Futureproofing Software with Clean Architecture

Dmitry Sagatelyan

Certified LabVIEW Architect

LabVIEW Champion

sagatedm@arkturtech.com

CLA Summit 2019

Arktur Technologies LLC

Austin

Abstract

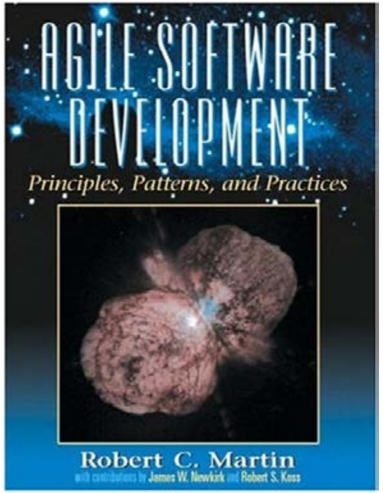
“Clean Architecture” is the latest (2017) book penned by Robert C. Martin (AKA Uncle Bob). It takes us beyond SOLID Design Principles and Package Design Principles into the realm of Policies, Business Rules, Layers, Architectural Boundaries and The Dependency Rule: “Source Code Dependencies must point only toward higher-level Policies”.

It provides much needed guidance on building applications that can withstand requirement changes at incremental cost (AKA Software Futureproofing). It shows that Frameworks, Operating Systems, Data Bases and User Interfaces (including “The Web”) are details – and, as such, should be decoupled from code implementing high-level Policies and Business Rules. It is not possible providing a helpful “Clean Architecture” review in a single presentation. Instead, I will try making a case for investing time and effort in digesting it by going over top-level ideas and several unorthodox conclusions.

About Myself …

- Software Engineer by Training
- Master of Science in Computer Science
- Using LabVIEW since 1998 (LV 5.0)
- Certified LabVIEW Architect (2012 – 2026)
- LabVIEW Champion
- 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 – a first (?) LabVIEW Reuse Library built on SOLID Principles (2011-19)
- Full Time LabVIEW Consultant in San Francisco Bay Area (Arktur Technologies LLC)
- Presentation List

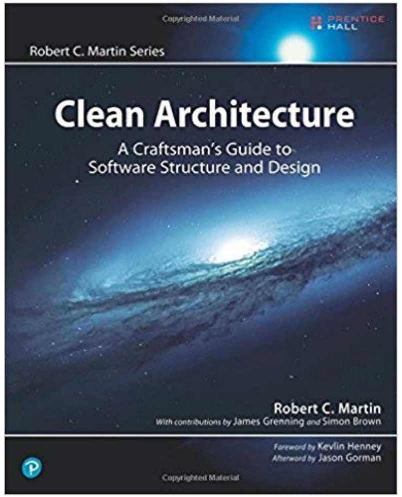
The Must Read Books


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

2003

Robert C. Martin
AKA 'Uncle Bob'

2017


Copyrighted Material
PRENTICE HALL
Robert C. Martin Series
Clean Architecture
A Craftsman's Guide to Software Structure and Design
Robert C. Martin
With contributions by James Grenning and Simon Brown
Foreword by Kevlin Henney
Afterword by Jason Gorman
Copyrighted Material

- **Focused on SOLID and Package Design Principles (Module and Component Levels)**
- **Hard to Read: Plenty of Examples in C++**
- **Hard to Digest: Novel Concepts, Dense Content**
- **May be read by Topic**

- **Focused at the Architectural Level**
- **Requires prior understanding of SOLID and Package Design Principles**
- **Easy to Read: No C++ Examples**
- **Hard to Digest: May seem too generic to be useful – hard to extract actionable items**
- **Must read entire book**

Clean Architecture : Select Quotes

“Have you experienced the impedance of bad code and rotten design?”, p.2

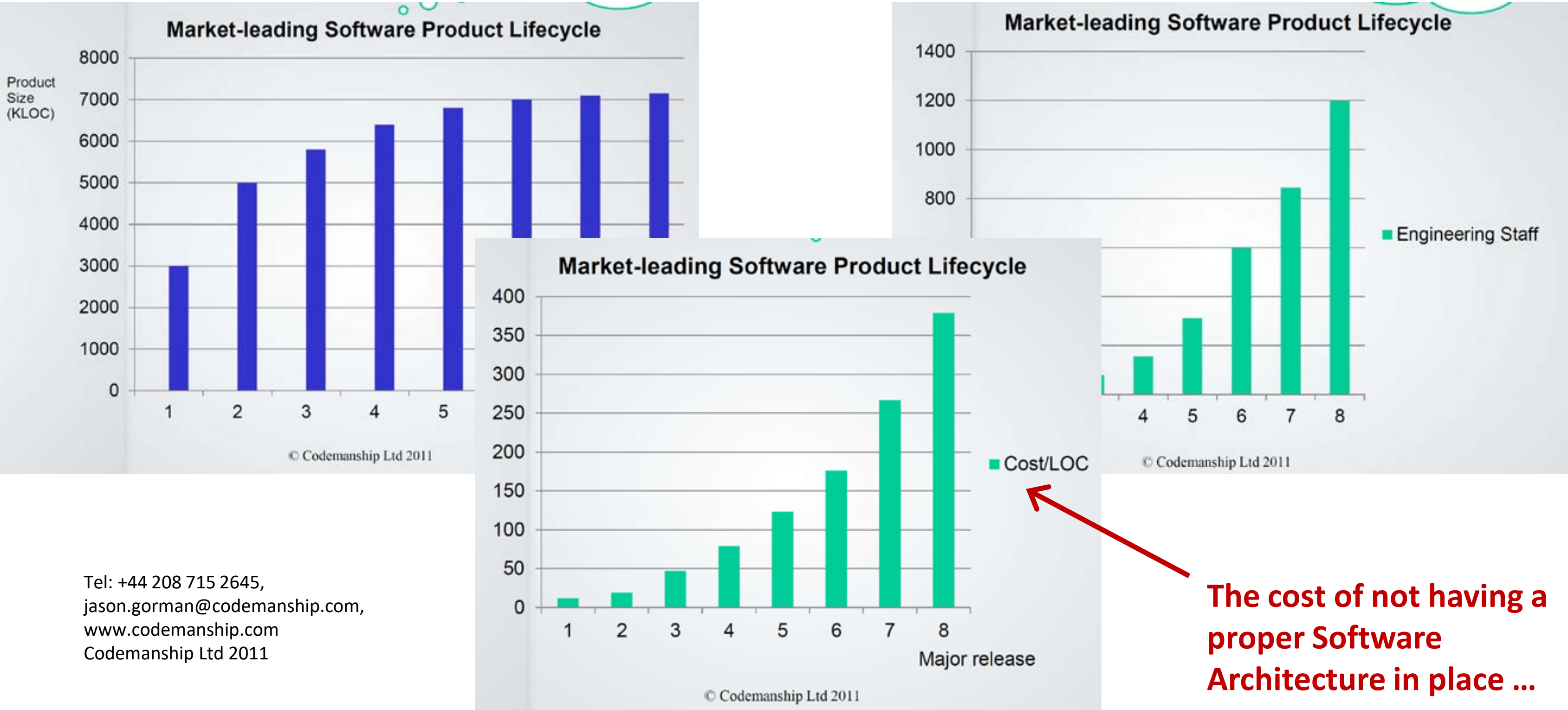
“The word ‘architecture’ conjures visions of power and mystery”, p.136

