AZ Interface

version 2.0.0

Andrei Zagorodni

2018-11-03

Content

1 Introduction

AZ Interface (*AZI*) is tool and solution for implementing Java-like interface architecture in LabVIEW projects.

Contrary to other solutions providing Java-like interface architecture, **AZ Interface** is simple while fulfilling basic programming demands.

1.1 Versions

Version number consists of four values:

1. *version* - altered with major changes causing compatibility and/or conceptual issues;
2. *subversion* – altered with introduction of major changes;
3. *fix* – minor changes, f. ex. a bug fix or minor performance improvement;
4. *build* – has meaning only for developer; f. ex. allows accounting of development packages, special assemblies, etc.

Altered *version* or *subversion* can cause a need in reading updated manual, while altered *fix* or *build* does not affect the way of use.

1.1.1 Version 0.0.0-pre-alpha

First functional version of the toolkit.

The version was presented at European CLA summit in Madrid, 2018.

1.1.2 Version 1.0.0-alpha

The version basically differs from v.0.0.0.

This version is result of brainstorming at European CLA Summit 2018:

- first, the concept was presented as a regular lecture;
- second, pitfalls were extensively discussed/brainstormed with Stephen Loftus-Mercer, National Instruments.
- third, the lecture was repeated and many other experts participated in brainstorming.

I highly appreciate contribution of all participants of these sessions /Andrei Zagorodni

1.1.2.1 Release 1.0.0.0

First public release of **AZ Interface** software.

1.1.2.2 Release 1.0.0.1

Public release including few small fixes.

Main fix: Improved HD folder selection algorithm for newly created **AZ Interface**.

1.1.3 Version 1.1.0-alpha

Reentrant execution of *AZI methods* is implemented. The reentrancy is "limited", see section 5.6.

1.1.4 Version 2.0.0-beta

Relationships between interfaces and interface-applying classes are altered. Each class “knows” about applied interfaces while interfaces do not “know” about implementing classes.

Upgrading from version 1 to version 2 is described in section 4.4.

1.2 Conventions

1.2.1 Shortenings and abbreviators

Abbreviator	Description
AZI	AZ Interface
`[aziName]`	Any name of AZI
BD	Block Diagram
HD	Hard Disk
`[LabVIEW]`	Location of LabVIEW in this computer; for example `C:\Program Files (x86)\National Instruments\LabVIEW 2016\`

Abbreviator	Description
`[methodName]`	Any name of a method
OOP	Object-Oriented Programming
SW	Software; AZ Interface software

1.2.2 Font conventions

- **Bold** is used for anything that appears literally in a LabVIEW environment or in LabVIEW program. For example, for menu, labels that cannot be altered.
- *Italic* шы used for terms.
- `Constant Width` is used for values: paths, names, etc.
- `[ ]` – brackets surround selectable values.
- *Green Italic* is used for private notes.

1.3 Concept of Interface in other languages

Concept of Interface was developed to substitute multiple inheritance in some *Object Oriented languages* (*OOP* languages). Probably the most known of them is Java.

Similarly to LabVIEW, a Java *class* can have only one parent *class*; i.e. class hierarchies have tree-like structures. *Java Interface* allows creating "cross-links" between trees; i.e. simulate multi-parent behavior. The concept is illustrated by

Figure 1.

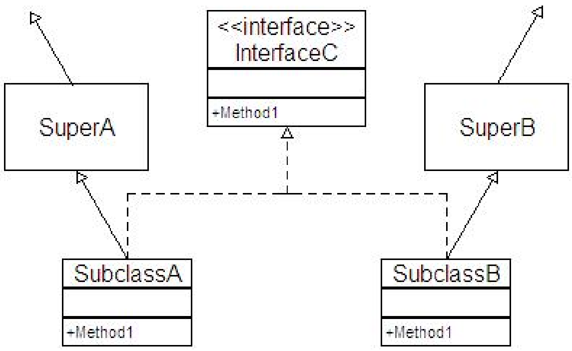

Figure 1 Interface in UML diagram

`SubclassA` and `SubclassB` belong to different hierarchical trees. `InterfaceC` provides common behavior to these *classes* with no effect on hierarchical positions of classes `SuperA` and `SuperB`.

Java Interface provides an own data type allowing to work at corresponding abstraction layer.

Java Interface can be considered as an *Abstract Class* having only abstract methods. Attributes are not allowed in *Interfaces*. Otherwise `InterfaceC` behaves exactly in the same way as any super-class.

2 AZ Interface Background Ideas

2.1 Solutions

AZ Interface (*AZI*) utilizes capacity of *Call By Reference* node.

