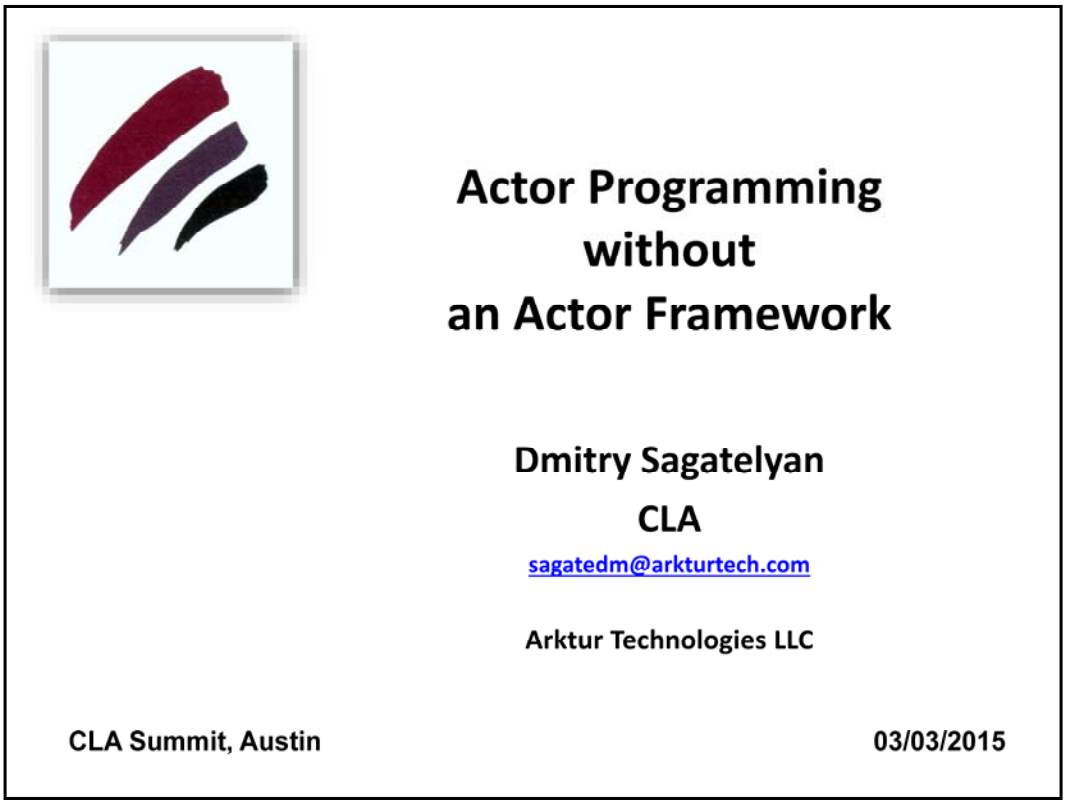


About myself:

I Come from a Computer Science & SW Engineering Background. Early interest in Procedural Programming Languages with Strict Type Checking (Pascal, Java, Eiffel, Modula-2, etc.)

Worked on system integration, data acquisition and real-time control systems @ the Academy of Sciences of the USSR for ~20 years

Established and chaired a Special Interest Group on Programming Language Modula-2 (1986 – 1991)

Switched to LabVIEW [5.0] in 1998 and am not going away from data flow any time soon

Involved in Actor Programming since 2005

Developed several frameworks in LabVIEW (see ArT_Actors White Paper https://decibel.ni.com/content/thread/13739)

Full-time LabVIEW Consultant since 2009 (Arktur Technologies LLC)

Arktur Technologies is a LabVIEW Consulting Boutique providing services in development of real-time control, data acquisition and data analysis systems with deployment options spanning PCs, Embedded Targets and FPGAs. Core areas of expertise include scalable SW architectures, R&D support, development of diagnostic software and shipping-quality end-user applications.

Presentation Outline

1. AMS Application Overview
2. Motivation
3. A few words on Actor Programming & Actor Framework
4. Agile SW Design Principles Recap
5. AMS: Actor Configuration and Lifetime Management
6. AMS: Messaging & Event Aggregator Pattern
7. AMS: Adding Actors
8. Discussion

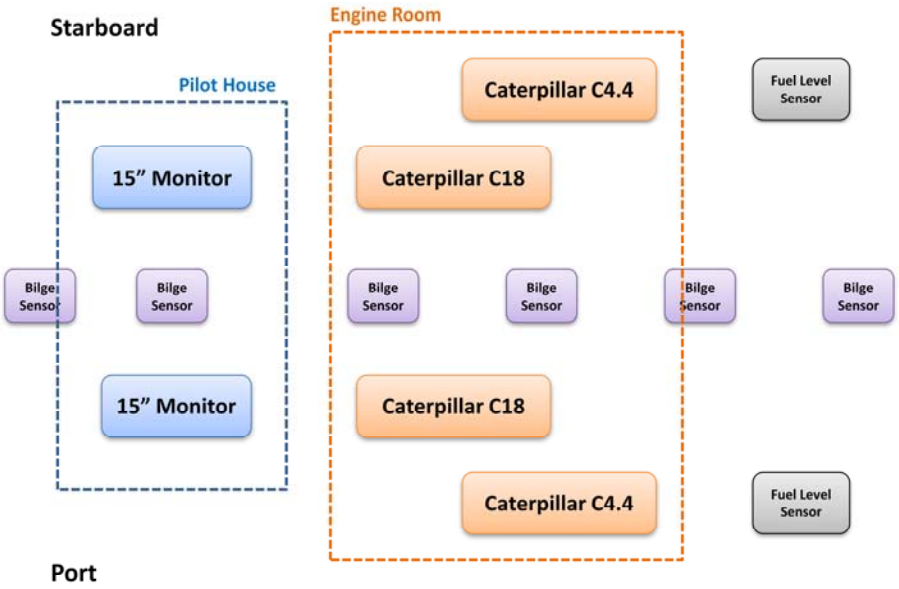
MV “Acania” Monitoring System Project

Built in 1930, 136 ft long, Steel hull, 300 tons displacement

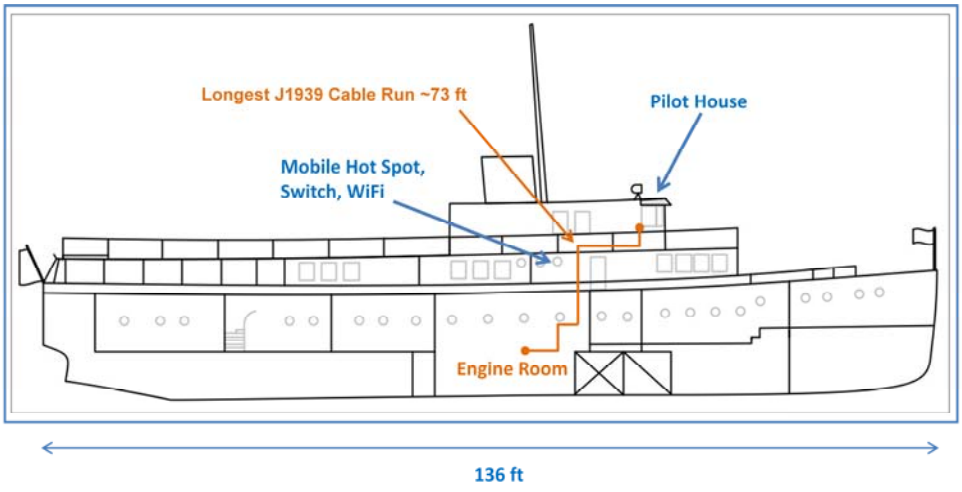
Rumored to be owned by Al Capone and used for rum running during Prohibition

