FULL FORCE AHEAD

25 YRS OF NIWeek May 20–23, 2019 | Austin, Texas
Download and Login to the NIWeek Mobile App

VIEW WEEKLY SCHEDULE

FIND YOUR SESSION

BUILD YOUR SCHEDULE

More Information at ni.com/niweek
Traits: A New Approach to Designing Reusable Code

Dmitry Sagatelyan

Certified LabVIEW Architect
LabVIEW Champion
sagatedm@arkturtech.com

Arktur Technologies LLC
NIWeek 2019
Abstract

Reusing properly engineered code saves time and money. OOP is currently the primary vehicle for driving Code Reuse. LabVIEW OOP is based on Single Inheritance. It does not provide support for Interfaces or more recent code reuse mechanisms, such as Mixins or Traits.

Presentation reviews different code reuse techniques, issues hindering design of reusable code with Multiple Inheritance and Java/C# Interfaces and describes ways to work around lack of Interfaces in G.

It concludes with an introduction to Traits (a new concept of fine-grained reusable code units), a proposal for implementing Traits in LabVIEW and examples showing how to use Traits for designing reusable code.
About Myself ...

- Software Engineer by Training
- Masters of Science in Computer Science
- Using LabVIEW since 1998 (LV 5.0)
- Certified LabVIEW Architect (2012 – 2026)
- LabVIEW Champion
- Full Time LabVIEW Consultant in San Francisco Bay Area (Arktur Technologies LLC)
- Passionate about:
  - Using Contemporary SW Engineering Methods in LabVIEW (ex: SOLID Principles)
  - Actor Programming
- Regular presenter @ NIWeek, CLA Summits, Bay Area LUG
- Designed several Actor Frameworks (2005/07/11/15)
- Designed ArTLib – the first (?) LabVIEW Reuse Library built on SOLID Principles (2011-19)
- Presentation List
Presentation Outline

1. What is Code Reuse?
2. Code Reuse Techniques
3. LabVIEW Code Reuse Constraints
4. Issues with Multiple Inheritance & Java/C# Interfaces
5. Introduction to Traits
6. Proposal: Traits in LabVIEW
7. Discussion
How to Become More Productive?

Two ways to become a more productive LabVIEW Developer:

• Learn to wire faster than Darren Nattinger
• Learn to [seamlessly$^1$] design Reusable Code

However:
• Wiring fast tends to add similar code instead of creating reusable code units
• Testing Code: N (similar units) vs 1 (reusable unit)
• Maintenance: N (similar units) vs 1 (reusable unit)

• Maintenance Cost/Development Cost Ratio is $\sim [1.5 \ldots 9]$ (per $[1]$)
• Do the math yourself ...

$^1$seamlessly: without much extra effort
What is Code Reuse?

“A Software Component is reusable if it is applicable in contexts other than the one it had been developed” [12]

**Code Reuse**
- A Single copy of Class X is shared by all callers
- Class X code cannot be altered by caller (must be read-only or in binary form)

**Code Recycling**
- Each Application gets its own copy of Class X
- Each copy has to be tested
- Bug Fixes & new Features need to be implemented in each copy
- Copies tend to diverge in time
Reusable Code Libraries & Frameworks

How to:
- Discover ?
- Trust ?
- Support ?

This is the only reuse method you have without OOP ...

496 Libraries/Toolkits in LabVIEW Tools Network
Code Reuse through Inheritance

Inheritance abuse – creating artificial class relationships to achieve ‘better’ Code Reuse...
Code Reuse through Dynamic Dispatch

Child class overrides a subset of parent class methods, reusing the rest

Dynamic Dispatch enables applying Open-Closed Design Principle (OCP):
“Software entities should be open for extension, but closed for modification”
Encapsulation enforces decoupling of Client code from Component code implementation details - enabling **Client code reuse** - Components implementing the same interface can be swapped without any changers to Client code ...

LabVIEW class data is always Private

**Accessor Method** are required to Read or Write Private Data values

‘Too many’ Accessor Methods in a class are a smell of **Leaky Abstraction**
Code Reuse through **Subtype Polymorphism**

- **Serializable** array \([n]\) of **Serializable**
- **Launch.vi** Launches
- **Viewer** Displays
- **Sort.vi** array \([n]\) of **Comparable**

**Reused Code**

**Interface**

- **Actor**
  - Implements **Serializable**
  - Implements **Viewable**
  - Implements **Comparable**

- **My Actor** Extends **Actor**
- **Control GUI** Extends **Mapper**

- **File**
  - Extends **Comparable**

- **Text File** Extends **Comparable**

Single Inheritance severely limits using Subtype Polymorphism
Code Reuse through SOLID Design

SRP The Single Responsibility principle
OCP The Open-Closed Principle
LSP The Liskov Substitution Principles
ISP The Interface Segregation Principle
DIP The Dependency Inversion Principle