Each *AZI* is assembled as a *native LabVIEW class*. No class hierarchies are allowed between *AZI*-s.

Relation between *AZI* and *Class* applying the interface is defined by adding the *AZI* in list of ***Friends*** (***Community scope***) of the *Class*.

2.2 Features

AZI-s allow creating abstraction levels independent on hierarchical structures of classes.

AZI-s allow abstraction of functionality independently on implemented OOP model; i.e. same methods of the same *AZI* can be applied in *native LabVIEW classes*, GOOP3 classes, GOOP4 classes, and G# classes.

LabVIEW code created with toolkit can be opened, edited and run without installation of the toolkit. The code is not limited to LabVIEW development environment; corresponding EXE-files can be run under conventional LabVIEW RTE. However, developer must take care about inclusion of invoked code in build specification (that is the same for any LabVIEW code invoked with *Call By Reference* node).

2.3 Limitations

- The code is not imperative; for example inconsistency between terminal patterns is not shown in **Error list** window.
- No hierarchy between *AZ Interfaces* can be established.
- Current version is tested only for *My Computer* branch of *LabVIEW Project*. Use of the toolkit with other targets was not tested yet. This limitation will be resolved in future.
- **AZ Interface Consistency tool** announced for v.0.0.0 is not included in following versions. Need in functions of this tool disappeared due to altering of the whole concept. New tool will be created in future if new needs will be identified.
- Connector pane of *AZI methods* use terminal pattern 6x4x4x6 only. Altering the terminal pattern would cause errors that are difficult to fix.

- Connector pane terminals of each *AZI method* must be assigned before the method is applied in one of *classes*. Later changes could require significant efforts. *I am still thinking how to do such operations easier.*

3 System Requirements and Installation

3.1 Requirements

Current version of the toolkit is developed for LabVIEW 2016 and expected to be fully functional with following versions of LabVIEW.

No additional package is required.

Ask me if you need the toolkit for an earlier version of LabVIEW. I can downgrade the code.

3.2 Installation

No installer is supplied with current version of the toolkit. Files must be manually copied in corresponding LabVIEW directories.

3.2.1 File location

Files must be copied in different directories of LabVIEW. The table below refers to `[LabVIEW]` directory that, for example to,

`C:\Program Files (x86)\National Instruments\LabVIEW 2016\`

Content of the following source directories must be copied into corresponding target directories.

Supplied files	**Target LabVIEW directory**
`GProviders`	`[LabVIEW]\resource\Framework\Providers\GProviders\`
`Providers`	`[LabVIEW]\resource\Framework\Providers\`
`help`	`[LabVIEW]\help\`

3.2.2 Recompiling

In some cases files of the toolkit must be recompiled after the copying; f. ex. VIs must be re-saved accounting to new locations of sub-VIs.

To do it open consequently two VIs. These VIs are used only for manual installation. Ignore messages concerning altered file locations. Order of opening could be important:

1. Open LabVIEW.
2. Open `LabVIEW]\help\AZ Interfaces\_1_all_help_AZ_Interfaces.vi`
3. Open `LabVIEW]\resource\Framework\Providers\AZ_Interfaces\_3_all_providers_AZ_Interfaces.vi`
4. Click menu **File** > **Save All**.
5. Close all VI-s.
6. Restart LabVIEW.

4 Primary Functions of the Toolkit

Note: when working with *AZI* and *AZI*-applying *Class* all involved files must not be write-protected. Remove write-protection from the *AZI* , the *Class*, and all their members.

4.1 Creating AZI

1. Right-click the **My Computer** or any **Virtual Folder** and select menu **AZ Interfaces** > **Create AZ Interface**.
2. **Create Interface** dialog will be opened.
3. Write name of new *AZI class*, use other input fields if needed.
4. Click **Create Interface**.

Pink background indicates invalid input value; f. ex. invalid name, name in use, etc.

LabVIEW class will be created in selected location. Newly created *AZI* includes three members:

- `cast_to_Interface.vi` – utility method called only by automatically created methods of *AZI*-applying *classes*.
- `method_refs.ctl` – utility type definition that is part of *AZI* private data.
    - The type definition is also used in automatically created methods of *AZI*-applying *classes*.
- `read_Object.vi` – method is used for back-casting from *AZI* data type to data type of particular class.
    - The method should usually be followed with node *To More Specific Class*.
    - The method optionally destroys instance of *AZI* (but not instance of the class), see section 5.5.3.

4.2 Creating AZI method

1. Right-click the *AZI* class in LabVIEW project and select menu **AZ Interfaces** > **Create Interface method**.
2. Write name of the method in the opened dialog and click **Create method** button.
3. Open *Front Panel* of the newly created method.