“When we think of a software architect, we think of someone who has power, and who commands respect”, p.136

“The dilemma for software developers is that business managers are not equipped to evaluate the importance of architecture. That’s what software developers were hired to do. Therefore, it is the responsibility of the software development team to assert the importance of architecture over the urgency of features”, p.17

“This was a startup. We worked 70 to 80 hours per week. We had the vision. We had the motivation. We had the will. We had the energy. We had the expertise. We had equity. We had dreams of being millionaires. We were full of shit”, p.365

Signature of a Mess



Tel: +44 208 715 2645,
jason.gorman@codemanship.com,
www.codemanship.com
Codemanship Ltd 2011

The cost of not having a proper Software Architecture in place …

Future-Proofing

“Future-proofing is the process of anticipating the future and developing methods of minimizing the effects of shocks and stresses of future events”

Wikipedia

Observation 1:

“We live in a world of changing requirements …”

Observation 2:

“Requirements change in ways that the initial design did not anticipate”

Goal:

“… make sure that our software can survive those changes”

[2] R.Martin, pp. 92 & 89

Software Values to Stakeholders

"Every software system provides two different values to the stakeholders: behavior and structure", p.14

- **Behavior is defined by Requirements**
- **Structure supports implementing Requirements**
- **But Structure has very little effect on the Behavior**
- **Instead, Structure determines the cost of implementing and maintaining the system**
- **A system with bad Structure might work well, but would be very expensive to maintain**
- **This Software System Structure is typically referred to as Architecture**

What IS Software Architecture ?

*Architecture represents the significant design decisions that shape a system, where significant is **measured by cost of change**.*

—Grady Booch

*The architecture of a software system is **the shape** given to that system by those who build it … The purpose of that shape is to facilitate the development, deployment, operation, and maintenance of the software system contained within it.*

—Uncle Bob

Architecture is about the important stuff. Whatever that is.

—Martin Fowler

Goal of Software Architecture

"The goal of software architecture is to minimize the human resources required to build and maintain the required system.", p.5

" … the architecture of a system has very little bearing on whether that system works. ***There are many systems out there, with terrible architectures, that work just fine.*** *Their troubles … occur in deployment, maintenance, and ongoing development."*, p.136

"The way you keep software soft is to ***leave as many options open as possible, for as long as possible****. What are the options that we need to leave open? They are the details that don't matter."*, p.140

If you think good architecture is expensive, try bad architecture*.*

—Brian Foote and Joseph Yoder

What Makes a Software Architect ?

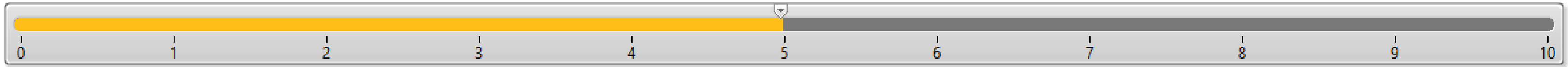
- Is a Tech Lead
- Works with Marketing, Management and Customers
- Contributes code (eats own dogfood)
- Scope: Entire Software System

- Makes all important decisions early on (for others to work on)
- Walks on water
- Risks becoming an architectural bottleneck

- Makes system-wide decisions (ex: draws architectural boundaries)
- Finds Solutions (looks out for and resolves important system issues)
- Mentors and Guides Development Team
- Delegates decision making to qualified developers

Who Needs an Architect ?
Martin Fowler

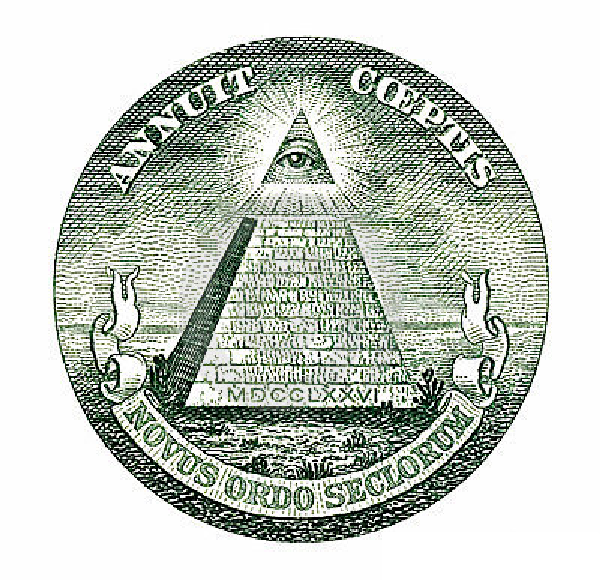
Where do **you** stand ?



Agile Software Design: The Big Picture

1. Agile Design Philosophy
2. The Dependency Rule
3. Package Design Principles
4. SOLID Principles
5. Design Patterns
6. Code

Programming Language-Agnostic



Agile Software Design Philosophy

“We live in a world of changing requirements …”

“Requirements change in ways that the initial design did not anticipate”

“… make sure that our software can survive those changes”

[2] R.Martin, pp. 92 & 89

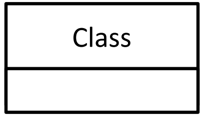
Translation:

- **Make your code as good as it needs to be at the moment**
- **Refractor it as requirements change**

Refactoring comes at incremental cost when code is designed with Agile Software Design Principles in mind …

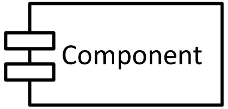
Agile Philosophy = Continuous Refactoring