Acquired by David Olson in 2008. Underwent full restoration in 2008 - 2014

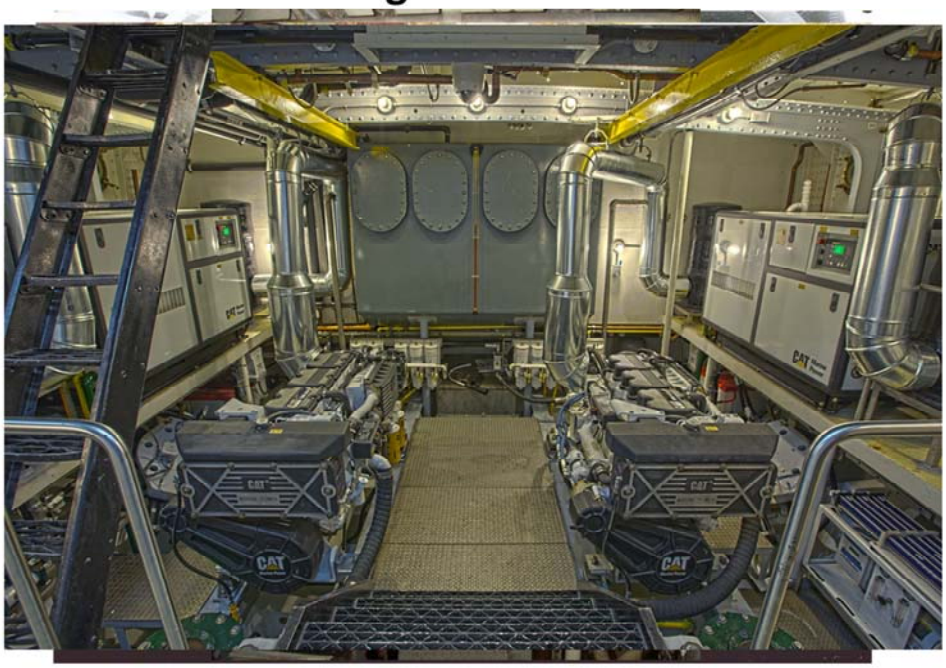
AMS Hardware



MV Acania Side View



Engine Room

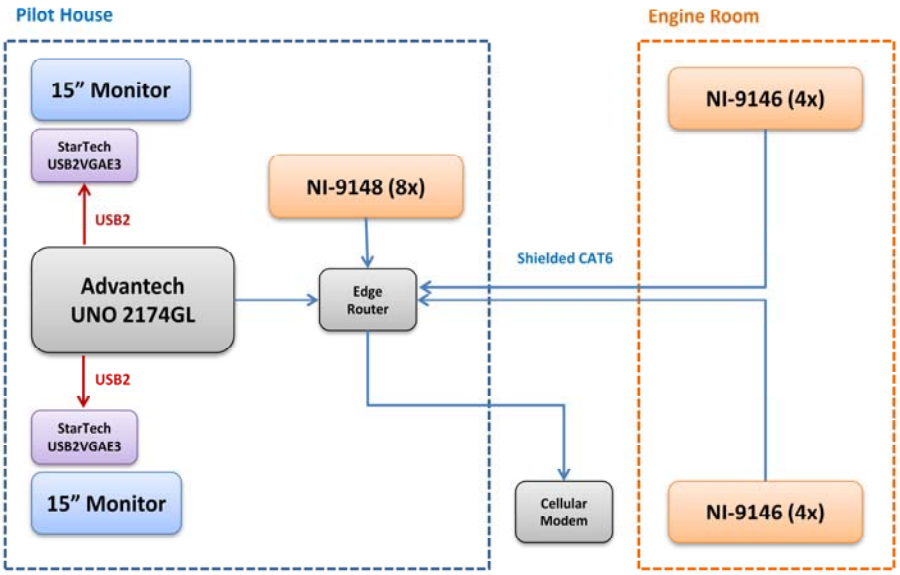


Pilot House

AMS 100% LabVIEW 'Vintage' GUI

Slide Notes

AMS Control Hardware



AMS Control Hardware

NI-9148

NI-9853

NI-9853

Advantech UNO 2174 GL

NI-9146

NI-9853

Logitech R800

StarTech
USB2VGAE3
1920x1200

NI-9146

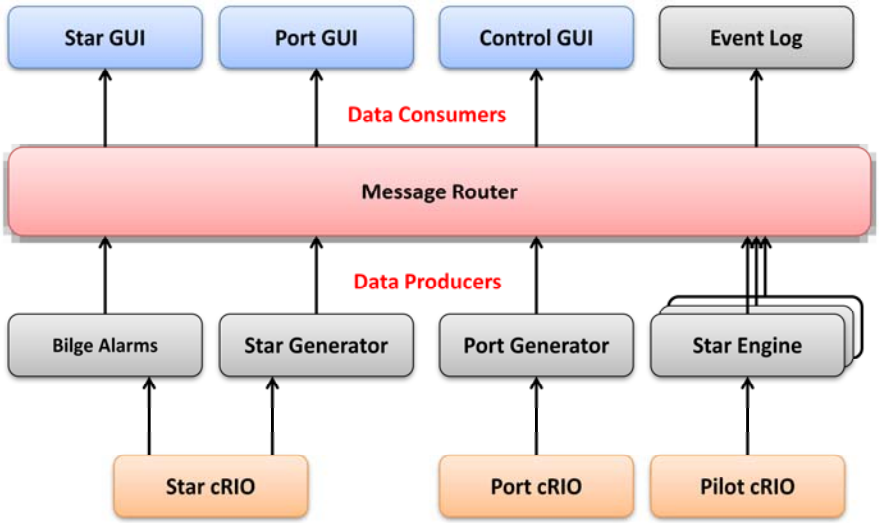
NI-9853

NI-9421

NI-9421

AMS Information Flow

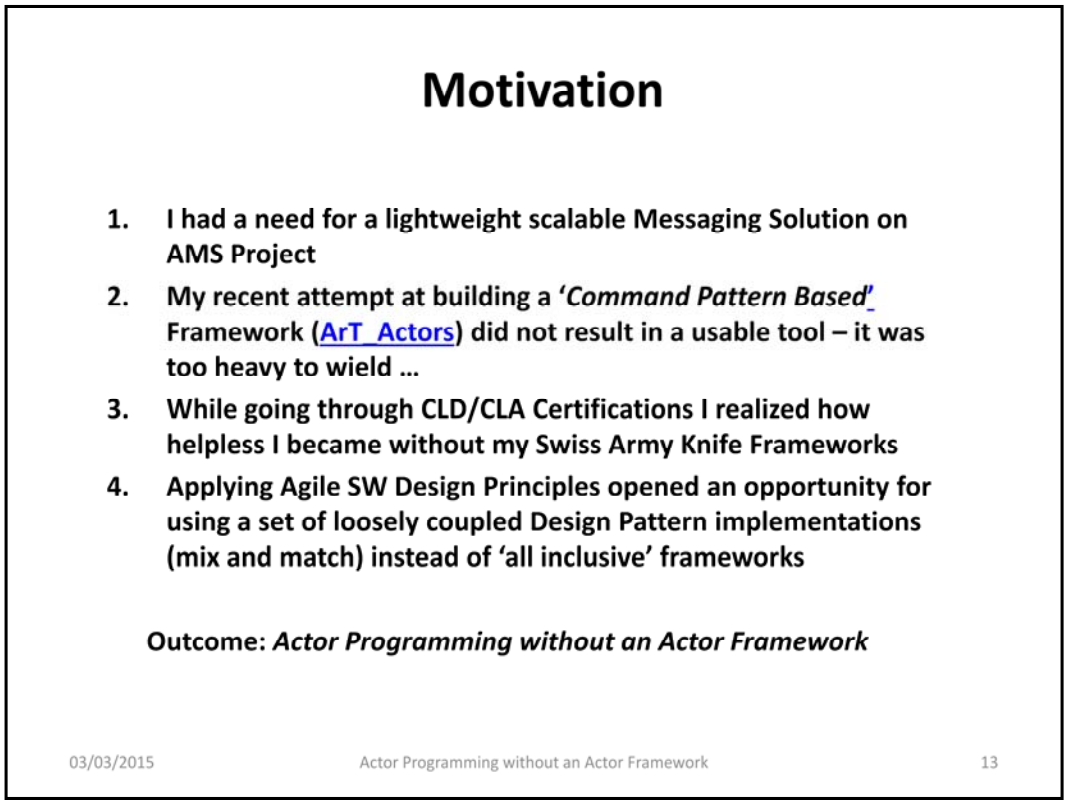
~ 120 Channels
@ 1 Hz to 50 Hz rate



AMS Application Code Summary

	Total	AMS	FPGA	Messaging	Reuse Libraries
VIs/Methods	**586**	233	57	76	274
Classes	**57**	22	8	15	20
Actor Classes	**9**	8		1	
Actor Commands	***156***	*152*		*4*	
Actor Objects	**25**	9		7 : 9	

- **N:M – # of Pub-Sub Topics : # of Actor Message Transports**
- **Comparing number of VIs/methods & classes to an AF-based Project may be misleading**
- **Using AF on AMS Project would result in 152 extra Message Classes**



I used to depend on my frameworks/tools too much in the past (was a revelation when preparing for CLD/CLA exams). "All Inclusive" frameworks help kick-starting a new app, but they also keep you locked within certain programming solutions/techniques and resist exploring new ways of doing things. I am now trying to avoid using tightly integrated frameworks in favor of loosely coupled design pattern implementations (mix & match)

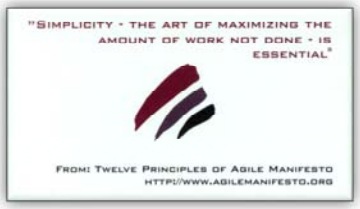
"Simplicity – the art of maximizing amount of work not done – is essential", I needed a lightweight, flexible & scalable solution to support Actor Programming.

I used to think (5-7 years ago) that actors/components should be self-contained & encapsulate/abstract from callers as many details as possible (ex: persistent settings support, subsystem instantiation, etc.). Not anymore. My current attitude – ***Dependency Inversion trumps Encapsulation***.

Dependency Inversion promotes Actor Decoupling by making all Actors depend on a Message Interface (static coupling to an Interface) – preventing static coupling of "callers" to "actors". **Dependency Injection** provides a common sense path to implement such decoupling – all Actors are instantiated and owned by Assembler class – with Actor Lifetime linked to Assembler object lifetime (by default).

I wanted to be able implementing Actors using any available Message Handling mechanisms – Queues, User Events, etc. I did not want to be forced into using Command Pattern – sometimes it does make sense, but for most Actors in my recent

On Project Goals …



1. **Meet all Requirements**
2. **Deliver on Schedule**
3. **Stay within Budget**

Agile Manifesto Principles (**Values**)

eXtreme Programming, Scrum, etc. (**Processes**)

4. **Adapt to ever changing Requirements**

A fact of life rarely emphasized in Software Design and addressed by **Agile Principles of Object-Oriented Design (Software Domain)**

Items 1-3 are addressed by Development Principles (see: Agile Manifesto) and Development Practices (Extreme Programming, Scrum, etc.) – it is all about Team Values and Processes.

Item 4 is in the SW Engineering Domain and is addressed by Agile SW Design Principles.

This presentation is about the latter

Agile Software Design Levels

- Philosophy
- Design Principles (5 + 6)
- Design Patterns (50+)
- Code

Legal Disclaimer: This is my personal view of the subject. Consume at your own risk …

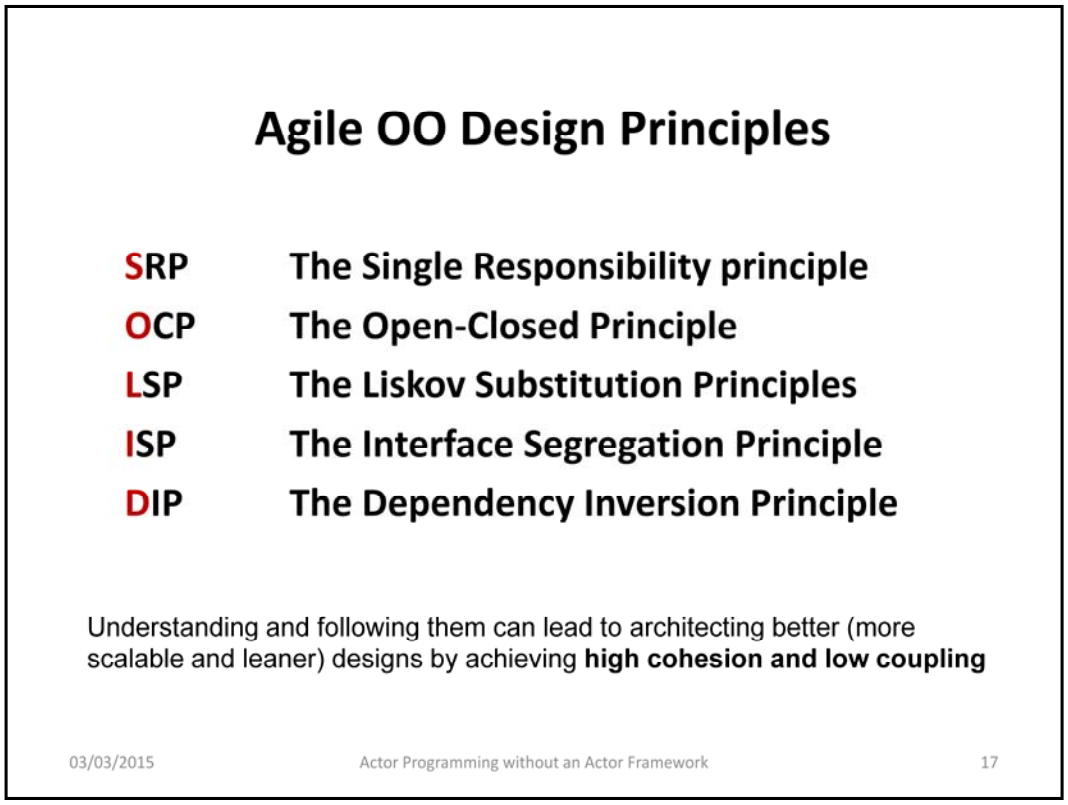
Agile Software Development Philosophy

We live in a world of changing requirements and our job is to make sure that our software can survive those changes ([1] Martin p.91)

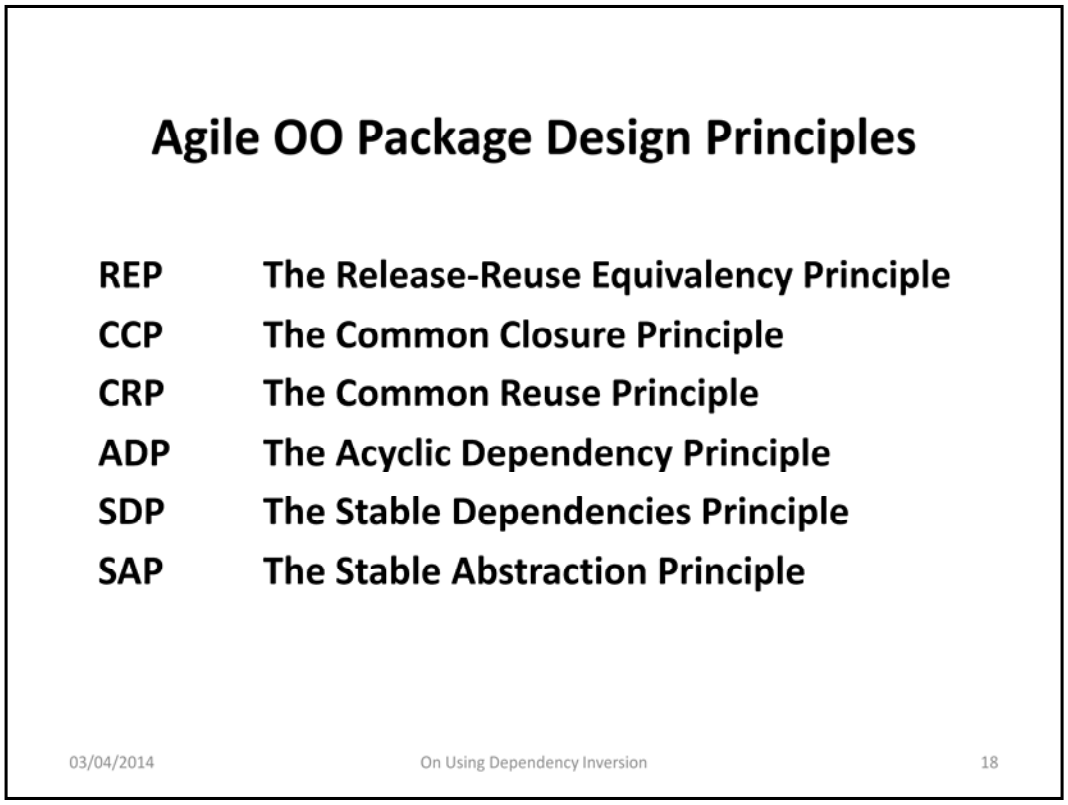
Summary:

- Make your code as good as it needs to be at the moment.
- Refactor as requirements change.
- Refactoring is an intrinsic part of Agile Development Philosophy.
- Refactoring comes at acceptable cost when the code is designed with Agile Design Principles in mind.
- **In a nutshell, Agile Philosophy is "Continuous Refactoring".**