4. Create controls and indicators and connect them to *terminal pattern* of the VI.
    - Do not select another *terminal pattern*; only 6x4x4x6 pattern is supported.
    - Do not disconnect existing *terminals*.
    - ATTENTION: altering *terminals* (number of *terminals*, they assigning in *terminal pattern*, data types) after overriding the method in *AZI*-applying *class*(es) will cause a need in extensive manual work (see section 5.3). Thus be careful at this step.
5. Do not edit *Block Diagram* of the *method*.
6. Optionally alter *icon* of the *method*, these changes will propagate in *icons* of corresponding *methods* in *AZI*-applying *classes*.
7. Save the method.
8. Save the whole *AZI* (Select class in the project then right-click menu **Save > Save All (this Class)** or select menu **File > Save All**).

Block Diagram (*BD)* of the newly created method (see Figure 2) contains default code and terminals of user-created controls/indicators. This *BD* will be automatically altered when the method is first applied in any *AZI*-applying *class* (see section 4.3.2).

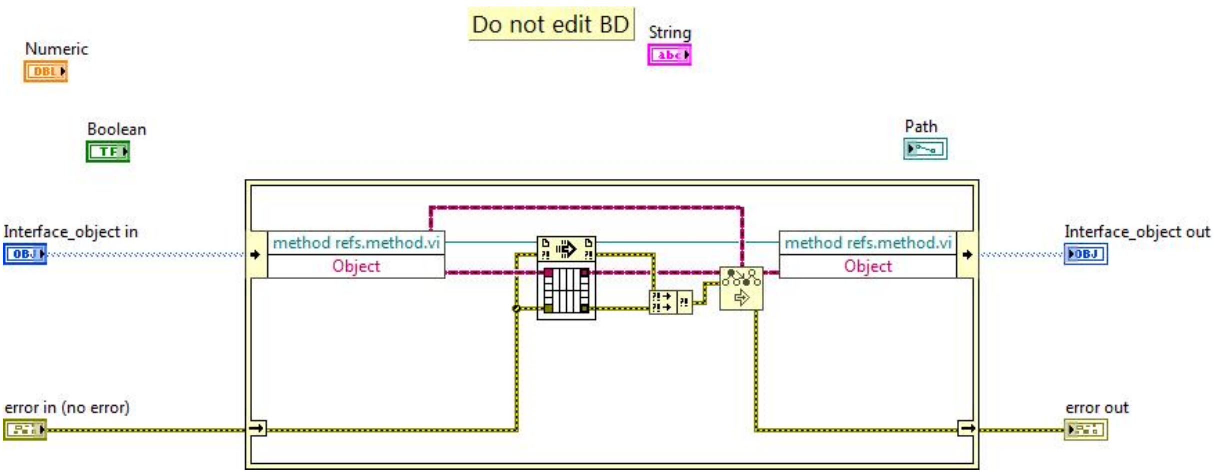

Figure 2 Example of newly created *AZI method*

4.3 Applying AZI and AZI methods to Class

The same dialog is used for applying *AZI* to *Class* and for implementing *AZI Method* in the *Class*.

1. Right-click any class in the project then select menu **AZ Interfaces** > **Apply Interface**.

2. The dialog appears listing all available *AZI*-s (Figure 3). List **interface methods** is populated with methods of *AZI* selected in list **Interfaces**.

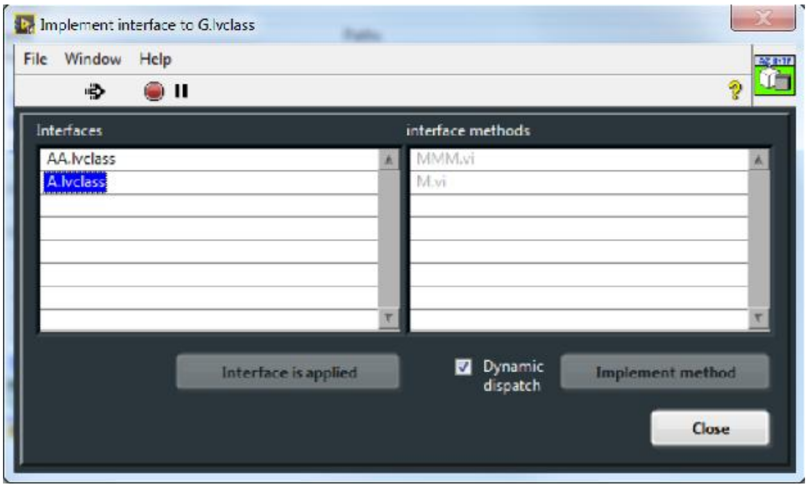

Figure 3 GUI used for applying *AZI* to *Class*.