SOLID Principles

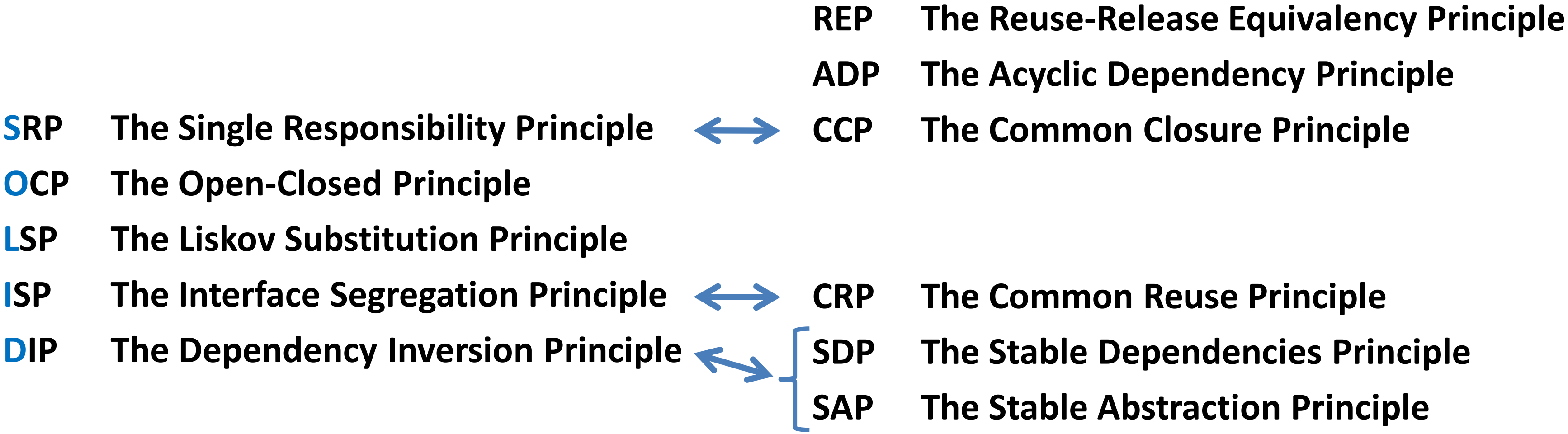


Applied at Module Level
(.lvclass & .lvlib)

Package Design Principles



Applied at Component Level
(VIPM & NIPM Packages, PPLs)



REP The Reuse-Release Equivalency Principle

ADP The Acyclic Dependency Principle

SRP The Single Responsibility Principle ⟷ CCP The Common Closure Principle

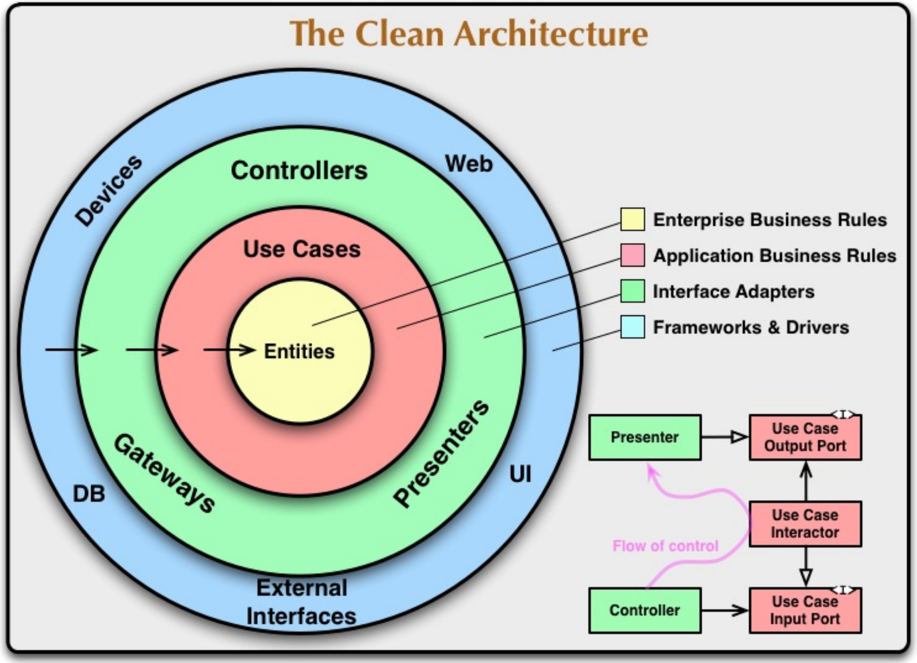
OCP The Open-Closed Principle

LSP The Liskov Substitution Principle

ISP The Interface Segregation Principle ⟷ CRP The Common Reuse Principle

DIP The Dependency Inversion Principle ⟷ SDP The Stable Dependencies Principle, SAP The Stable Abstraction Principle

The Clean Architecture



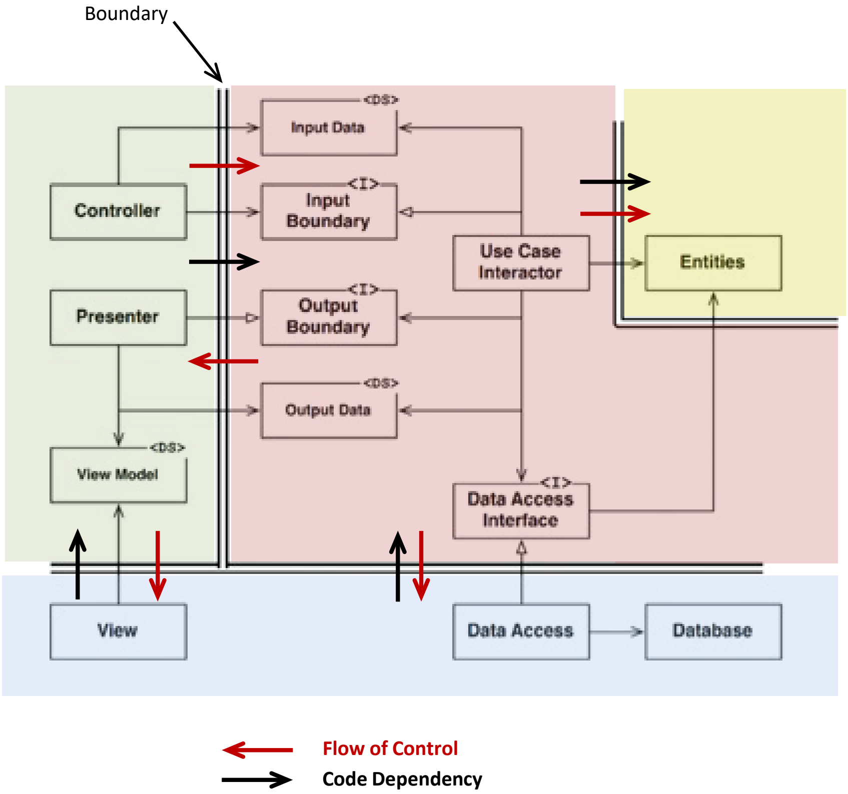
The Dependency Rule:
Source code dependencies must point only inward, toward higher-level policies

Goal: to leave as many options open as possible, for as long as possible

Clean Systems are:

- **Independent of Frameworks**
- **Independent of the UI**
- **Independent of the Database**
- **Testable**