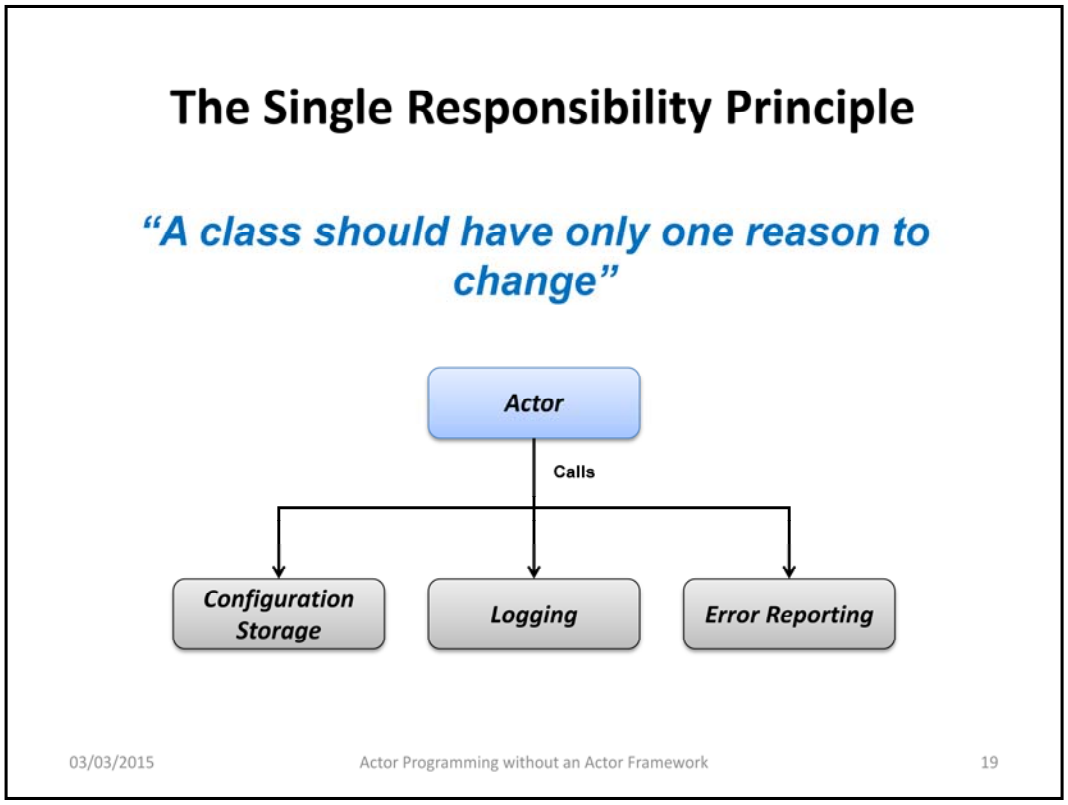
Agile Philosophy = Continuous Refactoring



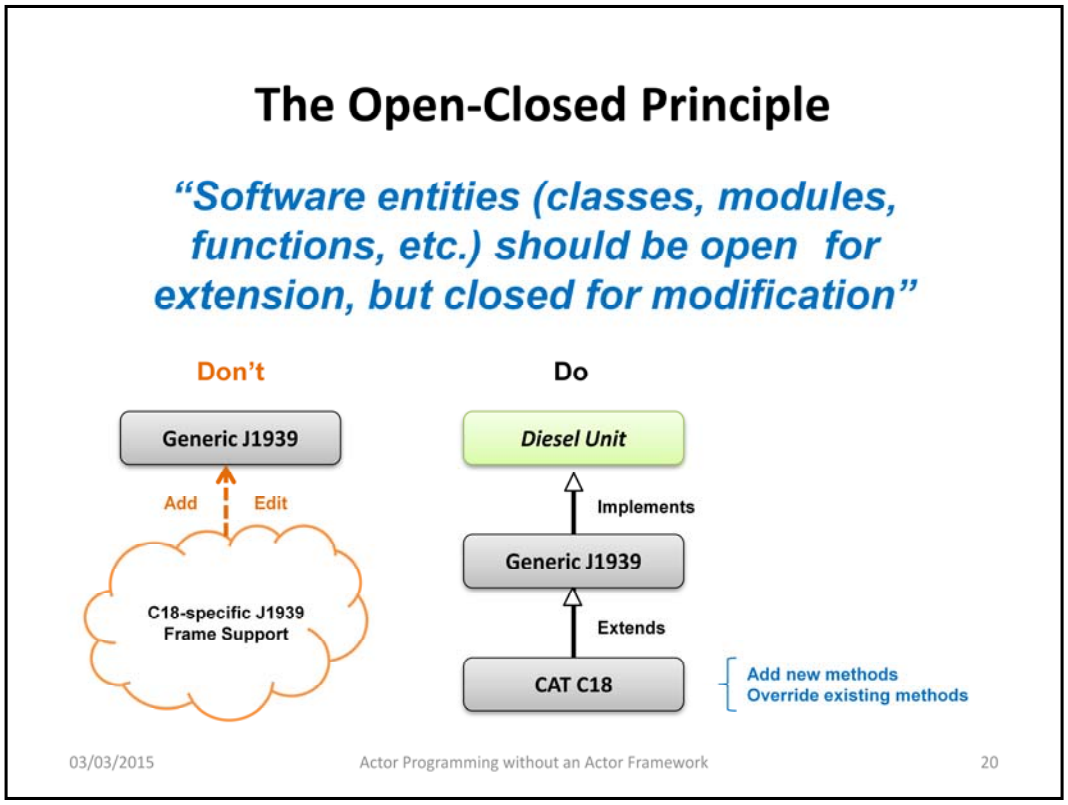
Agile Principles are guidelines – consciously breaking them for a good reason is OK. It does not take much to abuse them either … SW Design is as much art as a science after all [and team mindset provides a mold for the resulting design and code].



The Package Design Principles are not critical for general LabVIEW Development. However, they may come handy when designing/using Packed Project Libraries …



Actor class shall not implement **code** for Loading/Storing Configuration, Logging or Error Reporting. It shall delegate such actions to dedicated classes and call methods on those classes

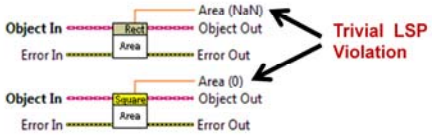


1. Example : new Requirements call for adding CAT C18 specific functionality to a Generic J1939 class. Resist urge to roll such changes into J1939 class that is already released as used in other projects. Instead, refactor your applications to have a Diesel Unit Interface and make the new CAT C18 class to extend J1939 class with new methods and override methods that handle J1939 functionality in a non-standard fashion …
2. OCP was coined by Bertrand Meyer in 1988
3. "When a single change to a program results in a cascade of changes to dependent modules, the design smells of Rigidity. The OCP advises us to refactor the system so that further changes of that kind will not cause more modifications. If OCP is applied well, then future changes of that kind are achieved by adding new code, not by changing old code that already works." ([1] p.100)
4. "**Extending** the behavior of a module does not result in changes to the **source or binary** code of the module."
5. "Since **closure** cannot be complete it must be **strategic**" ([1] p.105)
6. One's need to close a Module against "probable" changes takes common sense & experience …
7. Conforming to OCP is expensive … (Unneeded Complexity smell)
8. "How do we know which changes are likely? … **we wait until the changes happen!**" ([1] p.105)
9. "**In many ways OCP is at the heart of object-oriented design**. … it requires a dedication on the part of the developers to apply abstraction only to those parts of the program that exhibit frequent change." ([1] p.108)

Do not be zealous in applying OCP – "**Resisting premature abstraction is as important as abstraction itself**" ([1] p.109)

The Liskov Substitution Principle

"Subtypes must be substitutable for their base types "



Design by Contract (Bertrand Meyer) is a technique enforcing LSP and supported in Eiffel (**Assert** statement)

Contract is specified by declaring **preconditions** and **post-conditions** for each Method. A Child Class shall never violate Base Class contract. Otherwise using Dynamic Dispatch Methods becomes a slippery slope …

Design by Contract: supported in Eiffel (Assert statement)

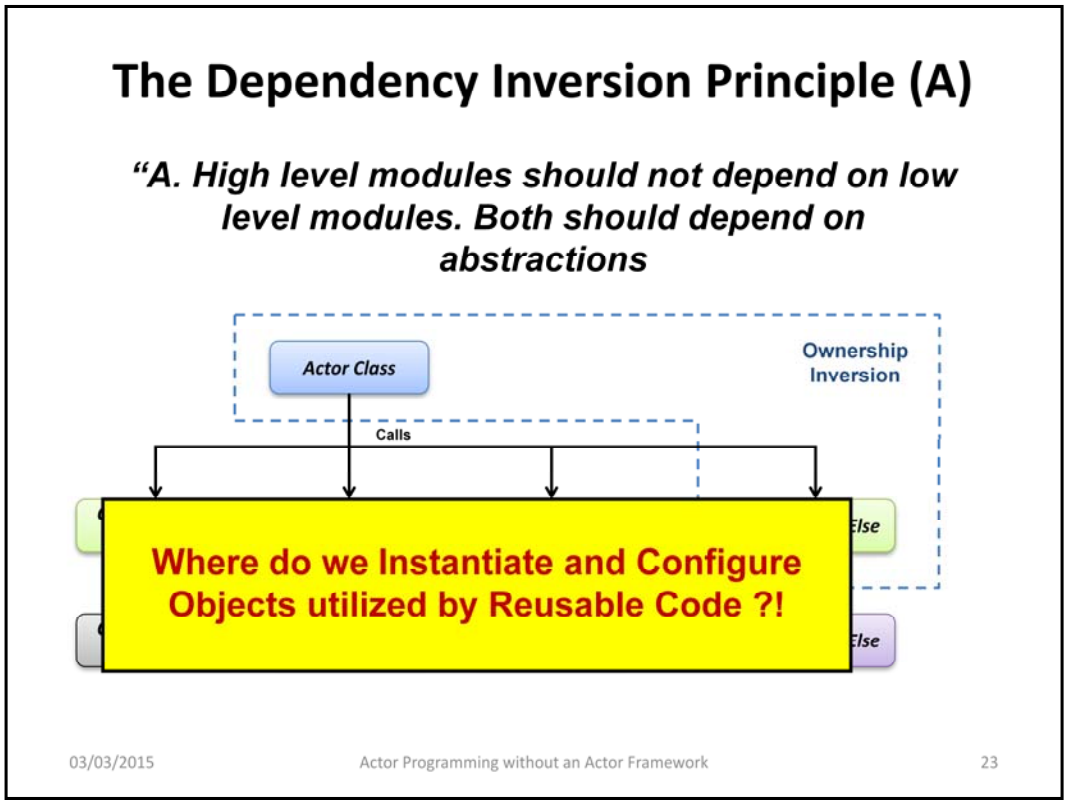
Child class methods should, at a minimum, honor Parent class Contract. Avoiding side-effects may be hard to achieve and even harder to discover

The Interface Segregation Principle

"Clients should not be forced to depend on methods that they do not use."

An Interface is considered *Fat* if it could be broken into groups of methods, where each group serves a different set of clients ([1] p.135)

"Clients of an Object do not need to access it through the interface of the object. Rather they can access it through **delegation** or through a **base class** of the object" ([1] p.138)



By substituting *Module* with *Class* and *Abstraction* with *Interface* we get – "High Level Classes should not depend on Low Level Classes. Both should depend on Interfaces", where *Depend* means calling a class method or using a class typedef … BTW, there is no *Interface* construct in the G language. I use an *Abstract* classes to implement *Interfaces*, where Abstract Class is a class with empty implementation of its Methods. To be on the safe side – such empty implementations shall return a run-time error when called directly from the code …

"Abstraction: the amplification of essential and the elimination of the irrelevant" ([1] p.194)

"Resisting premature abstraction is as important as abstraction itself" ([1] p.109)

"Abstract classes are more closely associated to their clients than to the classes that implement them" ([1] p.101)

A well known example of DIP.A is Hardware Layer Abstraction … but it also works well with pure SW entities.

The Dependency Inversion Principle (B)

"B. Abstractions should not depend on details. Details should depend on abstractions"

DIP.B provides guidance for building Interface Hierarchies

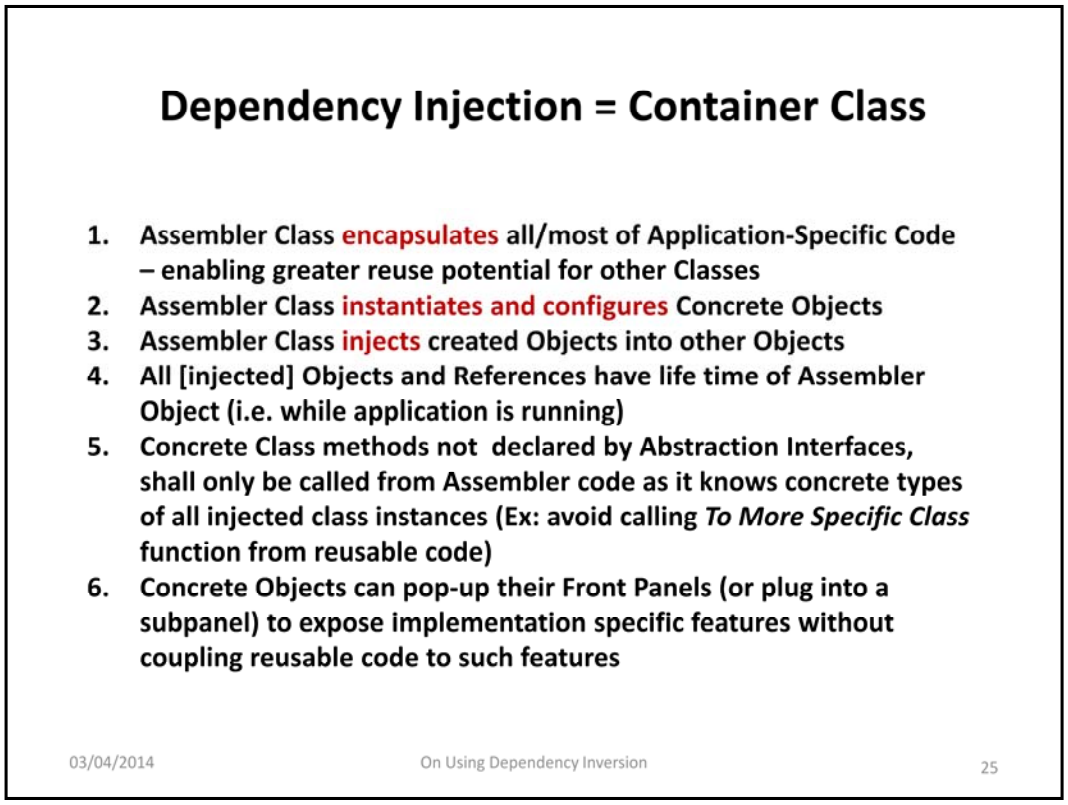
"Abstraction: the amplification of the essential and the elimination of the irrelevant" ([1] p.194)

