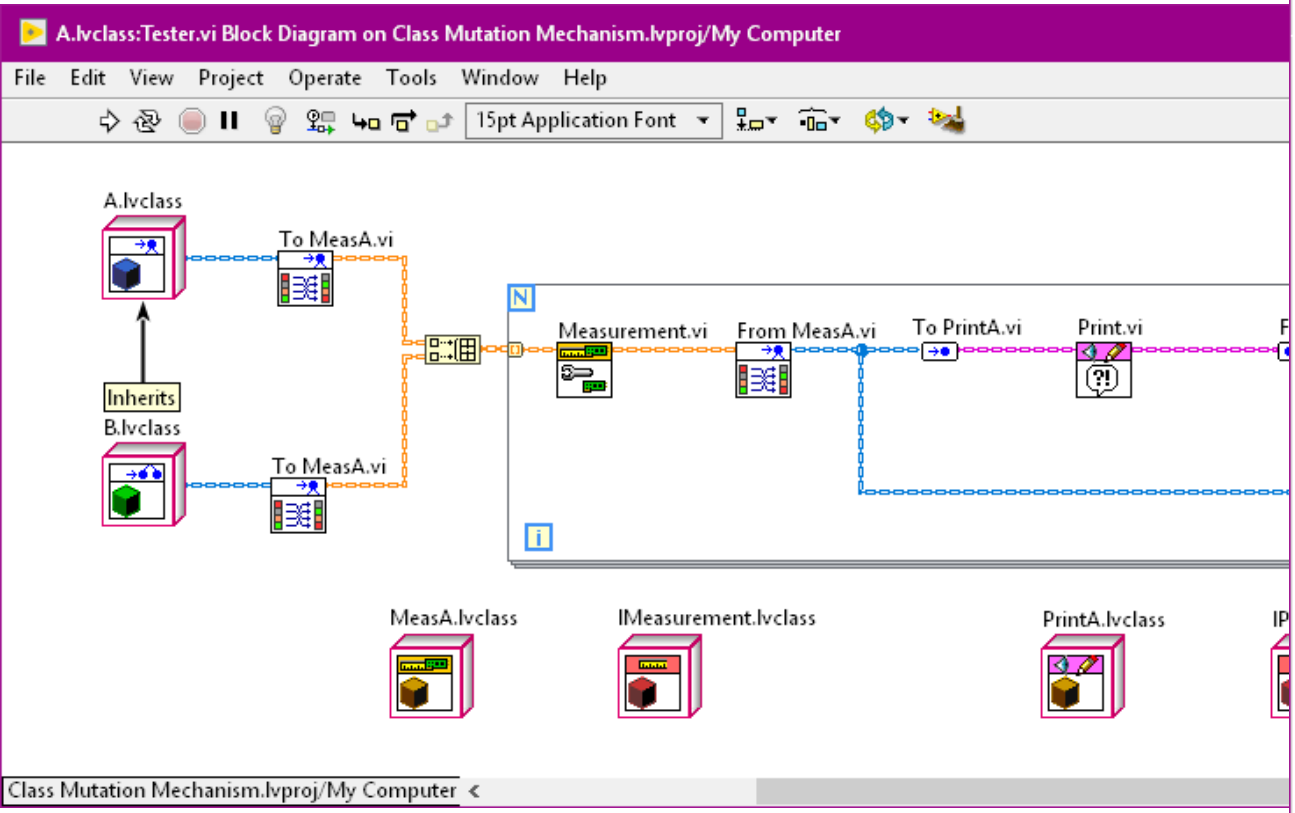
Class translation – Translation To Interface