Architectural Boundaries



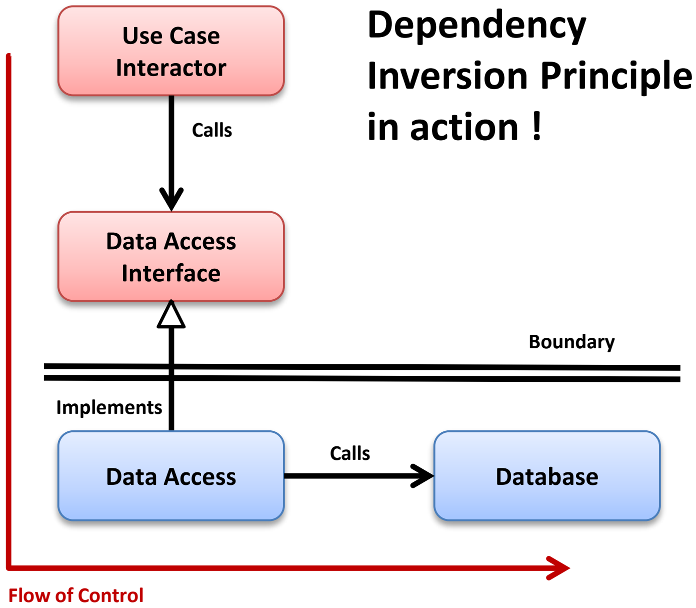
Boundary crossing @ runtime:

- A VI/method call
- Sending Message
- Triggering Event
- Use simple data structures or objects for passing data

The Dependency Rule:

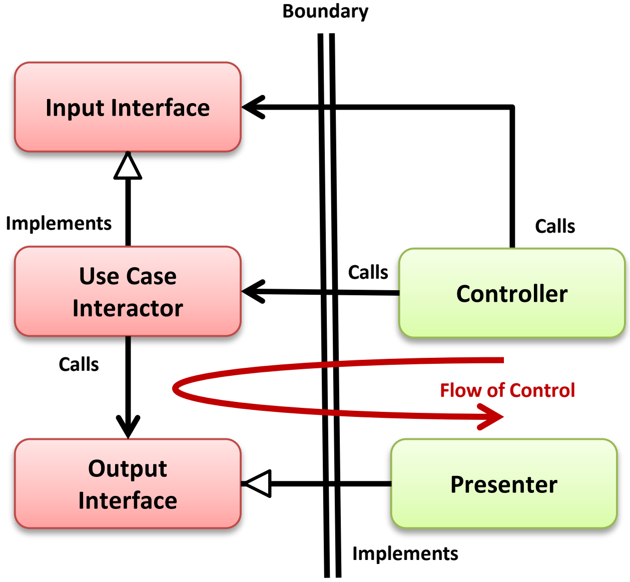
Source code dependencies must point only inward, toward higher-level policies

How to Cross Boundaries ?



Dependency Inversion Principle in action !

Partial (one way) Boundary



Full (two way) Boundary

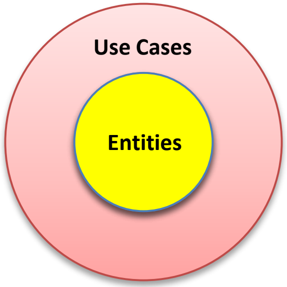
Policy and Details

- Policy embodies all business rules and procedures of the System
- Policy constitutes true value of the Software System

- Details enable humans and other systems to communicate with Policy.
- Details do not impact behavior of the Policy
- Details include IO devices, Databases, Web Systems, Servers, Frameworks, Communication Protocols, etc.

- Architects goal is to clearly separate Policy from Details

Business Rules



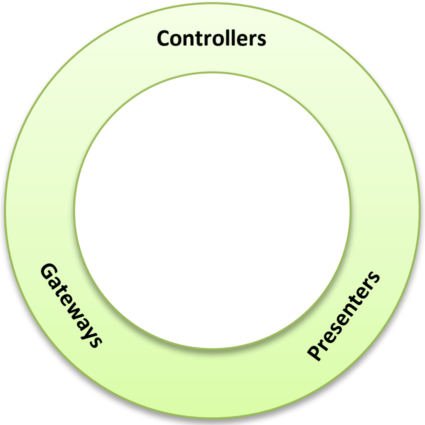
- Business Rules are domain-specific rules and procedures that make or save the business money
- **Critical Business Rules do not require use of a computer** - they may be executed manually as well …
- **Entities** = Critical Business Rules + Critical Business Data

Example: **Tax Preparation Software**

- **Entities** – Federal/State Tax Forms
- **Use Cases** – Tax Form Entry Workflows

- **Use Cases** contain rules that specify how and when Critical Business Rules are applied within a software system
- Use Cases are application-specific (do not exist outside a software system)
- Use Cases orchestrate interactions between **Users** and **Entities**

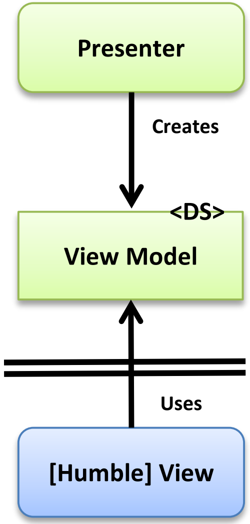
Interface Adapters



- Clean Architecture is based on MVP Design
- All Controllers/Presenters belong here
- MVC “Views” are split into Presenters & Views
- All Actors, Worker Loops and Message Handlers should be confined to this level

- Gateways translate data structures generated by Frameworks to data structures used at the Use Case Level
- Gateways decouple Use Case Level from interacting with external Services via protocols

Humble Objects

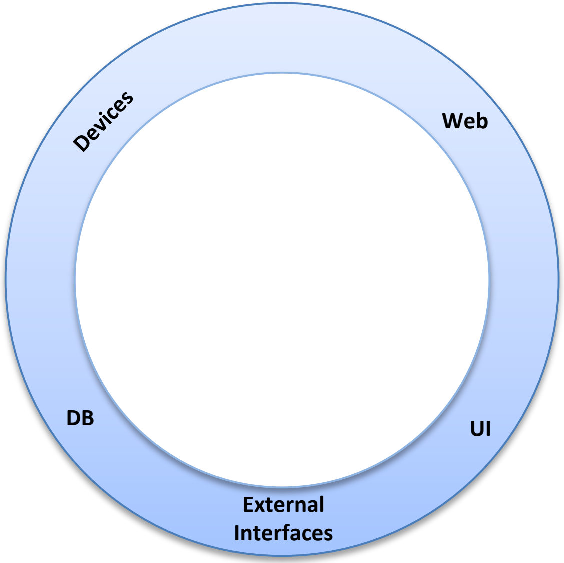


The *Humble Object* Design Pattern separates behaviors that are hard to test from behaviors that are easy to test

- Presenter contains all [event handling] logic
- Presenter is easy to unit-test

- View is a Humble Object – it renders “View Model” data structure to an output device (i.e. display) with no business logic inside
- It is hard to automate testing View, but View is a small class