4.3.1 Applying AZI to Class

1. Select an item from list **Interfaces**. The list shows all *AZI*-s available in the *Project*.
   - If selected *AZI* is already applied to the *Class*, button at bottom of the list is disabled exposing text "Interface is applied" (see Figure 3). In this case select another *AZI*, continue working with methods (section 4.3.2), or click **Close**.
2. Click button **Apply interface**.

Applying *AZI* to a *Class* results in:

- The *Class* is added in *AZI* lists of *Friends*.
- The *AZI* is added in *Class* lists of *Friends*.
- New method is added to the *Class*:
  - o The method is named `cast_to_[aziName].vi`, where `[aziName]` is name of the *AZI*.
  - o This method is used for casting of corresponding *Object* to type of the *AZI*. In some sense the casting is similar to one performed by nodes **To More Specific Class** and **To More Generic Class**.
  - o The method `cast_to_[aziName].vi` is initially broken. It will be repaired automatically (its *Block Diagram* altered) when any *AZI method* is applied in the *Class* (see section 4.3.2).

4.3.2 Implementing AZI methods in Class

1. Select an item from *AZI* list **Interfaces** (see Figure 3).
   - List **interface methods** at right-hand shows *methods* available in this *AZI*.
   - If selected *AZI* is not yet applied to the *Class*, button **Apply interface** at bottom of the list is enabled. In this case click button **Apply interface**, select another *AZI*, or click **Close**.
2. Select method in the list **interface methods**. Methods already applied in this *Class* are disabled.
3. Click button **Implement method**.

Applying *AZI method* to a *Class* results in:

- The method is added in the *Class*:
  - o The method has necessary terminal pattern.
  - o *Block Diagram* of the method is initially empty. All coding of the method (including wiring of class terminals) must be performed manually.
- Utility method `util_[aziName]_cls_[methodName].vi` is created:
  - o Name of the utility method contains name of the *AZI* (`[aziName]`) and name of the actual method (`[methodName]`).
  - o The utility method is created automatically and should not be altered.
- *Block Diagram* of `cast_to_[aziName].vi` method is automatically altered.
  - o The method is repaired if it was broken (the method being created is initially brocken).
- *Block Diagram* of corresponding *AZI* method is rewired if it was not done earlier.

4.4 Upgrading from v.1 to v.2

Version 2 of the toolkit cannot be used for further development of *AZI-s* created with version 1. If a project contains v.1 *AZI*, right-click will open menu **AZ Interfaces** > **Upgrade Interface to v2**.

Upgrading v.1. to v.2 does the following:

- Sets access scope of *AZI method* **cast_to_Interface.vi** to be *Public*.
- Empties *AZI's* list of *friends*.
- Sets internal property of the *AZI* to be v.2.

- Upgrade of an *AZI* does not affect code of *AZI*-applying classes. However, some members of these classes could be recompiled at next run.

Note: There is no tool to upgrade *AZI-s* created in version 0. This is because v.0 was not released to LabVIEW community.

5 How to Use

5.1 General example

Use of *AZI*-s can be illustrated by *Block Diagram* presented in Figure 4. Three classes are not hierarchically related while all three apply the same *AZI*.

Objects belonging to three different OOP models are created (GOOP, G#, and Native LVClass) then processed at common abstraction level of the *AZI*. Finally, the objects are cast back to initial class types.