See"HALitosis …" presentation by Norm Kirchner for real life code smells coming out of violating DIP.B

"Abstraction: the amplification of the essential and the elimination of the irrelevant" ([1] p.194)

DIP.B was the hardest to wrap my head around …

Abstraction is the Metaphor – it should not change when details change …



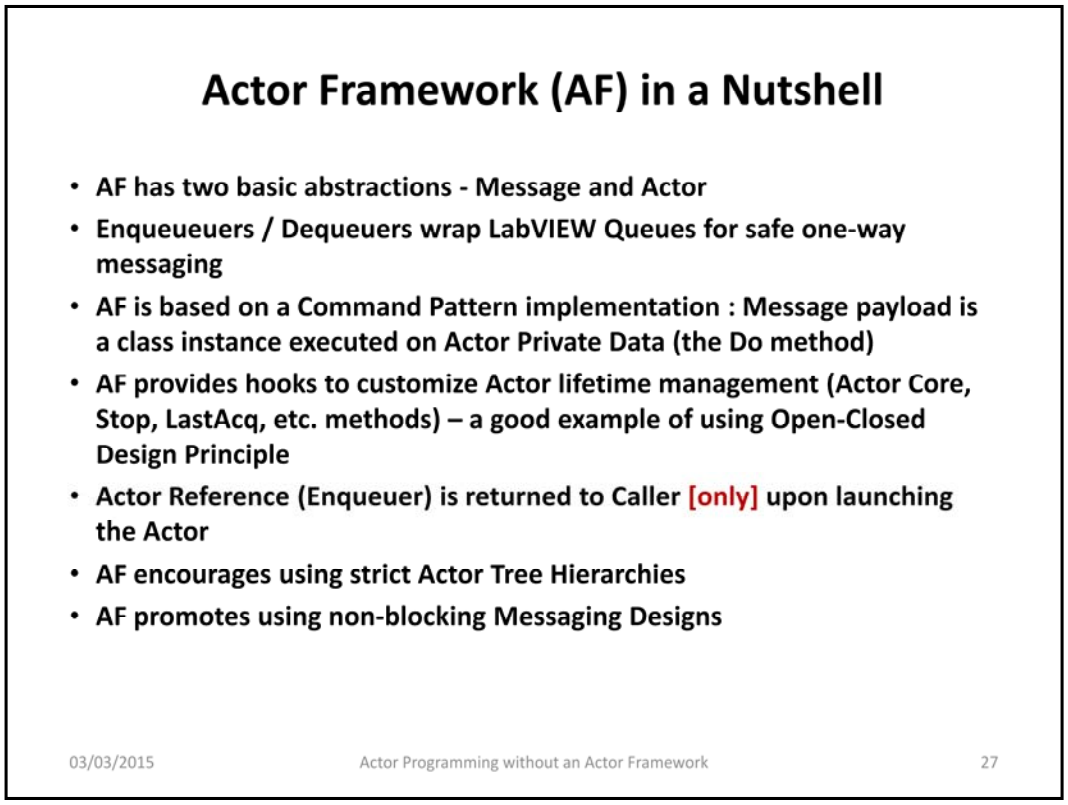
1. Assembler Class hosts all Application-Specific Code – enabling greater reuse level for Injected Components
2. Configuration Storage, Logging and GUIs are examples of Application-Specific Code (subject to change). Both shall be handled at the Assembler Level
3. All class methods, not defined by Abstraction Interfaces, can only be called from Assembler code (Assembler knows concrete types of all “injected” class instances)
4. Same is true for creating/destroying concrete Class Instances
5. Objects can pop-up their GUI/FP (or plug-in into a subpanel) to expose concrete features to User without coupling Reuse Class to such features.
6. All injected Objects have a life time of Assembler Object (i.e. while application is running)

Disclaimers

1. **I do not use Actor Framework for code development. In this presentation I will be referring to AF as a baseline to better communicate my design choices. Please correct my AF-related statements if wrong.**

2. **Do not hold me to the letter of canonical Design Pattern definitions. Such definitions evolved in by-reference synchronous space and are often not applicable to by-value asynchronous designs. My goal is to communicate such designs to fellow LabVIEW Architects without making things harder than they should be …**

I would be talking about Asynchronous Design Pattern flavors –the [only ?] ones relevant to Actor Programming.



AF has two basic abstractions – Messages & Actors and addresses the following issues:

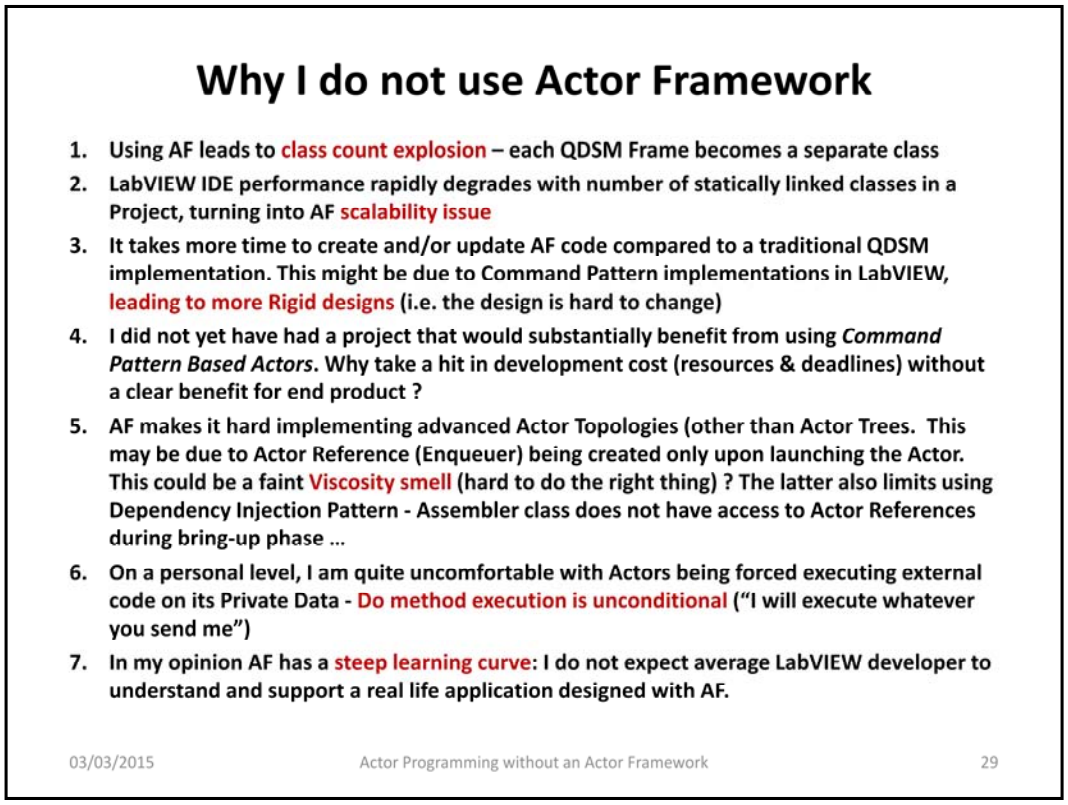
- Orderly Actor Instantiation/Destruction
- Encouraging/Enforcing a *safe* Actor Hierarchy (tree)
- Safe Message Transport
- Means for adding/redefining Actor functionality (Message.Do() method)
- Reference Lifetime Management
- Custom start-up procedures
- Etc.

To my taste, Actor Management and Message Transport coupling is unnecessarily strong in AF …

What I like about AF

1. AF is an Elegant Design
2. AF is a Scalable Design
3. But, most important, AF started a meaningful discussion on Actor Programming in LabVIEW Community

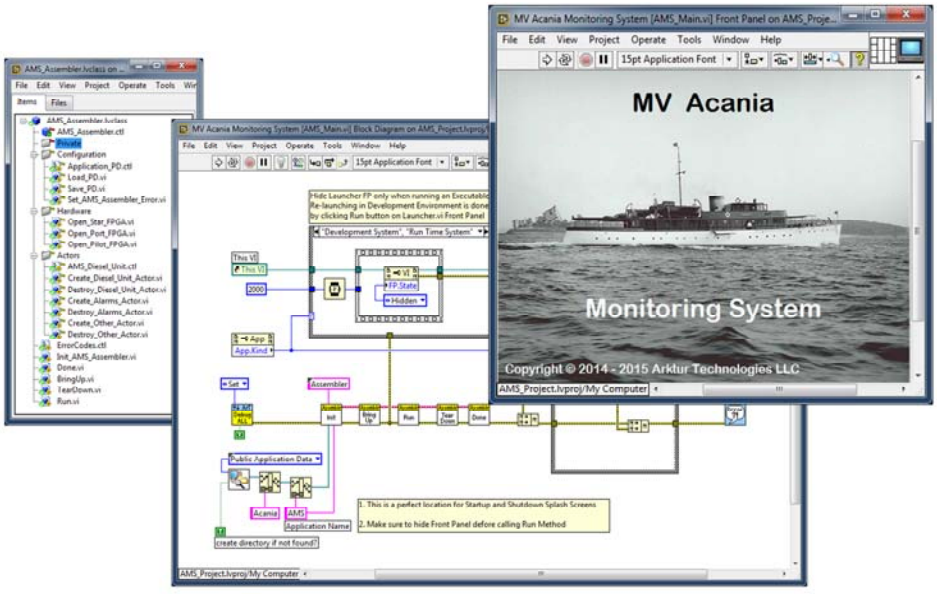
One of G undisputed strengths is its ability to support massively parallel SW Designs. Actors may be a way for unlocking this strenght for a regular LabVIEW Developer …



AF is an elegant design, but :

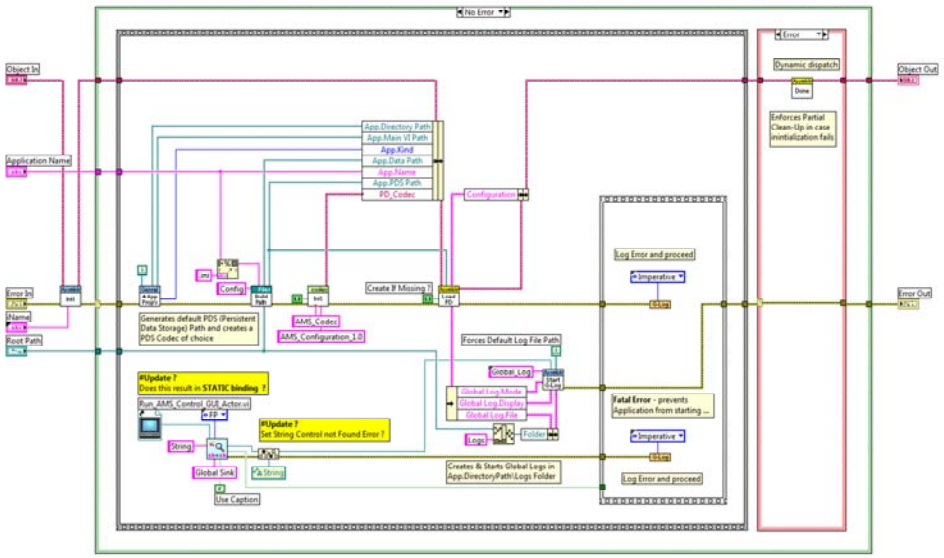
1. AF is at odds with Dependency Injection - Actor reference becomes available only upon launching the Actor. But I need it at the Bring-Up phase …
2. AF designs have a scalability issue with current LabVIEW IDE Implementation (class deluge)
3. One class per message results in substantial amount of boilerplate code (taken care by Message maker), polluting code base with unnecessary details …
4. Rigidity – it takes more effort adding/updating a Message than editing a single QDSM case
5. AF Actors do not mix well with non-AF code - non-actor applications require substantial re-factoring effort to interact with AF Actors.
6. I do not have projects that might benefit from using Command Pattern-based Actors. Why take a hit in development cost (resources) without a clear benefit for end product?
7. One cannot expect an average LabVIEW developer to understand and support a real life application designed with AF. Lack of LVOOP expertise together with Command Pattern implementation results in a very steep learning curve
8. Check out Daklu's and SE's Presentation on AF for more arguments
9. To my taste, Actor Management and Message Transport coupling is unnecessarily strong in AF …