Practicing SOLID Design Principles results in design of highly reusable code. Components are decoupled from each other through properly defined Interfaces – limiting changes to only those Components that are directly affected by changes in Requirements ...
Acania Monitoring System Code Reuse

Traits: A New Approach to Designing Reusable Code

AMS Actor

Diesel Unit

Bilge Alarm

Other

AMS Main

Simulators

Diesel Unit Actor

Bilge Alarm Actor

Other Actor

AMS Assembler

TAP Message Broker

Message Broker

Global Log

ArT Log

ArT Host Error

ArT Error

AMS-Specific Code

Reuse Libraries

Interfaces

Control GUI Actor

Pilot GUI Actor

Keyboard Handler

Boat Platform Code

AMS-Specific Code

Creates

Configures

Runs

Extends

Creates

Configures

Runs

Extends

Extends

Implements

Extends

Extends

Extends

Starts

Implements

Implements

Implements

Implements

NIWeek 2019

Traits: A New Approach to Designing Reusable Code
# Acania Monitoring System Code Statistics

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>AMS</th>
<th>FPGA</th>
<th>Messaging</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIs/Methods</td>
<td>583</td>
<td>233</td>
<td>57</td>
<td>76</td>
<td>274</td>
</tr>
<tr>
<td>Classes</td>
<td>57</td>
<td>22</td>
<td>8</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>Actor Classes</td>
<td>9</td>
<td>8</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Actor Commands</td>
<td>156</td>
<td>152</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Actor Objects</td>
<td>25</td>
<td>9</td>
<td></td>
<td>7:9</td>
<td></td>
</tr>
</tbody>
</table>

AMS code has 309 Boat-specific VIs:
- 80% (250) are Reusable
- 12% (37) are Recyclable at 50% of original effort
- 8% (22) are Recyclable at 75% of original effort

Repurposing AMS code to another boat should take ~11% of original effort (9x productivity increase) plus budget for implementing new functionality (if any).

Which gives me substantial leverage bidding on the next boat project, while keeping a comfortable profit margin ...
Austin, we have a Code Reuse Problem

Our main disadvantage as G Developers comes from
Upper Limit = 1

LabVIEW 8.2
(2006)

Superclass

Class

C++ 2.0
(1989)

Superclass

Class

Java/C#

Superclass

Interface

Class

Scala
(2004)

Superclass

Trait

Class

Extends

0..1

Extends

0..N

Extends

0..1

Extends

0..N

Implements

0..N

With
By-Value Interfaces in LabVIEW

Issues:

- Single Interface per Class
- Does not prevent calling ‘abstract’ class methods

... better not skip this call!
ArTLib Core Class Hierarchy

Single Inheritance: reuse is achieved by placing a class in the right position in Class Hierarchy

Where should I place ArT_Persistable Class?
Simple Unit Test Design

Cannot be implemented with Single Inheritance

ArT_Error
  ↓
ArT_Host_Error
  ↓
ArT_Your_Error

ArT_Identifiable
  ↓
ArT_Initializable

ArT_Identifiable_UT
  ↓
ArT_Unit_Test

Unit_Test_Harness.vi

ArT_Class
  ↓
ArT_Your_Class

ArT_Class_UT
  ↓
ArT_Your_Class_UT
Working Solution... finally!

But have to live with ISP violation...
By Reference Interfaces in G

Synchronous Interface

1. A regular ‘By Value’ stateless Class
2. OK to branch and pass around
3. Can be easily extended via Single Inheritance
4. A pain to create & manage by hand …
Multiple Interface Support in G Traits: A New Approach to Designing Reusable Code

Façade Class

Has a DVR

HT_Configuration_Handler constructor:
returns all Façade API Objects

HT Configuration Handler can only be called via Façade API Objects !!!

This design works fine – but requires creating and managing boilerplate code for Façade Classes

HT_Configuration_Handler constructor:
returns all Façade API Objects

HT Scanner Config

HT Fluidics Config

HT Configuration Handler

HT Assembler Config

This design works fine – but requires creating and managing boilerplate code for Façade Classes

NIWeek 2019 23

Traits: A New Approach to Designing Reusable Code
Summary:

1. I would not recommend using either of three Multiple Interface Solutions for designing Reusable Code on regular basis.
2. Enabling Subtype Polymorphism is as important as keeping one’s code easy to write, understand and maintain. **Sacrificing one for another appears to be a false choice**
3. I would very much prefer LabVIEW R&D adding Subtype Polymorphism support to G …
The Diamond Problem (in C++)

1. Would D have one or two copies of ‘a’ ?
2. D calls A constructor once or twice ?
3. What if A constructor creates a resource ?
4. What is B and C Constructor call order ?
5. Which ‘a’ data copy gets passed to a call using Subtype Polymorphism ?
6. What happens when D calls af() – will this result in executing B::af() or C::af() ?