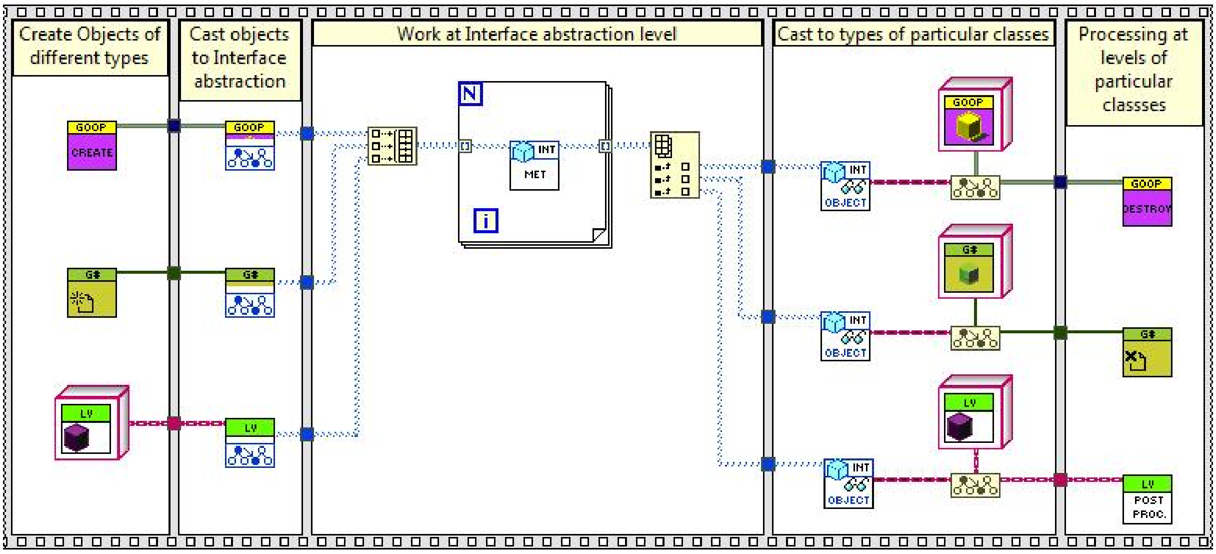

Figure 4 Example of *AZI* use.

5.2 Working with class hierarchies

5.2.1 Sub-classes of AZI-implementing class

Any child of an *AZI*-applying *class* can override *AZI methods*. There is no need to apply the same *AZI* to each sub-class of the hierarchy.

5.2.2 Two AZI-implementing classes belonging to same hierarchy

A need in applying the same *AZI* to different *classes* of the same hierarchy is rare (see section 5.2.1). *At least I cannot identify such a need.* However; this can be done changing type of object terminals of all conflicting methods to *Dynamic Dispatch*.

5.3 Altering terminals of AZI method

Connector pane terminals of each *AZI method* must be assigned before the method is applied in any *AZI*-applying *class*. However, a need in altering terminal signature could arise. Terminal signatures of the following VIs must differ only by type of object terminals:

- *AZI* method must have object terminals of *AZI* type.
- Corresponding class methods must have object terminals of corresponding class type.
- Utility methods `util_[aziName]_cls_[methodName].vi` (see section 4.3.2) must have object terminals of *LabVIEW Object* type.
- Corresponding element (*VI Refnum*) of `method_refs.ctl` (see section 4.1) belonging to the *AZI* must have object terminals of *LabVIEW Object* type; i.e. the element must have the same signature as utility method `util_[aziName]_cls_[methodName].vi`.

5.4 Instances of classes and interfaces

Note: control of class and interface instancing is especially important for use of *AZI*-s, which include reentrant methods. Missing to destroy instances of such *AZI*-s result in unused refs left in memory. This is true even if reentrant methods are not invoked in particular run of program.

5.5 Concept of "limited" reentrancy

5.5.1 Instancing of a class

Instancing/creating objects of LVOOP class is illustrated in Figure 5. New object (instance) is created with class constant or when class wire is branched.

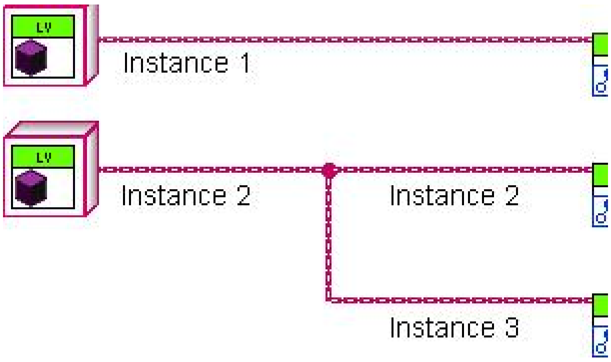

Figure 5 Instancing of a by-value class (LVOOP class)

Contrary to by-value classes, instances of by-ref classes (GOOP or G#) are created only with class constructor (see Figure 6). Forking of class wires copies only ref, which points to the same instance of the class.

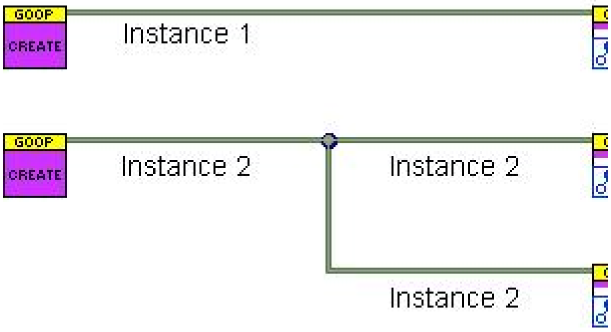

Figure 6 Instancing of a by-ref class (GOOP or G# class)

5.5.2 Instancing of an AZI

Each *AZI* instance carries wrapped object of native or by-ref object and references to methods of corresponding class. Thus *AZI* itself is by-ref solution, however behavior of particular *AZI* instance differs for different OOP models. Figure 7 illustrates instancing behavior of *AZI* wrapping by-value class while Figure 8 illustrating instancing behavior of *AZI* wrapping by-ref class.