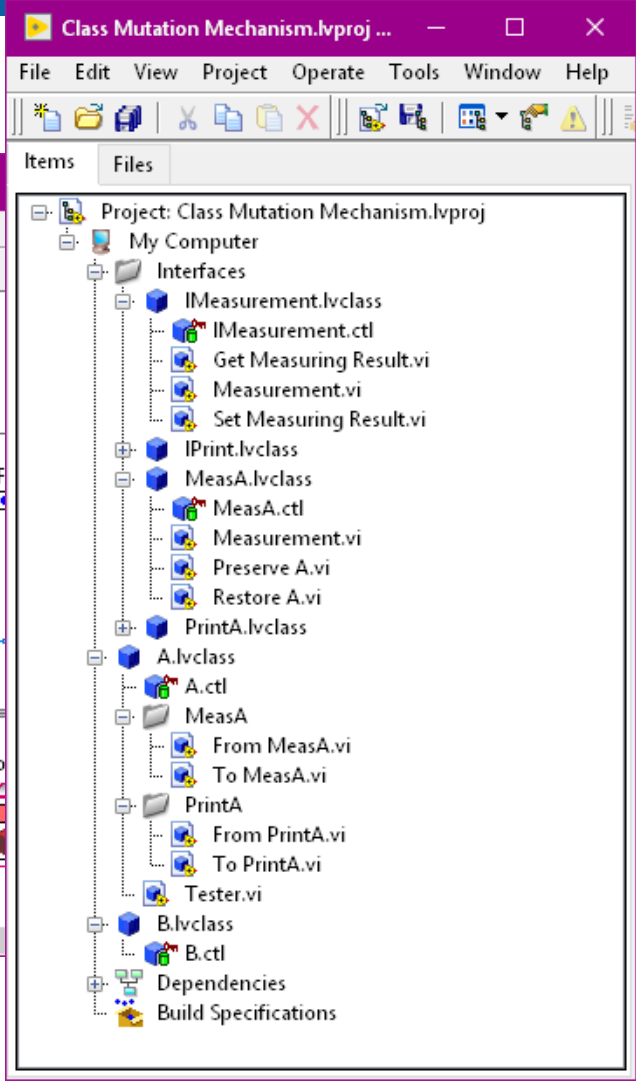


- Preserve the whole original class or any parts of it you need



Class translation – Calling the Interface Method

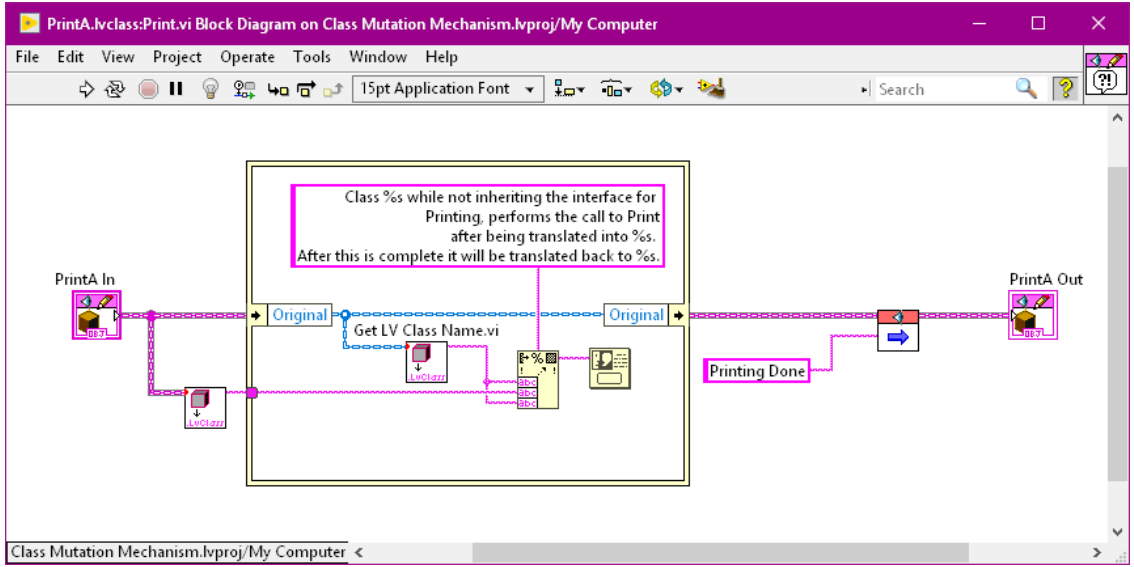




NATIONAL INSTRUMENTS