C++ requires explicitly using namespace (::) to resolve #6
C++ supports virtual multiple inheritance to resolve #1, #2 and #5 (see [3])
Multiple Inheritance – Modeling Issues

- A Pegasus is a Horse with Wings that can Fly
- Birds have wings and can Fly
- Thus `class Pegasus : Horse, Bird`

... but Birds LayEggs() and pegasi do not !!!

Major Modeling Issue with Multiple Inheritance:
We can reuse code by inheriting from a class ('is a'), but it comes bundled with all features of that class

Perhaps we need smaller units of code reuse and a different way of adding them to classes ...
1. A Class may **extend** a single Class (parent) and implement zero or more Interfaces
2. An Interface may **extend** one or more Interfaces via Inheritance
3. An Interface **may not contain State Data**
4. An Interface **cannot be instantiated** (has no constructor)
5. An Interface may only contain Abstract Methods (no implementation) and Constants
6. A Class has to implement all methods of all Interfaces (a blow to productivity)
7. Unrelated classes may implement the same Interface (enables Subtype Polymorphism)
8. Java 8 (2014) allows adding default method implementations to interfaces

```java
interface canFly {
    public void Takeoff (float runway); // meters
    public void Fly (float speed); // meters/sec
    public void Glide (float distance); // meters
    public void Land (float runway); // meters
}
```
Traits

Back in 2003 Stephane Ducasse et al. [8] conducted a comparative analysis of using multiple inheritance, interfaces and mixin inheritance for designing reusable code and proposed a concept of ‘lightweight composable units of code’ they called Traits.

The authors found that classes play two often conflicting roles:

• Primary role – generator of instances (classes have to be complete)
• Secondary role – units of code reuse (classes have to be small)

Moreover, while each class has a unique position in class hierarchy, ‘adding’ reusable code should not be constraint by class location in class hierarchy.

• Trait implements a set of reusable methods
• Trait may require a set of ‘dependency’ methods.
• Traits provide ‘structure and reusability within a class definition’ ([8], p.8)
• Traits behave like smart INCLUDE statements resolved by compiler at compile time
A Class may have a single Superclass (single inheritance)
A Class may be **composed** of multiple Traits
A Trait may be **composed** of multiple Traits
Trait composition enjoys **flattening property**: ‘the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the traits’ ([8], p.9)
Class semantics is not affected by Trait **order** in Class declaration
Trait has a **Type**, enabling Subtype Polymorphism
Traits do not have **State**
Traits cannot **directly** access class State, relying on class accessor methods instead (**glue code**)

[Diagram showing relationships between Superclass, Class, Trait, and multiple Traits]
Example: Factoring Out Generic Wrappers

Multiple Inheritance:
Factor Out Reusable Code into a Separate Class

Does Not Work !!!
Traits: A New Approach to Designing Reusable Code

Example: Factoring Out Generic Wrappers

Class A
read
write

Class SyncRW
sync_Read
sync_Write
acquire_Lock
release_Lock
direct_Read
direct_Write

Class B
read
write

Class SyncA
direct_Read
direct_Write
read
write

Class SyncB
direct_Read
direct_Write
read
write

Trait SyncRW
read
write
acquire_Lock
release_Lock

Class SyncA
write
acquire_Lock
release_Lock

Class SyncB
write
acquire_Lock
release_Lock

SyncB extends B with SyncRW
Traits: no boilerplate code required
Multiple Inheritance: 4 extra methods per SyncA/B class
Programming Languages Supporting Traits

Traits are implemented in more than 20 programming languages. Some implementations are done via runtime libraries, while others as a native language construct [10]. The latter include:

C++ (via Standard Template Library)
Groovy
Kotlin
Perl (roles)
PHP (inheritable templates)
Python (via higher-order mixins)
Ruby (Module Mixins)
Rust
Scala (Traits)
Smalltalk (Squeak and Pharo dialects)
Swift (via protocol extensions)
Traits in Scala

1. Scala Traits may have State
2. Trait state data may be directly accessed from Traits & Classes having that Trait
3. A Trait can restrict which Traits & Classes may be composed using the Trait
4. Trait/Class semantics depends on order of Traits composing the Trait/Class
5. Scala uses mixin-like Linearization to resolve method name conflicts (enables Stackable Traits Design Pattern)
Traits are SUPERIOR to Multiple Inheritance for modeling real-life systems

Modeling with Traits

Horse extends Animal with Legs with canRun ;
Pegasus extends Horse with Wings with canFly ;

Bird extends Animal with Legs with Wings ;
Ostrich extends Bird with canRun ;
Swan extends Bird with canFly
Interface Solution Selection Criteria