AMS Assembler Class



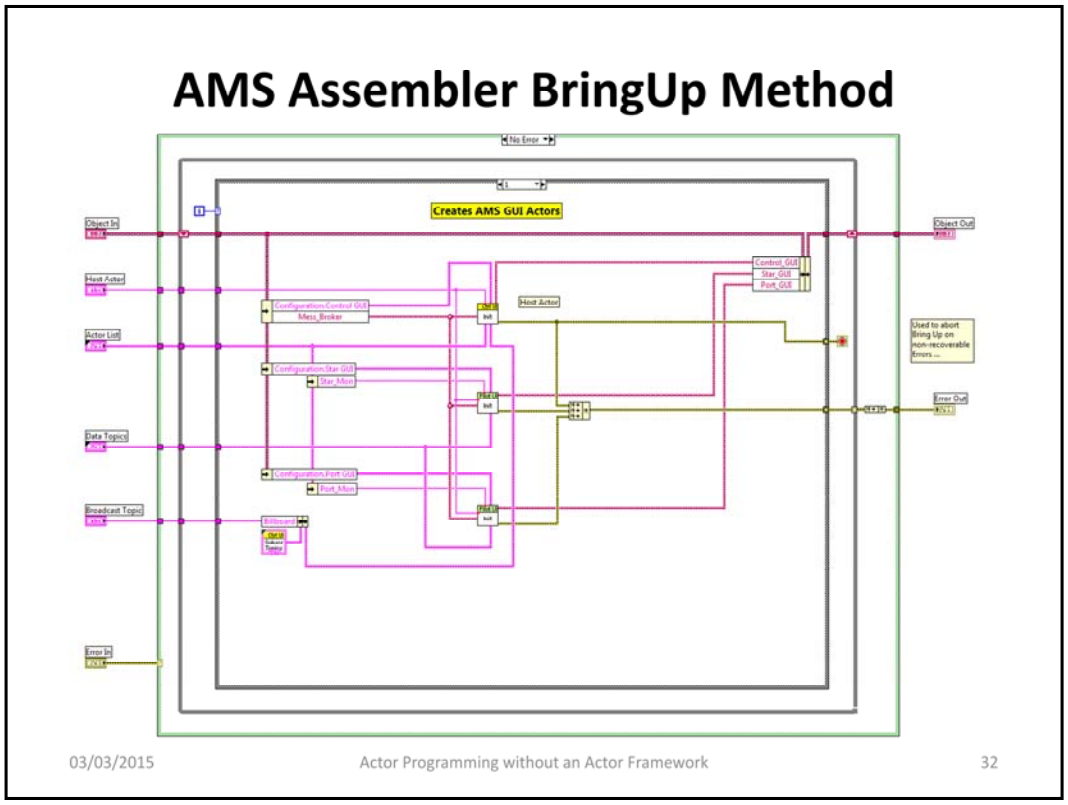
Assembler separates Reusable Code from Application Specific Details and is responsible for LabVIEW reference lifetime management

AMS Assembler Init Method



Assembler Constructor is responsible for creating/configuring all Application-wide ‘services’ :

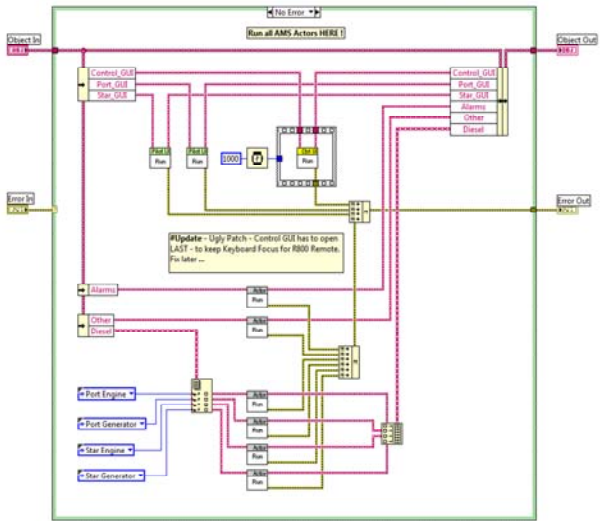
- Loading Persistent Data (Configuration File)
- Starting and Configuring Global Log
- [Starting Messaging System], etc.



I just hate Sequence Local Terminals – wish LabVIEW Allowed Shift Registers to pass data from one frame to the next – allowing to unbundle–change–bundle parts of a large data structure.

Primary Use Case: long initialization sequences, where a Stacked Sequence is primarily used for saving LV Diagram real estate …

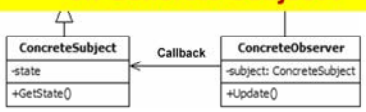
AMS Assembler Run Method



No need to override Actor Core methods for custom start-up sequence. All Actors are ready to run at this point and can be started in desired sequence. Ex: Pilot GUI Actor is set to run after a 1 second delay …

Observer Design Pattern (Canonical)

Bad :
Concrete Observer depends on Concrete Subject !



- One of original 24 Patterns in GOF Book
- This is a canonical diagram & implementation suitable **for synchronous by-reference classes**
- AKA **Publish-Subscribe** or Dependants
- LabVIEW Actors require a different approach

Source: http://www.blackwasp.co.uk/Observer.aspx

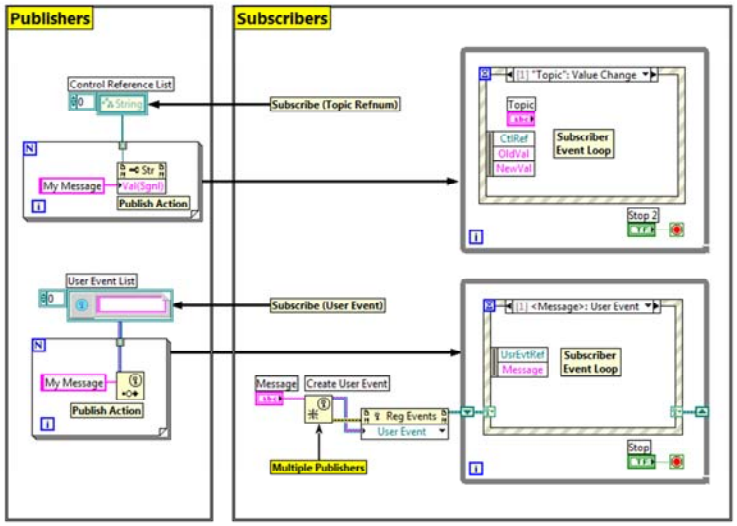
```
public abstract class SubjectBase
{
  private ArrayList _observers = new ArrayList();

  public void Attach(ObserverBase o)
  {
    _observers.Add(o);
  }

  public void Detach(ObserverBase o)
  {
    _observers.Remove(o);
  }

  public void Notify()
  {
    foreach (ObserverBase o in _observers)
    {
      o.Update();
    }
  }
}
```

03/03/2015 Actor Programming without an Actor Framework 34

1. Subject keeps an Observer List (observers)
2. Subject executes Observer call-backs on State change (o.Update())
3. Observer keeps a Subject reference
4. Observer executes Subject calls-back (Subject.GetState()) as part of its Update() method
5. **Bad :** Concrete Observer depends on Concrete Subject !

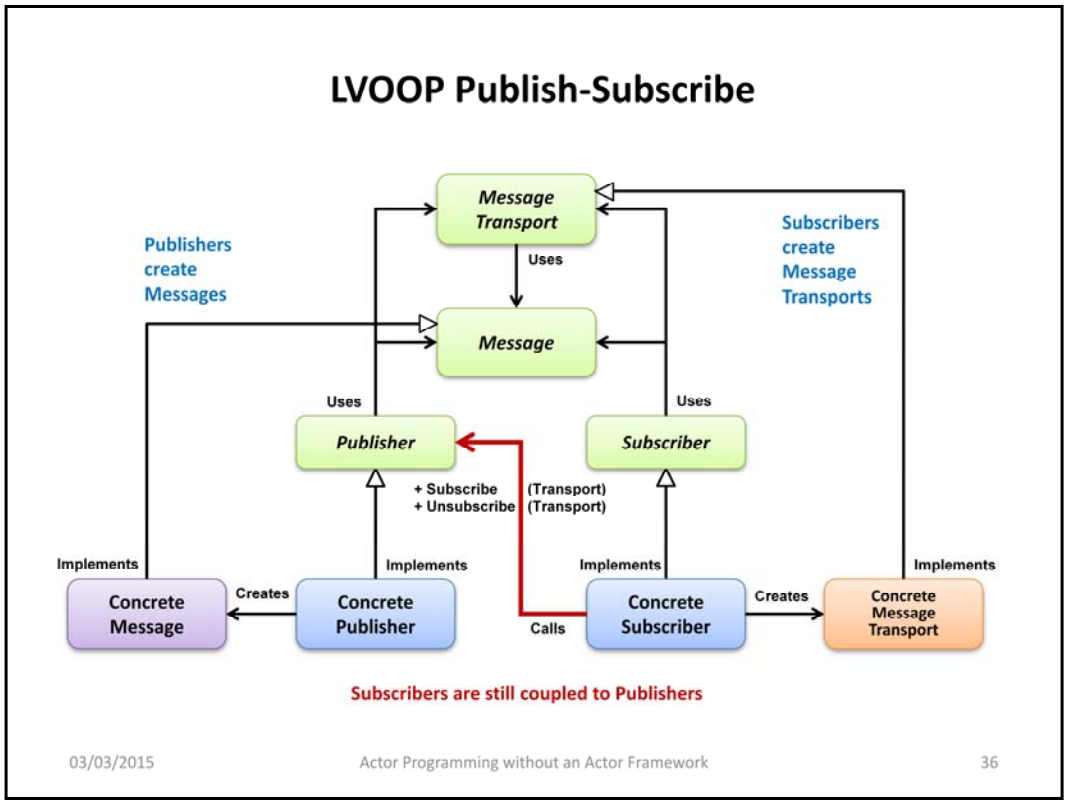
Such design does not work for by-value asynchronous objects !

LabVIEW Publish-Subscribe prior to LVOOP



Could also use LabVIEW Queues

1. Each Topic needs a dedicated pair of Register_Topic & Unregister_Topic VIs
2. All subscribers must use the same Event Mechanism to listen for Topic Messages
3. Publishers are decoupled from Subscribers
4. Subscribers are coupled to Publishers (need to send Subscribe/Unsubscribe requests to each Publisher
5. Publisher Event Loop may take a performance hit (response latency and jitter) when publishing messages at high rate



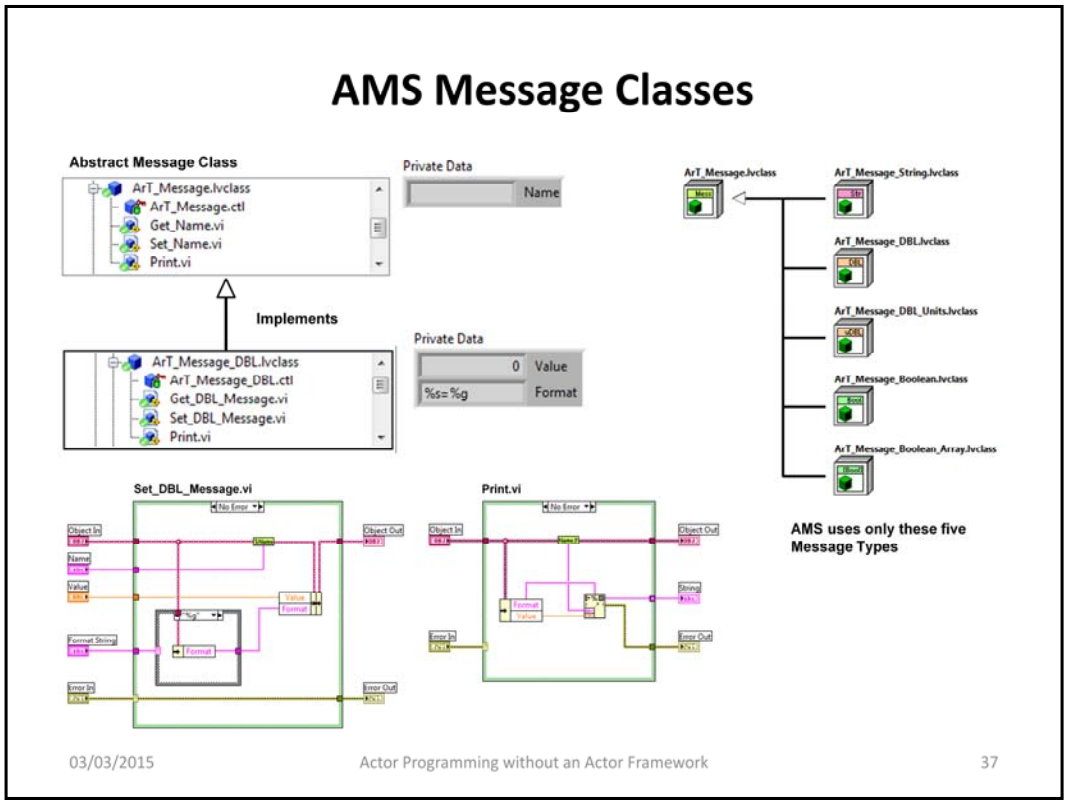
Notice Diagram asymmetry: Publishers create Messages, while Subscribers create Message Transports … this allows Subscribers with different Event Loop types to get Messages from any Publisher …