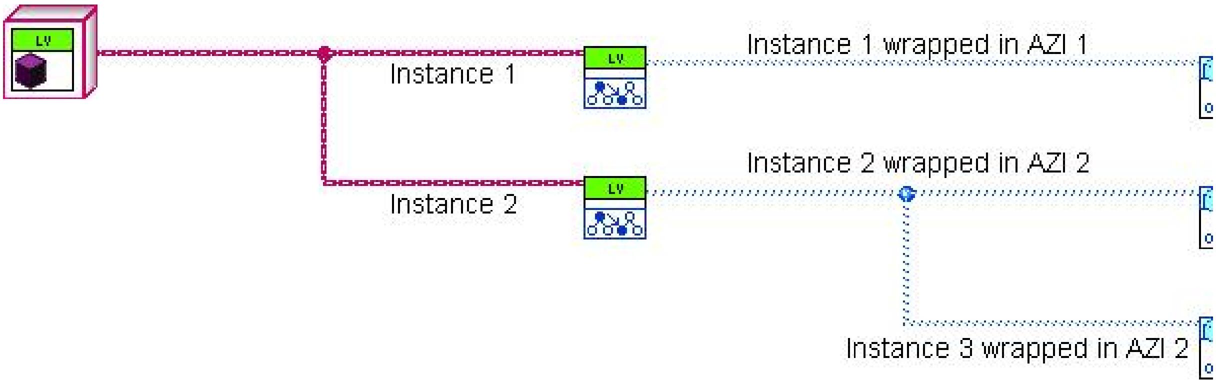

Figure 7 Instancing of a LVOOP class wrapped in AZI

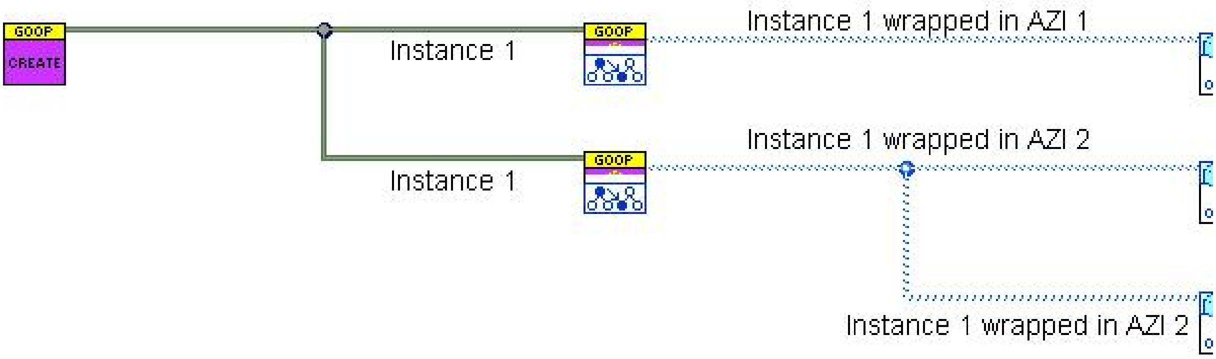

Figure 8 Instancing of a GOOP or G# class wrapped in AZI

5.5.3 Destroying instances of AZI

AZI is by-ref solution. Thus each instance of *AZI* should be destroyed when not needed any more. However, *AZI*-s having no reentrant methods carry only static references thus their destruction is meaningless.

AZI method `read_Object.vi` destroys instance of *AZI* closing dynamic VI references. However, only refs to reentrant methods are created dynamically thus only instances of such *AZI*-s must be destroyed.

Figure 9 illustrates concept of destruction.