Details : Frameworks & Drivers

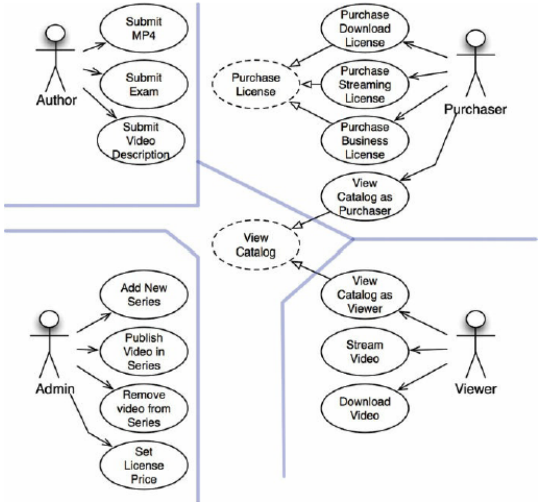


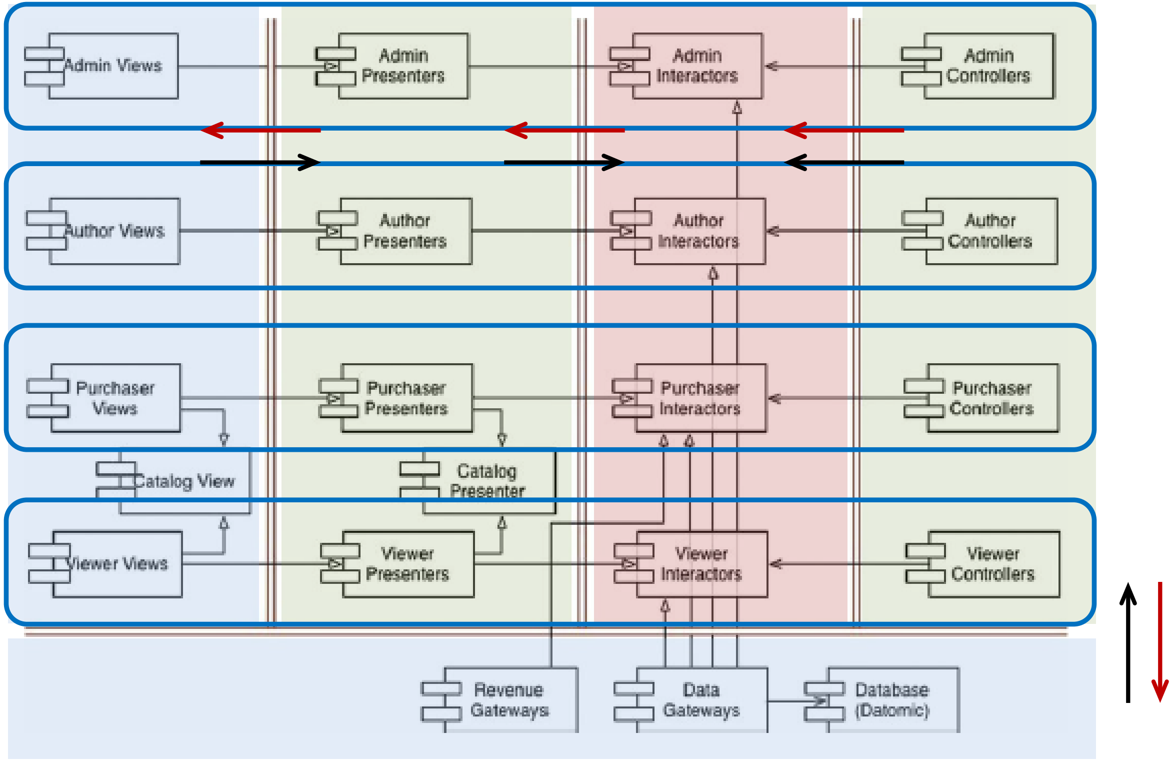
A good architecture emphasizes the Use Cases and allows deferring decisions about

- Database (Persistent Data Storage)
- UI (including Web)
- I/O (Devices)
- Frameworks

until much later in the project and makes it easy to changed your mind about those decisions …

Agile Video Sales App Example

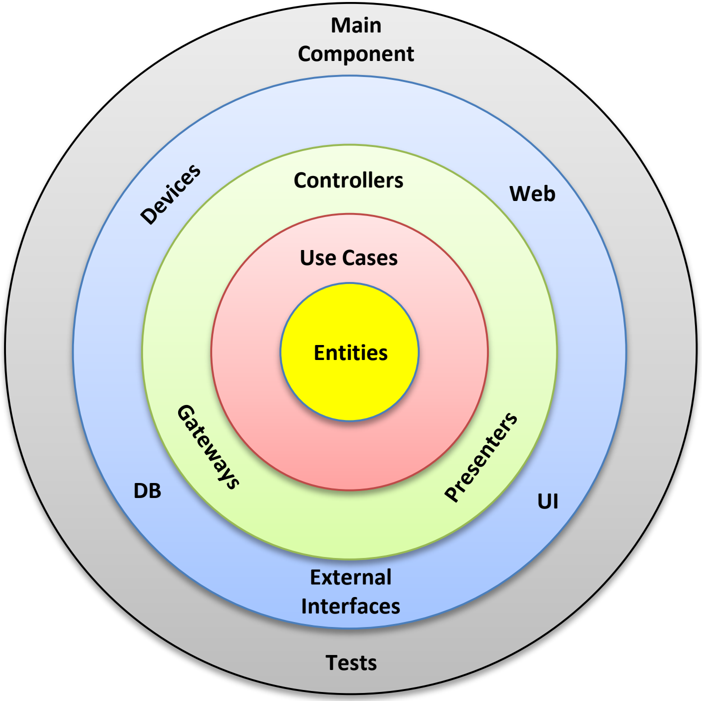




Flow of Control

Code Dependency

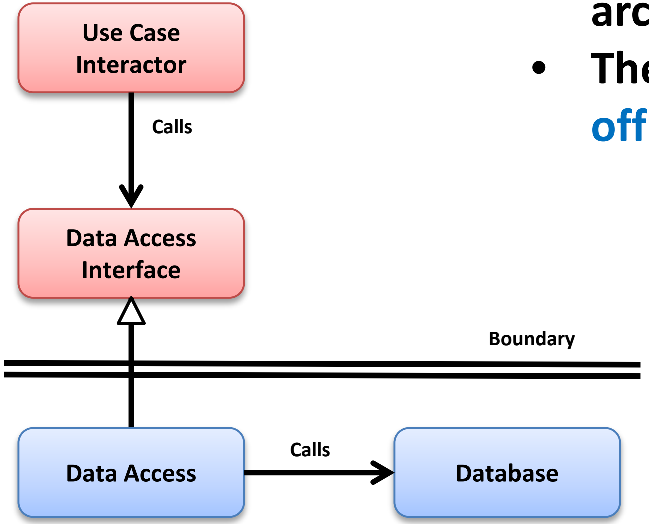
Main Component & Tests



- “Main.VI” – executes Assembler workflow
- Assembler Class –creates, configures and starts all application subsystems; gracefully stops all subsystems on application shutdown and releases all resources

- Unit Tests
- Integration Tests
- Behavioral Tests
- Acceptance Tests
- WTF Tests, etc.