Pros:

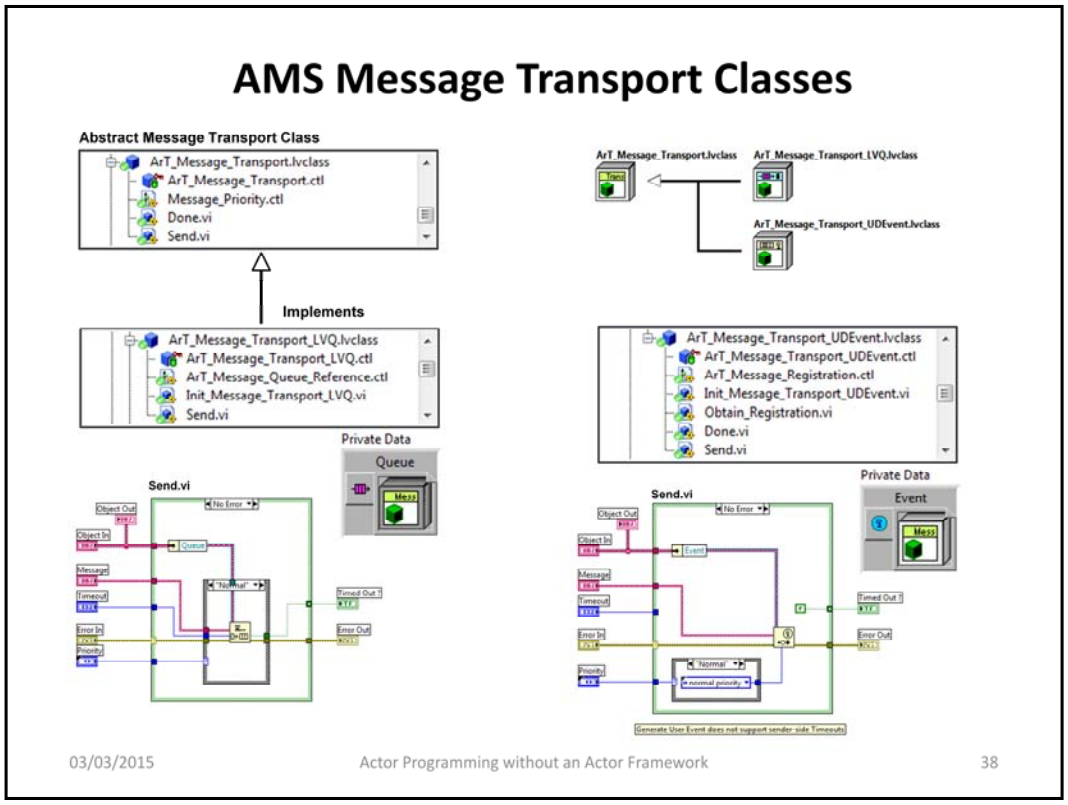
- Publishers are decoupled from Subscribers

Cons:

- Subscribers are still coupled to Publishers – must have Publisher reference to execute Subscribe/Unsubscribe methods
- Publisher must be up and running prior to Subscribers issuing Publisher.Subscribe (myTransport) calls
- Publisher Event Loop may take a performance hit (response latency and jitter) when publishing messages at high rate
- Each Publisher needs a dedicated pair of Subscribe/Unsubscribe methods for each Topic unless using a



1. Only 5 Message Types are used throughout AMS
2. Print method provides a simple way for logging Messages without knowing their Types (Visitor Pattern)
3. Set_DBL_Message is, essentially, a class constructor (static dispatch)
4. Get_DBL_Message is an accessor method (static dispatch)
5. ‘Message Subject’ would be a better choice than ‘Message Name’



1. Only 2 Message Transport Types are used throughout AMS
2. Message Transport is a **write only** object akin to AF Enqueuer. Sharing Subscriber's Message Transport does not allow snooping/grabbing Subscriber's messages from its queue.
3. Message Transport is a stateless object – OK to branch a wire and share it on the diagram
4. A wrapper class for AF Enqueuer can be easily added to Message Transport Class Hierarchy

Publish-Subscribe Summary

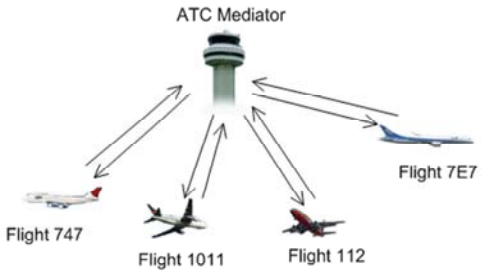
Pros:

- **Publishers are decoupled from Subscribers**

Cons:

- **Subscribers are still coupled to Publishers** – must have Publisher reference to execute Subscribe/Unsubscribe methods
- **Publisher must be up and running prior to Subscribers issuing Publisher.Subscribe (myTransport) calls**
- Publisher Event Loop may take a performance hit (response latency and jitter) when publishing messages at high rate
- Each Publisher needs a dedicated pair of Subscribe/Unsubscribe methods for each Topic unless using a Topic Look-Up Table

Mediator Design Pattern



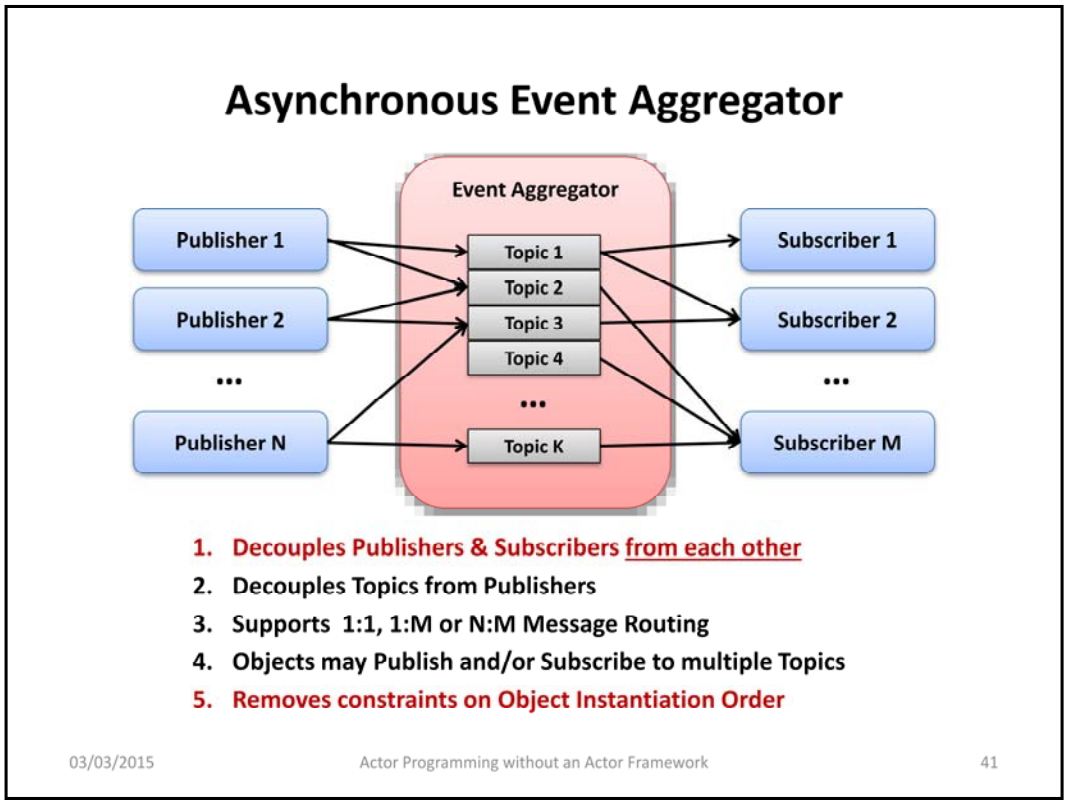
Mediator defines an object that controls how a set of objects interact.

1. **Planes communicate with Control Tower – not with each other**
2. **Control Tower imposes constraints (Policies) on arriving and departing Planes**

Monitoring System does not require Policy Enforcement – enters Event Aggregator Design Pattern

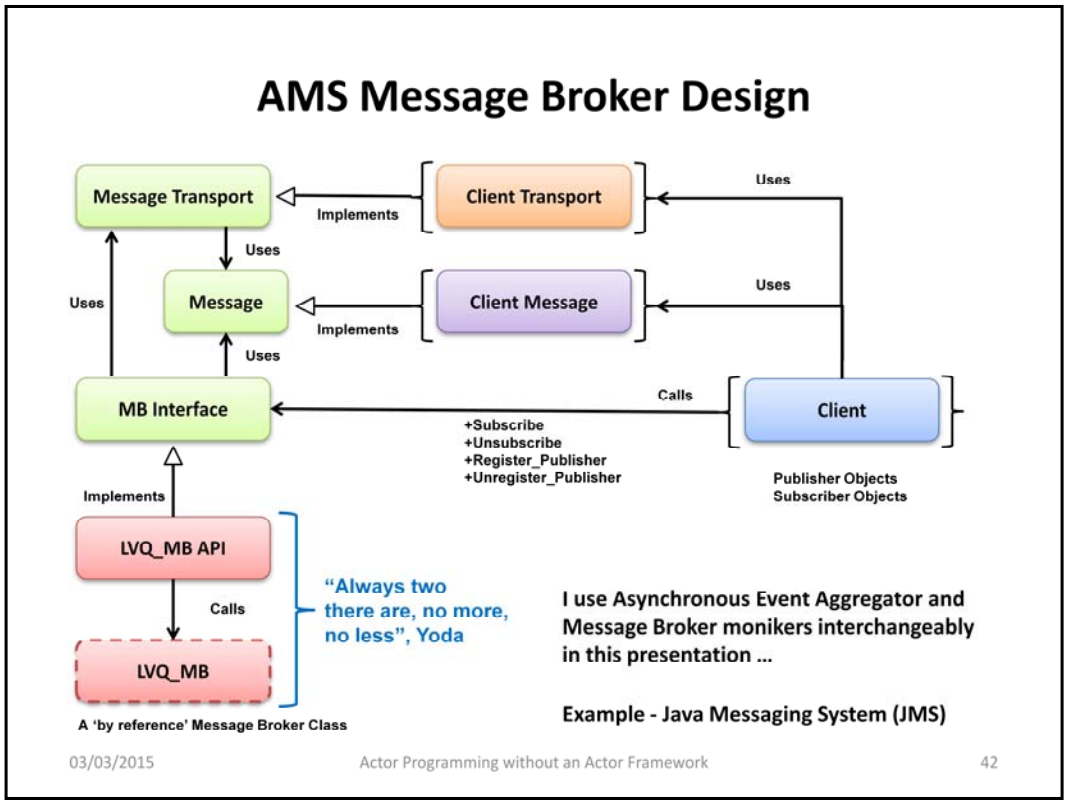
The mediator pattern is a design pattern that promotes loose coupling of objects by removing the need for classes to communicate with each other directly. Instead, mediator

objects are used to encapsulate and centralize the interactions between classes (http://www.blackwasp.co.uk/GofPatterns.aspx (Mediator & Observer are in GOF))



Removing Policies from a Mediator Pattern turns in into an Event Aggregator Pattern

http://martinfowler.com/eaaDev/EventAggregator.html

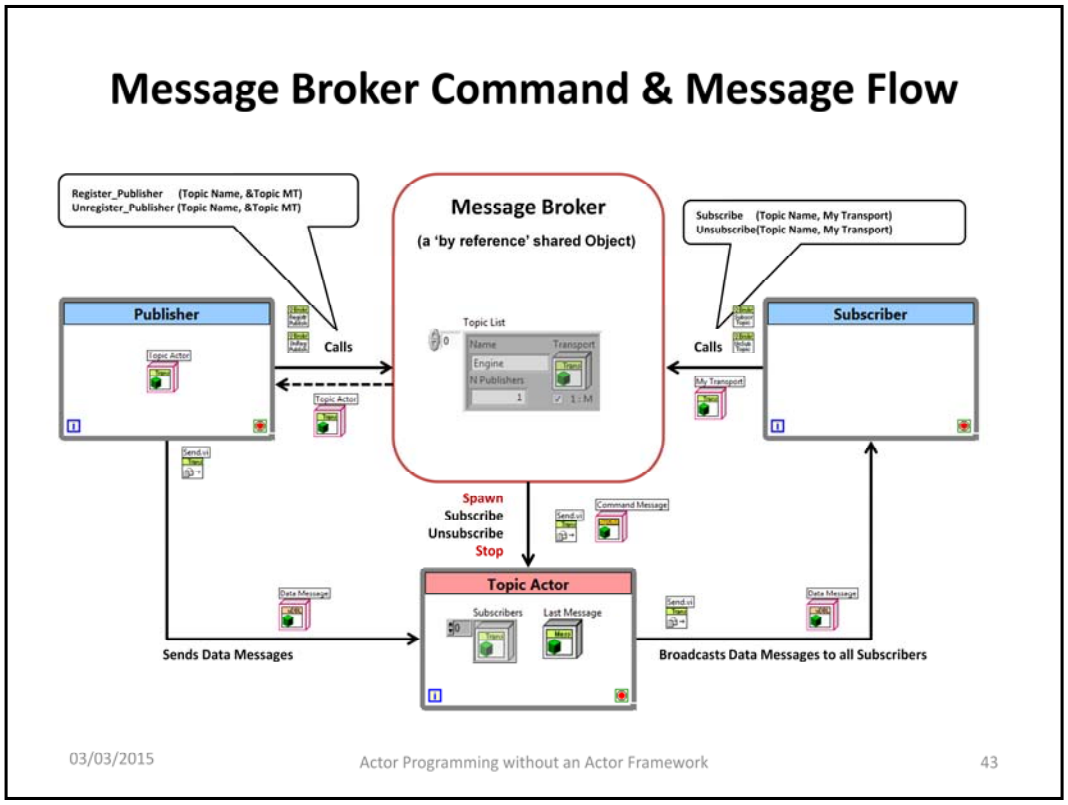


By-Reference LabVIEW Classes: The "Rule of Two" states that there would be only two Sith at one time, a Master and an Apprentice