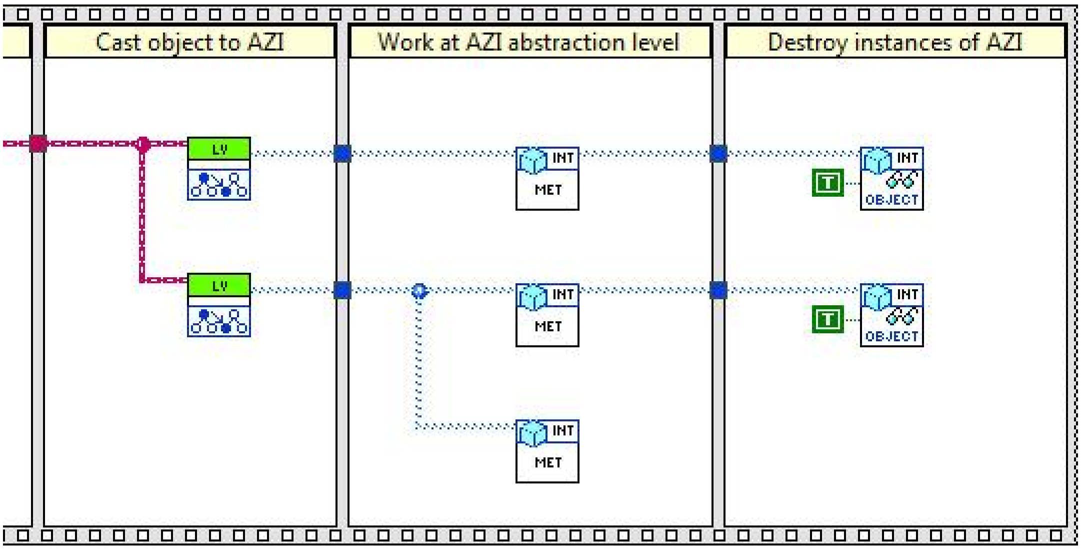

Figure 9 Concept of *AZI* destroying. Each instance created with method `cast_to_[aziName].vi` should be destroyed using method `read_Object.vi`

5.6 "Limited" reentrancy, parallel execution

Independent (parallel) execution of reentrant methods (clones) can be achieved only for different instances of *AZI*.

This can be illustrated considering behavior of reentrant method `MET.vi` as shown in

Figure 9. Three calls of the method are shown while only two can be executed in parallel. Middle and bottom calls cannot be executed simultaneously because they belong to the same instance of the *AZI*.

5.7 Programming, good programming style

5.7.1 Using read_Object.vi

Method `read_Object.vi` has two purposes:

- conversion from *AZI* data type to data type of particular class,
- destroying instance of *AZI*.

The concept is illustrated in Figure 10

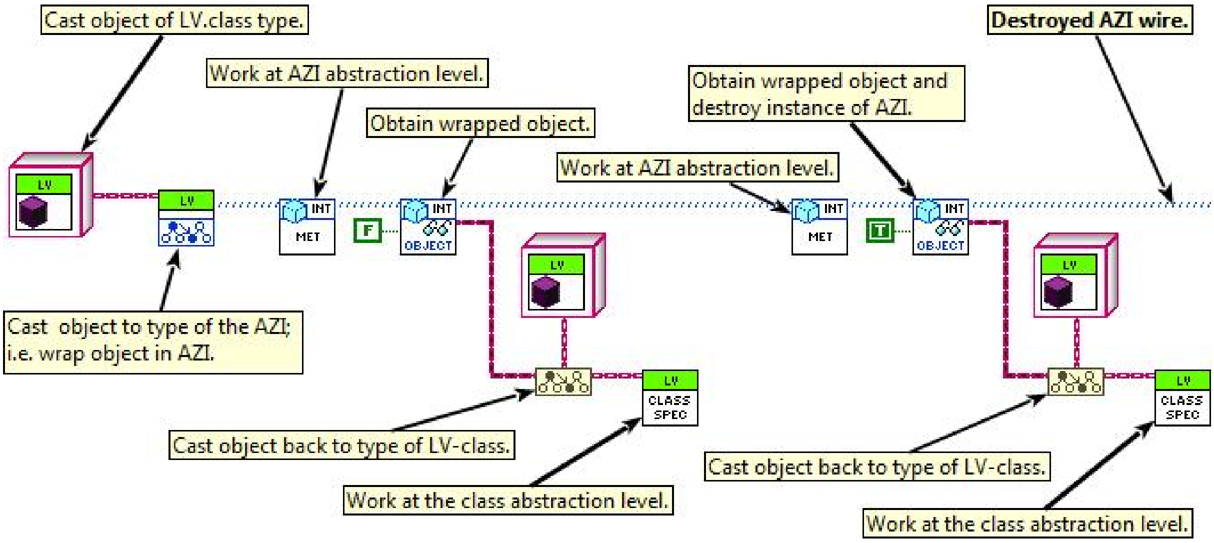

Figure 10 Using `read_Object.vi`

Please note that second call of `read_Object.vi` destroys the *AZI*. However, the *AZI* wire can be drown out of the method (`Destroyed AZI wire` in Figure 10). If the *AZI* does not include any reentrant methods, the destroying does not have any effect (and not required).

5.7.2 Race conditions

An instance of *AZI* is only a wrapper around conventional object. Thus most cases of race conditions should be resolved considering interactions between objects even if execution is performed at *AZI* abstraction level.