“Database” is a Detail



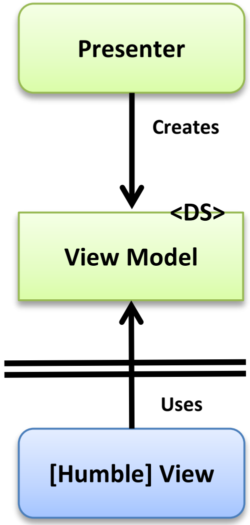
- The organizational structure of data (Data Model) is architecturally significant
- The technologies and systems that move data on and off a rotating magnetic surface are not

“Database” means “Persistent Storage” and includes:

- Bona fide databases (Oracle, mySQL, etc.)
- File Systems
- File storage formats (INI, XML, JSON, PNG,PDF, etc.)

Example: How do you persist Application Settings ?

UI (Web) is a Detail



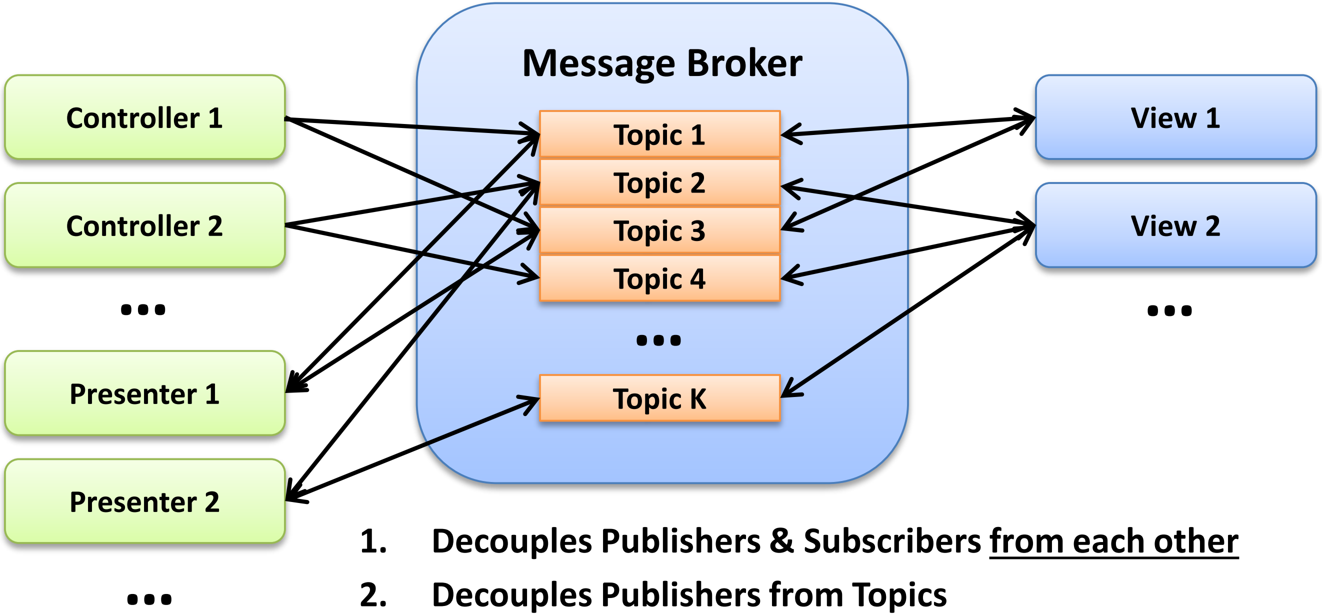
Once MVC Views are split into Presenters and Humble Views, the latter become a Detail:

- **Humble Views contain no business logic**
- **View Models are simple data structures containing string values and tags for each UI control/indicator**
- **Rendering View Model data structure becomes a trivial task using any available Web-based technology**

Deferring decision on Views implementation may save substantial time when starting the project and makes the design more scalable

Note: LabVIEW resists splitting UI code into Presenter/Control/View classes

Message Broker



Use Message Broker to decouple Humble Views from Presenters & Controllers at runtime …

1. Decouples Publishers & Subscribers from each other
2. Decouples Publishers from Topics
3. Supports 1:1, 1:M or N:M Message Routing
4. Objects may Publish and/or Subscribe to multiple Topics
5. Removes constraints on Object Instantiation Order

Frameworks are Details

“Good architectures are centered on use cases so that architects can safely describe the structures that support those use cases without committing to frameworks, tools, and environments.”, p.198

How do we protect our Applications from Frameworks ?!

“Architectures should not be supplied by frameworks. Frameworks are tools to be used, not architectures to be conformed to. If your architecture is based on frameworks, then it cannot be based on your use cases.”, p.197

A Framework or a Reuse Library ?

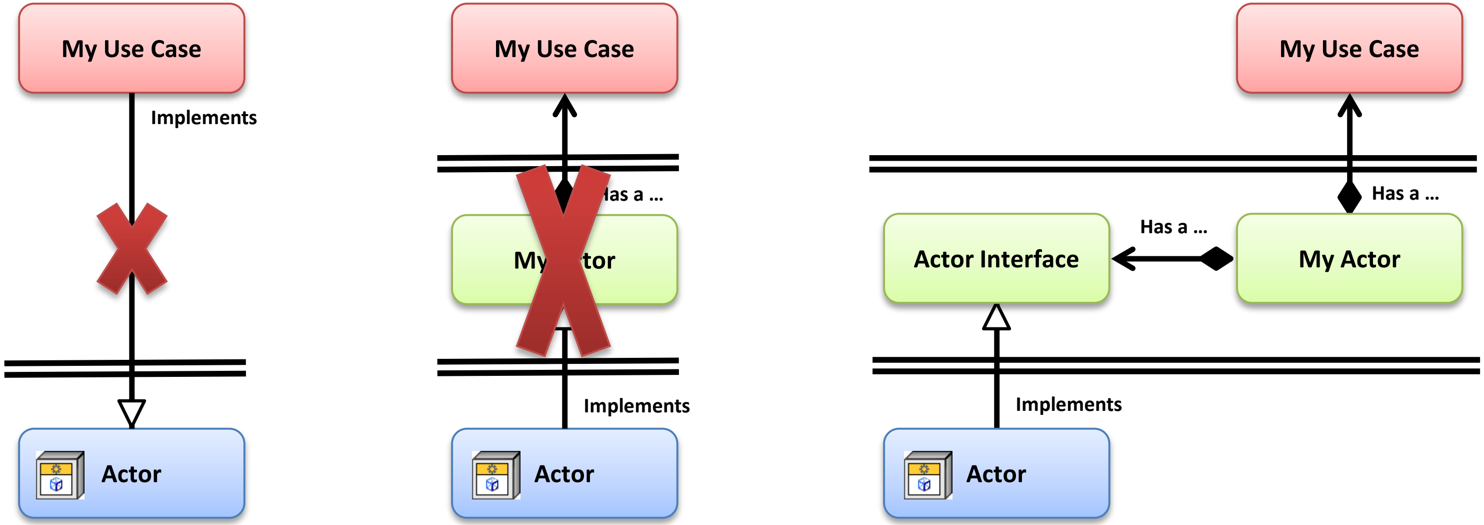
Frameworks have key distinguishing features that separate them from normal Libraries:

- **Inversion of Control**: In a framework, unlike in libraries or in standard user applications, the overall program's flow of control is not dictated by the caller, but by the framework
- **Extensibility**: A user can extend the framework – usually by selective overriding – or programmers can add specialized user code to provide specific functionality
- **Non-modifiable Framework Code**: The framework code, in general, is not supposed to be modified, while accepting user-implemented extensions. In other words, users can extend the framework, but cannot modify its code

Software Framework, Wikipedia

But, either way, the Dependency Rule should be observed …

Protecting Your Code from the Framework



Violates Dependency Rule

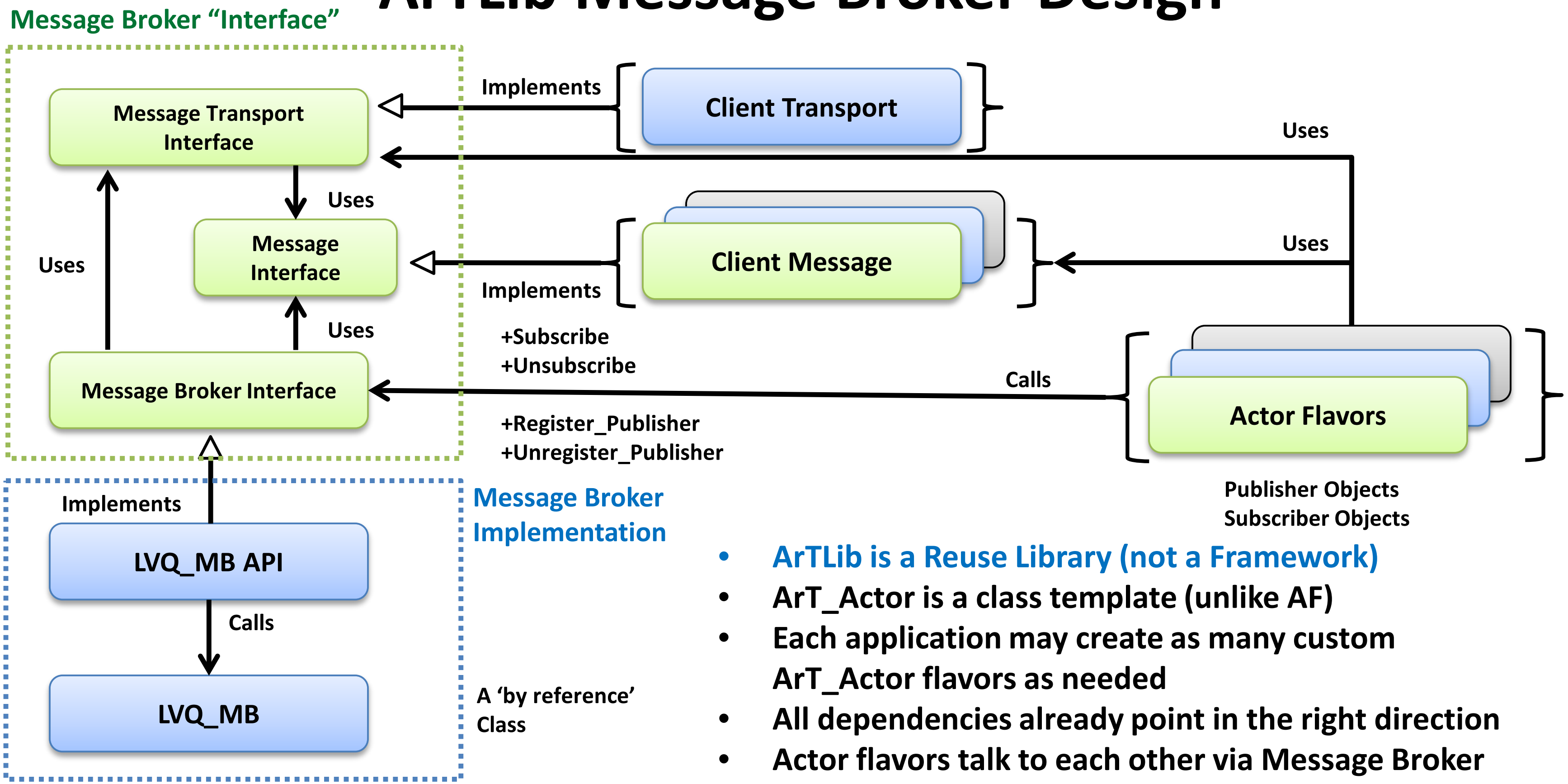
Cannot be Customized for different Use Cases

Works OK but Adds Complexity

This is what Architects Do:

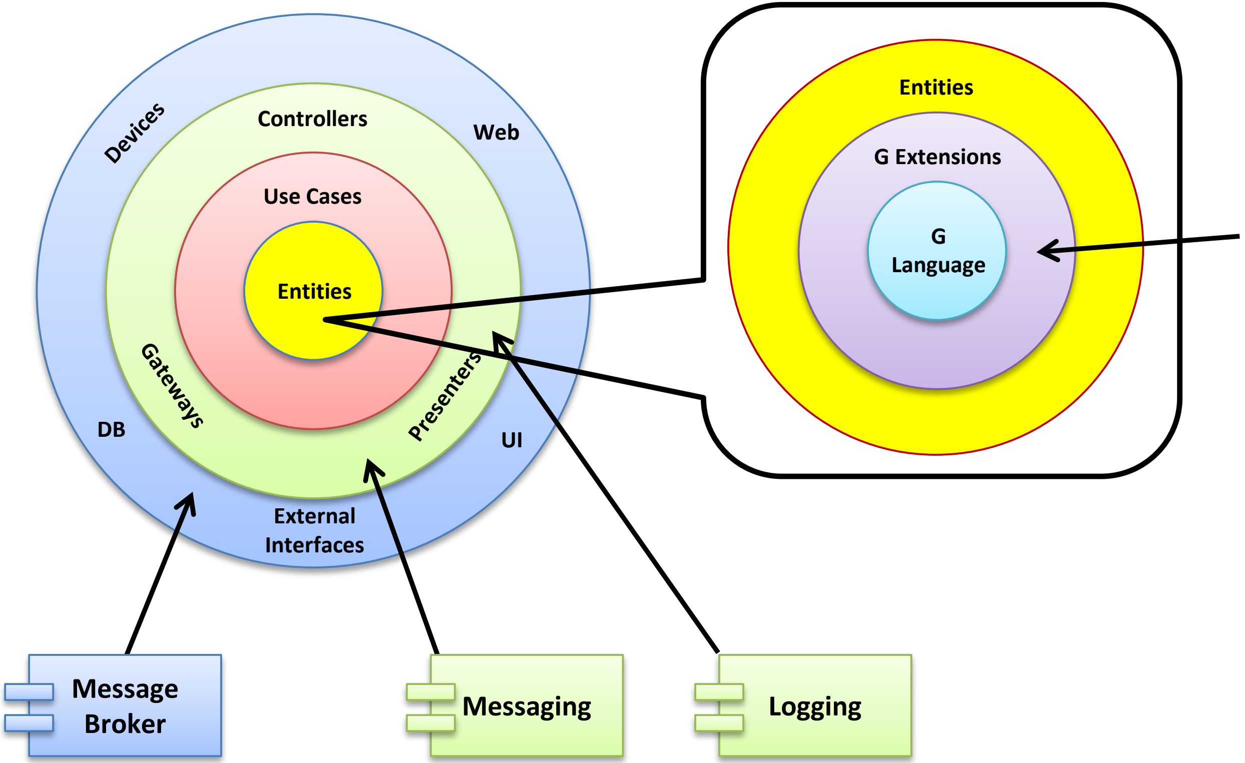
1. Draw Boundaries
2. Flip Dependency Arrows