In this presentation I use Message Broker and Event Aggregator terms interchangeably. However, there seems to be a [vague] difference between the two, as Message Broker may imply a specific functionality to some folks in Software Engineering Domain:

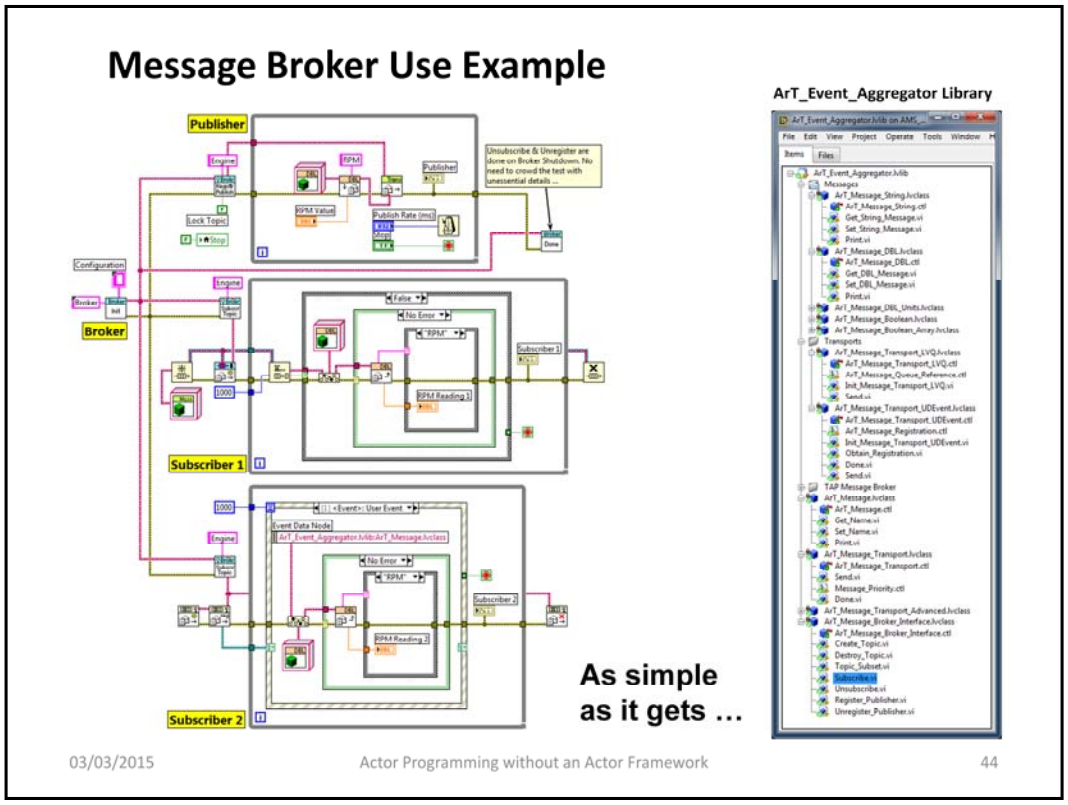
http://stackoverflow.com/questions/8184490/confusion-about-message-bus-command-dispatcher-patterns

A Message/Command Broker is concerned with connecting incompatible or differently designed independent systems. This is the use case where you want one application to interface with another and don't have the source code to one or both applications. So you create a broker which receives information from one side and provides this information on the other side taking into account any transformations necessary for these two apps to communicate. The example on MSDN is an ecommerce website which might need to talk to a payment processor, a shipping company, and an accounting system. You may not have the ability to change the source code for any of these apps (including the ecommerce system). Maybe the ecommerce system requires an IExamplePaymentGateway interface and your payment provider requires a IDifferentPaymentAPI interface. Maybe one API is implemented in XML and the other in JSON? Whatever the differences, your broker is responsible to make the connection possible.



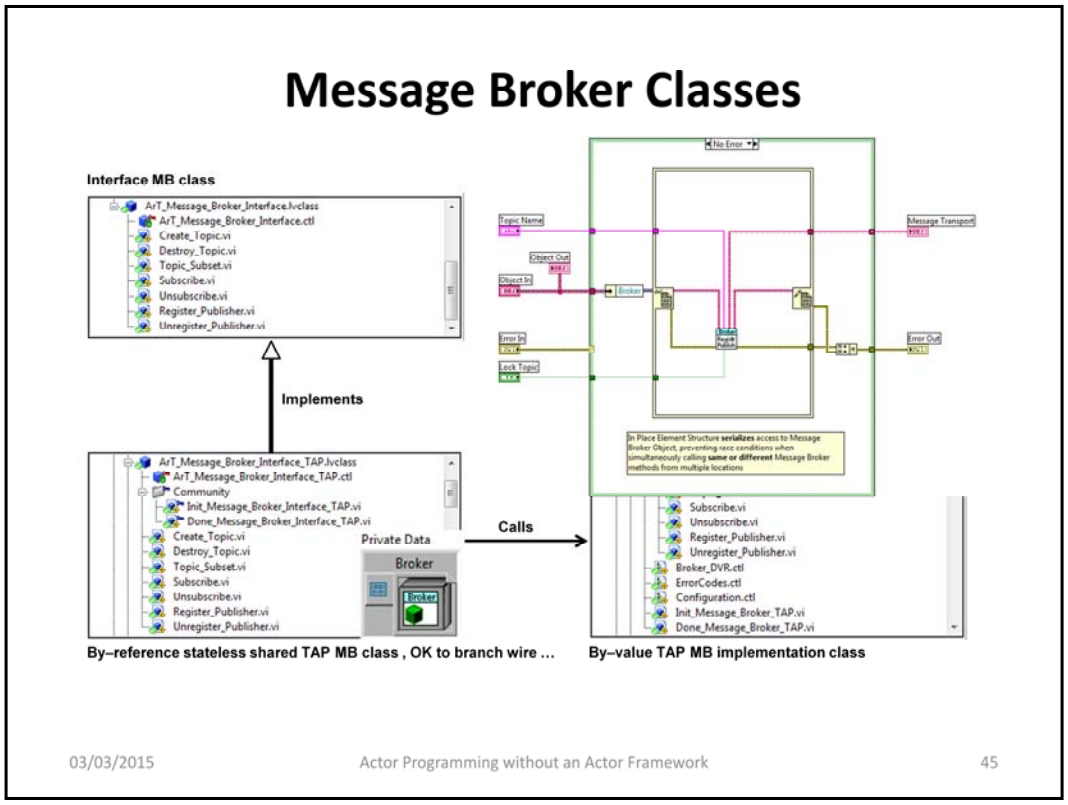
- Message Broker is a by-reference shared Object (class instance). It is not a QMS (Queue Message Handler) - nor has to be one
- Message Broker provides Publish and Subscribe Interface to callers
- Message Broker creates a Topic Actor to handle publishing Messages on a designated Topic.
- Delegating publishing to a Topic Actor removes messaging-related latency and jitter from Publisher Event Loop
- Message Broker maintains a list of Topics
- Each Topic Actor maintains a list of Subscribers it it's Topic

From a follow-up discussion – yes, it is possible for a Topic Actor to be a 'synchronous' shared by-reference Object (no Event Loop). In such case each Publisher would be calling Topic.Send () method from it's own Event Loop, possibly leading to extra latency and jitter. But in some use cases this may be a reasonable implementation option …

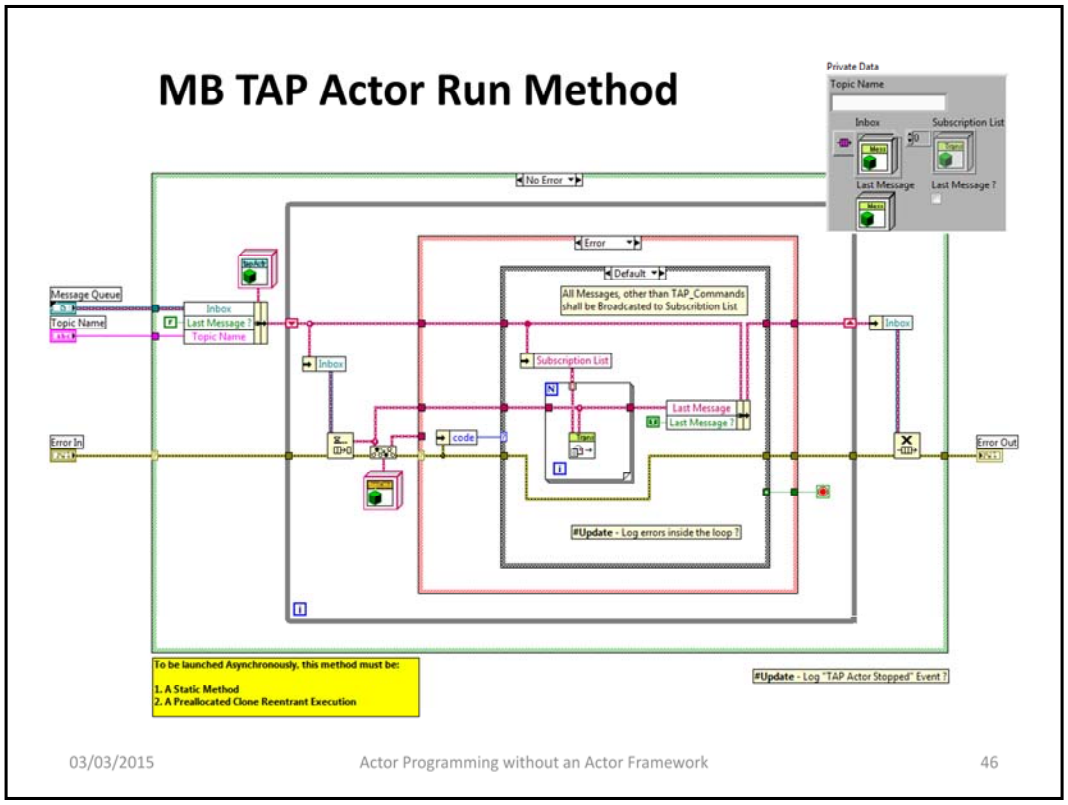


Example illustrates using a Message Broker implementation by two subscribers with different Message Transports – a LabVIEW Queue and a LabVIEW Event Structure.

It also shows the entire Event Aggregator Library used in Acania Monitoring System – making a point that implementing a extensible & scalable Messaging Platform does not have to be hard or complicated …



Shows a TAP (Topic Actor Publish) Message Broker implementation – a by-reference wrapper class and a by-value LabVIEW Queue-based Message Broker class



TAP Actor supports 3 private commands (Stop, Subscribe, Unsubscribe) issued by Message Broker

Any Message that is not a *Tap Command* is considered a Request to Publish (shown on the block diagram)

AMS Message Broker Summary

Pros:

1. Lightweight & Simple
2. Decouples Publishers & Subscribers from each other
3. Decouples Topics from Publishers - removes constraints on Publisher/Subscriber Initialization Order
4. Supports 1:1, 1:M and N:M Topic Modes
5. Decouples Topics from Subscriber Message Transport Choice (Queue, Notifier, Used Defined Event, etc.)
6. Easy to add a new Message Type
7. Easy to add a new Message Transport
8. Scalable – MB spawns a dedicated Topic Publisher Actor upon creating a new Topic. This way MB does not become a choke point

Cons:

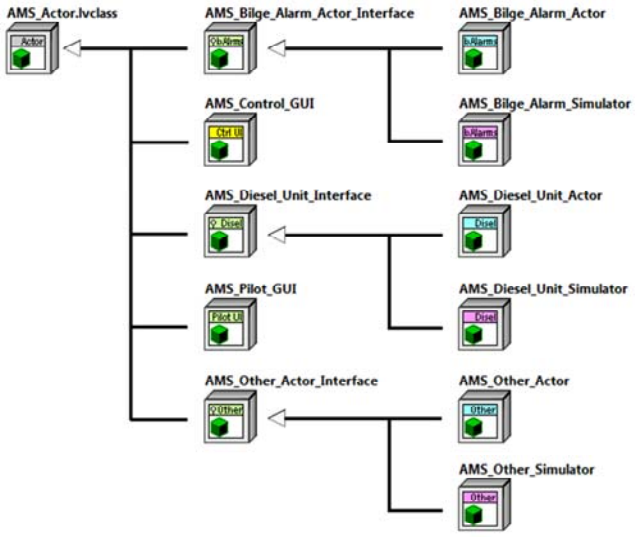
Exactly two hops per Message when using a Topic Actor

AMS Message Broker design/implementation effort took ~ 2.5 weeks. However, this came from several (5 or 6) years of prior Actor Programming experience.