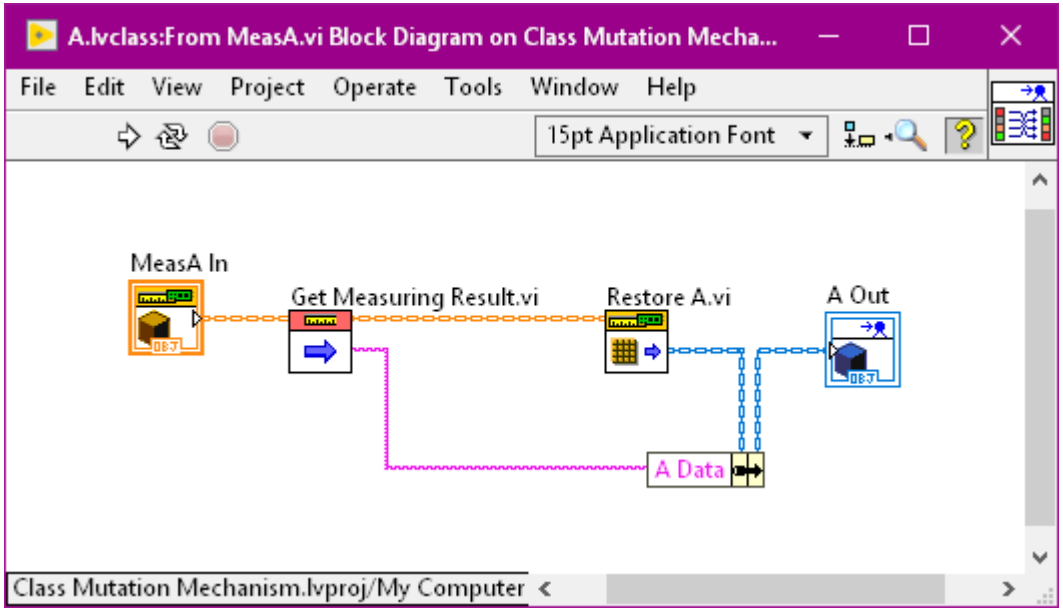
Class translation – Inside Interface Method

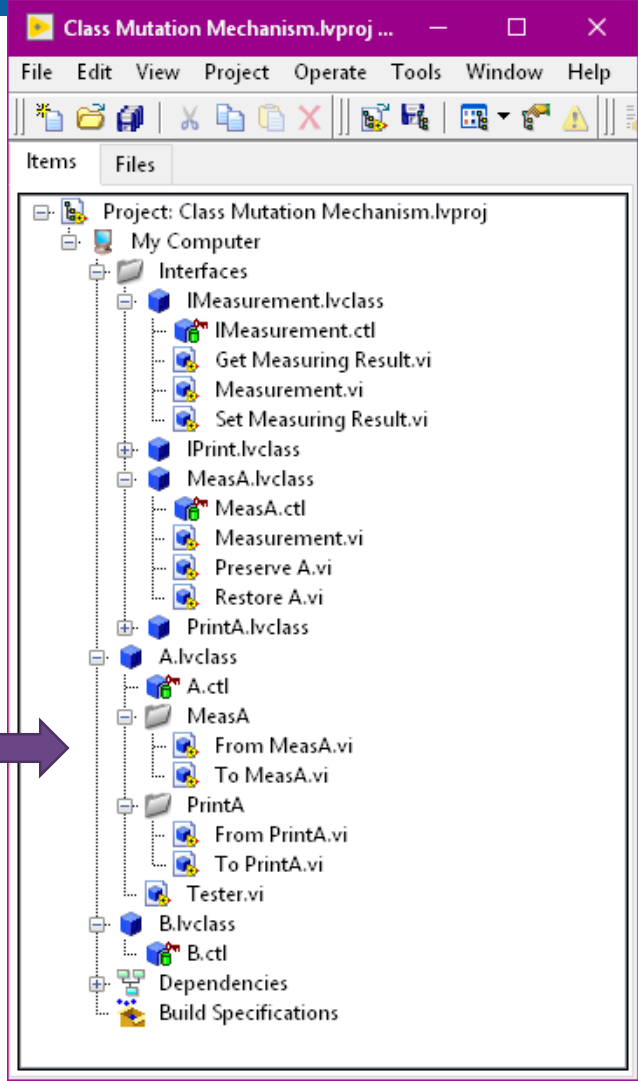
- Either call getters and setters to obtain and write data you need
- Or invoke the target methods themselves