However, destruction of *AZI* can cause race condition as shown in Figure 11. Upper flow in this figure can be completed before execution of other flows. If `read_Object.vi` is executed before middle or/and bottom clone of `MET.vi`, these clones cannot be run.

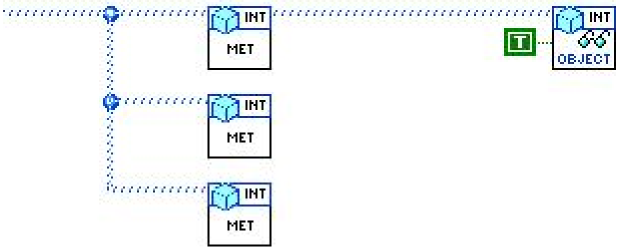

Figure 11 Race condition when destroying instance of *AZI*

The race condition can be resolved in different ways; f. ex. as shown in Figure 12.

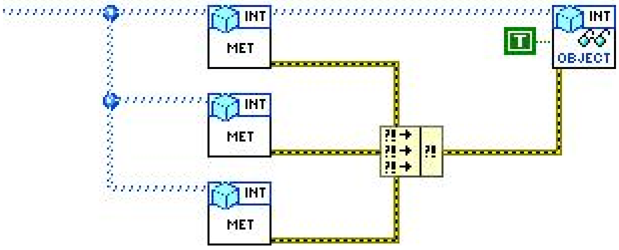

Figure 12 Resolved race condition

5.7.3 Destroying AZI – good programming style

Method `read_Object.vi` cannot launch any error. Thus the method can be used even during development of the code; i.e. when some *AZI* methods are not applied yet.

Destruction is meaningful only for *AZI* including reentrant methods. Thus only instances of such *AZI-s* must be destroyed.

However, using one call of `read_Object.vi` destructor per each call of `cast_to_[aziName].vi` can be considered as good programming style.

6 About and Contacts

Figure 13 About

6.1 License Agreement

1 Acknowledgement

CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS SOFTWARE. BY USING THIS FREEWARE VERSION YOU ACKNOWLEDGE THAT YOU HAVE READ THIS LIMITED WARRANTY, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT UNLESS YOU HAVE A DIFFERENT LICENSE AGREEMENT SIGNED BY ANDREI ZAGORODNI YOUR USE OF THIS SOFTWARE INDICATES YOUR ACCEPTANCE OF THIS LICENSE AGREEMENT AND WARRANTY. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DELETE THE SOFTWARE FROM ALL STORAGE MEDIA.

2 License

This Freeware License Agreement (the "Agreement") is a legal agreement between you ("Licensee"), the end-user, and developer of AZ Interface Toolkit Andrei Zagorodni ("Developer") for the use of this software product ("Software"). Commercial as well as non-commercial use is allowed. By using this Software or storing this program or parts of it on a computer hard drive (or other media), you agree to be bound by the terms of this Agreement. Provided that you verify that you are handling the original freeware version you are hereby licensed to make as many copies of the freeware version of this Software and documentation. You can alter this Software in any way but Developer does not carry any responsibility for consequences.

If you alter and/or further develop this Software, documentation (including "help" and "about") must include reference to original Software, name of its Developer and his contacts.

3 Limited Warranty and Disclaimer of Warranty

The AZ Interface Toolkit EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE SOFTWARE. THIS SOFTWARE AND THE ACCOMPANYING FILES ARE PROVIDED "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OF MERCHANTABILITY OR ANY OTHER WARRANTIES WHETHER EXPRESSED OR IMPLIED, OR NONINFRINGEMENT. THIS SOFTWARE IS NOT FAULT TOLERANT AND SHOULD NOT BE USED IN ANY ENVIRONMENT WHICH REQUIRES THIS. NO LIABILITY FOR DAMAGES. In no event shall AZ Interface Toolkit or its suppliers be liable for any consequential, incidental or indirect damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) resulting of the use of or inability to use this SOFTWARE EVEN IF the Software HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. The entire risk resulting of use or performance of the SOFTWARE remains with you.

4 Copyright

6.2 Contacts

Andrei Zagorodni

`andrei.zagorodni@novatorsolutions.se`

Please write **AZI** or **AZ Interfaces** in subject line.

6.3 Support and communications

I shall appreciate feedback about bugs and bottlenecks identified in this SW.

I promise to read your emails and reply within reasonable time. However the project is developed in my evenings and weekends. Thus the "reasonable time" will solely depend on my work load.

You are free to modify code of the software. However I do not promise to support the modified code.

Andrei Zagorodni