<table>
<thead>
<tr>
<th>Criteria (descending priority)</th>
<th>Single Inheritance</th>
<th>Multiple Inheritance</th>
<th>Interfaces Java/C#</th>
<th>Traits (Ducasse)</th>
<th>Traits (Scala)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling Power</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Conceptual Simplicity</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Better for Code Reuse</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Implementation Cost</td>
<td>1</td>
<td>4</td>
<td>?</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Priority: High

Solution Ranking:

1. **Modeling Power:** How good is it for modeling real-life systems
2. **Conceptual Simplicity:** Is it intuitive and easy to learn/use
3. **Better for Code Reuse:** How it affects code reusability
4. **Implementation Cost:** Implementation complexity and extra compiler overhead
LabVIEW Traits Proposal

This proposal is based on Traits as described by Ducasse et al. in [8] with a few features borrowed from Scala

• Class is a ‘generator of instances’ (objects)
• Trait is a ‘fine-grained unit of code reuse’
• Traits provide ‘structure and reusability within a class definition’ ([8], p.8)
• Traits behave like smart INCLUDE statements resolved by compiler at compile time (AKA ‘Include Statement Metaphor’)
• Traits (like Interfaces) enable Class Subtype Polymorphism
• Traits (unlike Interfaces) may have a single implementation
Classes and Traits

- A Class may have a **single** Superclass (single inheritance)
- A Class may be **composed** of multiple Traits
- A Trait may be **composed** of multiple Traits
- Trait composition enjoys **flattening property**: ‘the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the traits’ ([8], p.9)
- Class semantics is not affected by order of Traits in Class declaration (deviation from Scala Traits)
- Trait has a **Type** (enables Subtype Polymorphism)
Conflict Resolution

- **Method Name Conflict Resolution:**
  - Class methods replace Trait methods
  - Trait methods override Superclass methods
  - Peer Trait method conflict must be explicitly resolved via aliasing or method exclusion (see [8], p.11)

- A **single copy** of Trait methods is added to a Class (Trait Hierarchy may have multiple occurrences of same Trait)
- A Trait cannot **directly** access Class state data

- A Trait may **restrict** set of classes/traits it can be added to
  - by requesting composing class/trait Type[s]
  - by requesting a set of ‘required’ Methods
Traits May Have State

- Trait state data is **private** to the Trait (similar to Class state data)
- Trait may provide **accessor methods** for its state data
- A **single copy** of Trait state data is added to Class by compiler (as with C++ Virtual Inheritance)
- Stateful Traits ensure Data Encapsulation
- Stateless Traits (with the Class hosting Trait State) result in **Leaky Abstractions**
- Semantics of Stateful and Stateless Traits is the same – sans Leaky Abstractions ...
- Unlike Scala, **super** has no special meaning in Trait Composition (simply calls superclass methods)
Traits Eliminate the Diamond Problem

- Single Inheritance prevents Diamonds
- Trait Composition may create a Diamond
- Trait method name conflicts are explicitly resolved with Aliasing or Exclusion
- Flattening Property removes duplicate Trait entries (results in a single copy of Trait X State)
- Traits Y & Z manipulate same copy of Trait X State (risking side effects)

Class B

Trait_Z extends Trait_X
Trait_Y extends Trait_X
Class_B extends Class_A with Trait_Z with Trait_Y
LabVIEW Project **Structure View**

- **Project.Items** View should be used for editing class/trait definition
- New **Project.Structure** View includes ‘view only’ section of virtual folders with all trait and superclass methods
- **Project.Structure** View is to be used for
  - aliasing conflicting Trait names
  - excluding conflicting Trait Names
  - altering trait method access scope
- Overwritten methods appear as grayed out in **Project.Structure** View
- Classes *share* a single copy of Trait method definitions (enables code reuse)
- Traits cannot be stored in binary form (.lvlibp)
Working Solution
... finally!

But have to live with
ISP violation...
Unit Test Design with Traits

ArT_Valid_State

ArT_Unique_Name

ArT_Unique_Name_UT

ArT_Unit_Test

ArT_Error

ArT_Host_Error

ArT_Your_Error

ArT_Class

ArT_Class_UT

ArT_Your_Class

ArT_Your_Class_UT

Unit_Test_Harness.vi

Calls

Has a ...
Call

Has a ...
Call

Extends

Extends

Extends

Extends

With

With

With

With

With

With
References & Resources

Code Reuse

[3] C++ FAQ: Multiple and Virtual Inheritance
[6] AZ Interface Toolkit, Andrei Zagorodni, 2018
[7] Class Translation Mechanism, Piotr Kruczkowski, 2018

Traits

[11] LabVIEW, Interfaces & Traits, Dmitry Sagatelyan, CLA Summit 2016, Austin
[12] Scalable Component Abstractions, Martin Odersky, Matthias Zenger, OOPSLA’05, San Diego, 2005
Before you go, take the survey.
Discussion