NATIONAL INSTRUMENTS

Class translation – Translation From Interface





- Translate back the changes to data you care about

NATIONAL INSTRUMENTS

Analogous Python Example

Using it all together

```
#Create original class with no access to interface
a = A()
#Translate the original class into a class that has access to
interface
aIm = a.IntoAMeasure()
#Invoke the interface method on the translated class, by
invoking it on the preserved original
aIm.Measure()
#Translate back into the original if some data is returned by
value
a.FromAMeasure(aIm)
```

Analogous Python Example

Interface and class implementing it

```
#This is a standard interface specification, abstract method with some arguments.
class IMeasure():
      def Measure(self):
            print("this is the interface, no implementation")

#This is a standard implementation of an interface inheritance with the added twist that
there is some original class that we remember inside.
class AMeasure(IMeasure):
      original = 0

      def __init__(self, original):
            self.original = original
      def Measure(self):
            self.original.Increment() # ← This is the thing we want to see
            print("This is Measure method of AMeasure class, corresponding to AThing class.")
```

Analogous Python Example

Class not implementing interface, but using another class which does

```
#Class A class does not inherit from the interface, yet it uses another class that does
inherit from the interface for it.
class A():
      counter = 0

#Here we define the mechanism to mutate into the interface class AMeasure.
      def IntoAMeasure(self):
            return AMeasure(self)

#Increment function and the internal counter are visualizing that the mutated, interface
class can invoke methods on the original it stores, instead of inventing its own magic.
      def Increment(self):
            self.counter = self.counter + 1

#Here we mutate back to the original class. This might be optional.
      def FromAMeasure(self, bmeasure):
            self = ameasure.original
```

Class translation Mechanism – Pros & Cons

Good

- **Interfaces!**
- **Achieves the interface segregation principle and dependency inversion at the same time**
- Enforced separation of responsibilities
- Many developers can work independently
- The interface doesn't have to be pure virtual e.g. traits
- Code built by composition instead of inheritance
- No references – everything still by value

Considerations

- Need to implement the translation methods
  - Could by automated by a project provider or script
- Class will be "composed" of many interface implementation classes
  - Same as for Actor Framework
- Needs clear API between the class and its interface implementing classes
  - We should already strive for this in design

Accidental Benefits

Things I didn't think about but are quite awesome for LabVIEW

- Keeping the design inheritance tree short is preventing the architecture from becoming brittle
- Instant abstraction layer
- Allows sharing development tasks with "partial" classes
- Clean separation of development into libraries, based on capabilities of classes

Demonstration

FAQ

- **What about type casting and To More Specific Class?**
  - Less of a problem if you operate in the same VI and know the original class type, before translating to Interface implementing subclass. Otherwise handle type error.
- **What about malleable VIs?**
  - Great for static typing only but cannot adapt dynamically. Working with arrays is not possible.
- **Is there a way to make this less complex?**
  - In my investigation I figured out that the minimum number of extra VIs is four i.e. To and From interface class, with preservation and translation inside.
  - These need to be created once and can be scripted.
  - Calls to methods can then be implemented quite easily, but there is one extra layer of wrapping for each method.
- **How to prevent project bloating with exponential translation combinations?**
  - Always translate to and from the original class in all subclasses. This way the number of extra Vis is grows linear. Otherwise you go exponential… you don't want to go exponential.
- **It takes to much space on the diagram to translate to and from.**
  - Make the icons smaller, like primitives. Chain multiple interface calls together.
- **What about performance?**
  - If you only preserve the original class object there should be no performance hit. The only impact is some additional translation code. THIS IS NOT TESTED

Bonus Round - Actors

How do things look when you send messages

- Additional layer of abstraction between two classes implemented via message
- Message is a class
- Classes can be abstract
- Messages can be abstract interfaces
- Implementations of messages can inherit from abstract messages

- **Which means… multiple interfaces without any limitations!**
- **There is no need for class translation to be used**

Bonus Round - Actors

Instant dependency inversion