AMS Actor implementation effort - 3 days

Replacing a Topic Actor with a 'synchronous' Topic Publisher object implementation will reduce number of message hops to one. But this may come at a cost of increased publisher Event Loop latency and jitter.

Where are AMS Actors ?



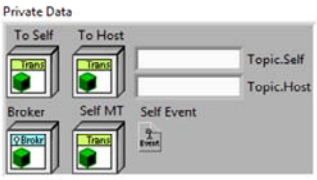
There is no need in AMS Root Actor.

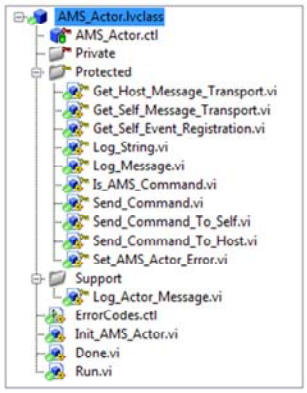
It has been introduced for convenience only – mainly as a way to reuse common AMS Actor functionality …

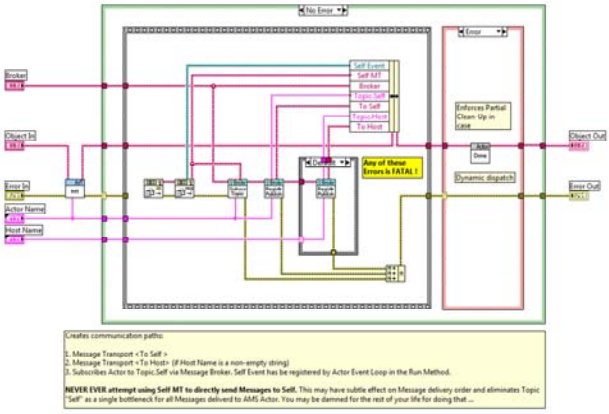
All Data Producer Actors provide a Simulator option to run, test and debug AMS code without being on the boat or starting engines.

Slide Notes

AMS Root Actor





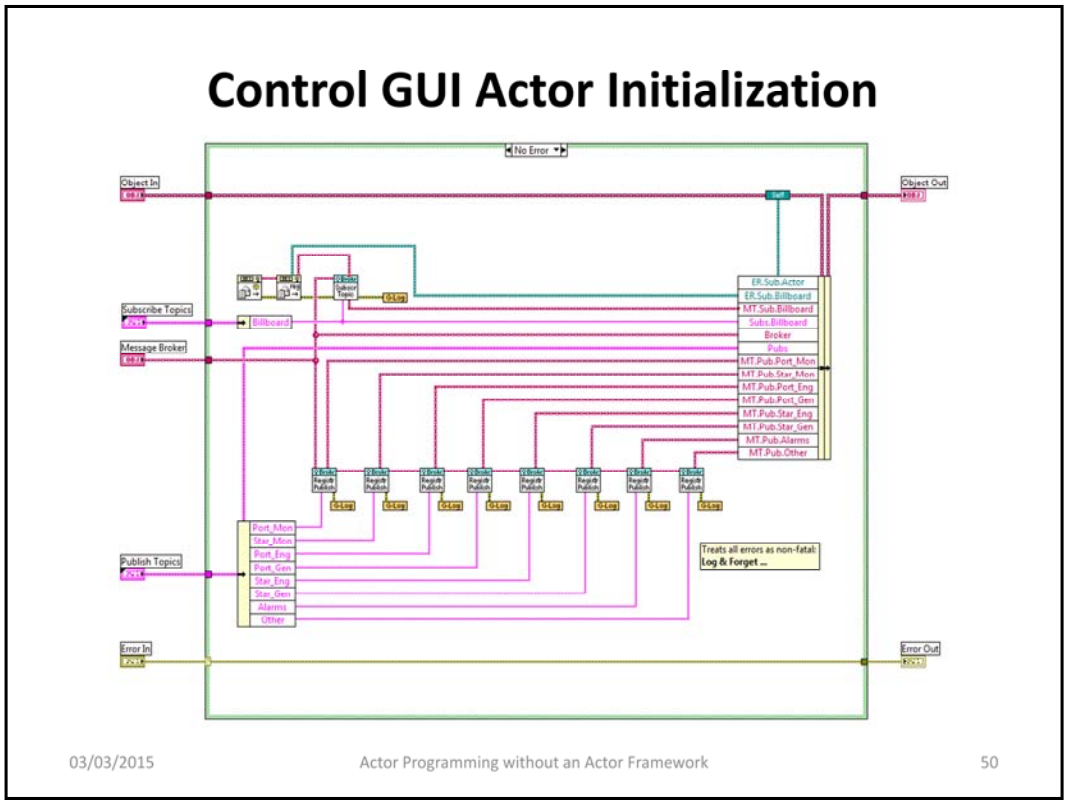


Actions:

1. Creates Message Transport <To Self >
2. Creates Message Transport <To Host> (if Host Name is a non-empty string)
3. Subscribes Actor to Topic.Self via Message Broker. Self Event has be registered by Actor Event Loop in the Run Method.

All AMS Actors use Private Topics for sending/receiving Command Messages to/from other AMS Actors (including sending Command Messages to Self)

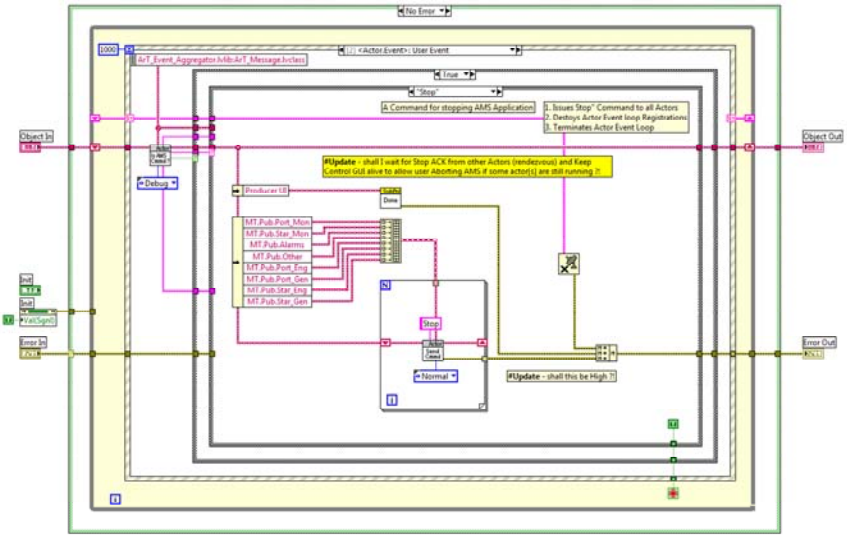
This allows listening for both, Commands Messages and other Messages (from multiple Topics) in the same Event Loop.



Registers for sending Command Messages to all Data Producer Actors

Registers subscription to application-wide announcement topic (Billboard) with Message Broker and obtains Message Event Registration for Control GUI Event Loop

Control GUI: Stopping All Actors



Crucial – Actors must listen for Command Messages on the same Message Transport as all other Messages – an Event Loop with two or more data sources creates more problems than it seems to solve …

However, adding a Message Stream input may be a different story. I would look at implementing streaming with a private Worker Loop (one stream per Worker)

Integrating AF with ArT Message Broker

I would be happy to brainstorm this subject with AF savvy architects …

AMS Messaging Platform Summary

1. Two basic abstractions - Messages and Message Transports
2. No need in Actor base class – custom Actor Flavors (with or without a common parent) can be created on demand for convenience
3. Messaging is handled in a uniform & scalable manner by Message Broker – providing seamless interaction between Actors of different Flavors
4. Developers get to decide on choosing Actor Flavors (Command Pattern vs User Event QMH vs LVQ QMH, etc) based on Requirement
5. Supports advanced Actor Topologies (in case Tree Hierarchy is not the best choice)
6. Enables using full scale Dependency Injection (creating Actor Reference is decupled from launching the Actor)
7. Actor & Message Transport lifetime is defined by Assembler object
8. Actor instantiation & configuration order is not critical (a Message Broker perk)

Slide Notes

References & Resources

Books

[1] **Agile Software Development**, Robert C. Martin, Prentice Hall, 2003

[2] UML Distilled, Third Edition, Martin Fowler, Addison-Wesley, 2004

In LabVIEW Domain

[3] On Using Dependency Inversion, Dmitry Sagatelyan, CLA Summit 2014, Austin

[4] ArT Actors: A Step Beyond Actor Framework 3.0?, Dmitry Sagatelyan, Actor Framework Community Forum, 2012

The Web

[5] **Twelve Principles of Agile Software**, agilemanifesto.org

[6] The Principles of OOD, butUncleBob.com

[7] Pragmatic SOLID – Part 5 – The Dependency Inversion Principle, Robert C. Martin

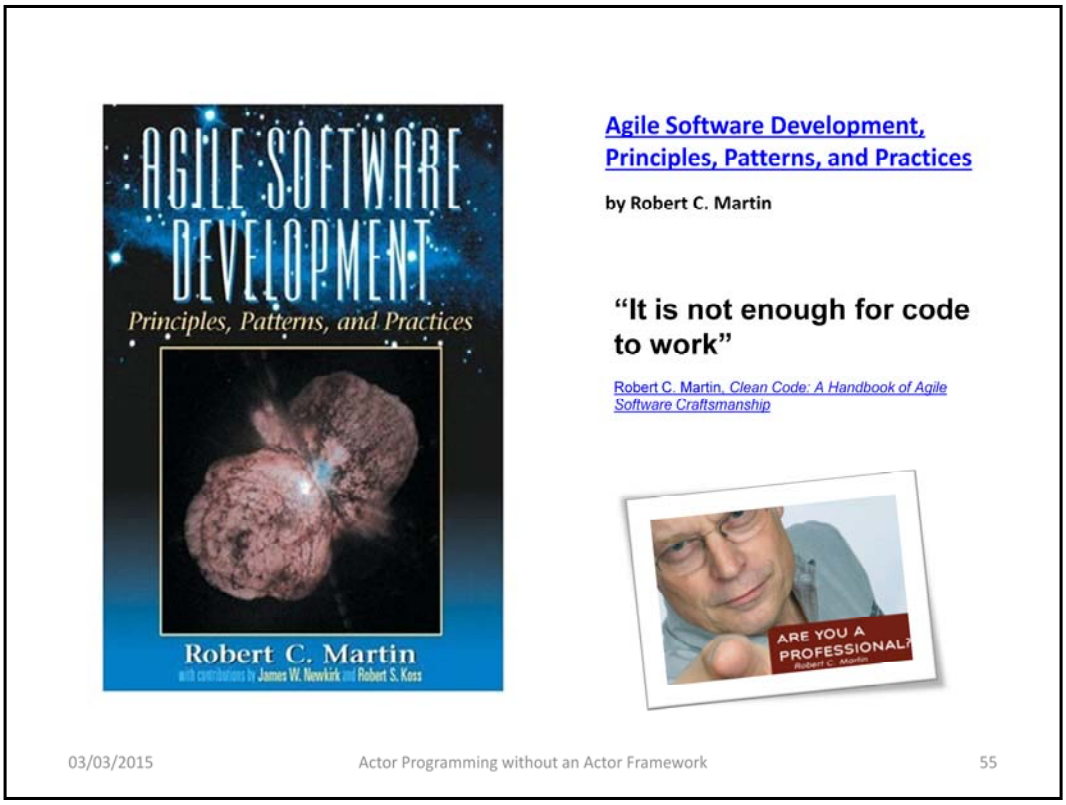
[8] Event Aggregator, Martin Fowler

[9] Dependency Injection, Griffin Caprio, MSDN Magazine, September 2005

[10] Dependency Injection Demystified, James Shore, 2006

[11] Gang of Four Design Patterns, blackwasp.co.uk

[12] Observer Design Pattern, blackwasp.co.uk



"It is not enough for code to work. Code that works is often badly broken. Programmers who satisfy themselves with merely working code are behaving unprofessionally. They may fear that they don't have time to improve the structure and design of their code, but I disagree. Nothing has a more profound and long-term degrading effect upon a development project than bad code. Bad schedules can be redone, bad requirements can be redefined. Bad team dynamics can be repaired. But bad code rots and ferments, becoming an inexorable weight that drags the team down. Time and time again I have seen teams grind to a crawl because, in their haste, they created a malignant morass of code that forever thereafter dominated their destiny."

Symptoms of Poor Design

1. **Rigidity** – The design is hard to change
2. **Fragility** – The design is easy to break
3. **Immobility** – The design is hard to reuse
4. **Viscosity** – It is hard to do the right thing
5. **Needless Complexity** – Overdesign
6. **Needless Repetition** – Mouse abuse
7. **Opacity** – Disorganized expression

[1] Robert C. Martin, p.85

Discussion