ArTLib Message Broker Design



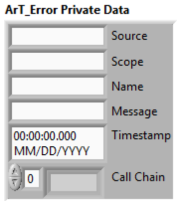
- ArTLib is a Reuse Library (not a Framework)
- ArT_Actor is a class template (unlike AF)
- Each application may create as many custom ArT_Actor flavors as needed
- All dependencies already point in the right direction
- Actor flavors talk to each other via Message Broker

Two More Layers



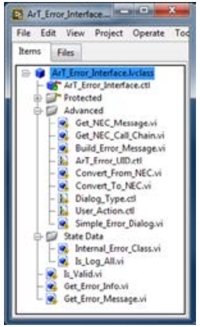
Since Error Handling
Permeates all Layers

Error Handling



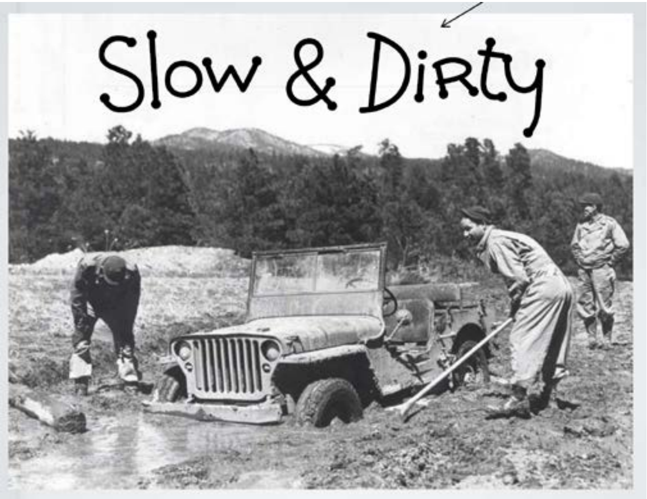
ArT_Error Private Data

	Source
	Scope
	Name
	Message
00:00:00.000 MM/DD/YYYY	Timestamp
0	Call Chain



```
ArT_Error_Interface.lvclass
  ArT_Error_Interface.ctl
  Protected
  Advanced
    Get_NEC_Message.vi
    Get_NEC_Call_Chain.vi
    Build_Error_Message.vi
    ArT_Error_UID.ctl
    Convert_From_NEC.vi
    Convert_To_NEC.vi
    Dialog_Type.ctl
    User_Action.ctl
    Simple_Error_Dialog.vi
  State Data
    Internal_Error_Class.vi
    Is_Log_All.vi
  Is_Valid.vi
  Get_Error_Info.vi
  Get_Error_Message.vi
```

Slow and Dirty



© Codemanship Ltd 2011

Slow and Dirty
Jason Gorman
Agile Wave Blog, 2011

- **Project success won't give you time to clean-up design later on – delivering the next "best-ever" version always becomes a top priority**
- **At startups, to survive the race, Architects must insulate themselves from never-ending prototype design changes by applying SOLID Principles to ensure uber-quick feature turn-around times**

> ***The only way to go fast, is to go well***
>
> *— Uncle Bob*

Summary : General

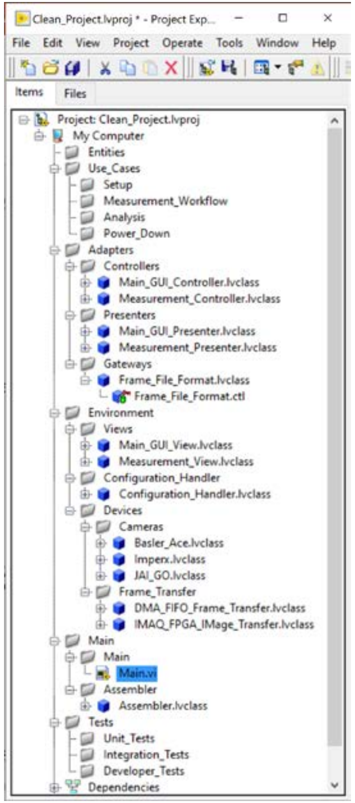
Clean Architecture is a mindset - use it from project start without investing upfront in an elaborate infrastructure …

Typical LabVIEW Projects start small with no/few formal Requirements.
Design steps :

- Use SOLID Principles to design and implement all classes
- Leave as many options open as possible, for as long as possible
- As application grows, start using Package Design Principles to group cohesive classes into Components
- Once having formal Use Cases – start arranging components on a per Use Case basis
- Always keep in mind the Dependency Rule and that adding Architectural Boundaries to existing code later on can be very expensive

Summary: How To Create Architectural Boundaries

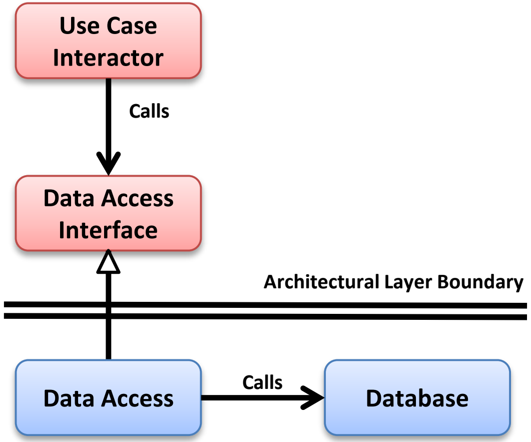
- Create virtual folders for each Architectural Layer:
  - Entities
  - Use Cases
  - Adapters
  - Environment
  - Main
  - Tests
- Keep classes in their Architectural Layer virtual folders
- Choose Class Names appropriate for the Layer:
  - /Use_Cases/Calibrate_HW
  - /Use_Cases/Calibrate_HW_Actor (no Actors allowed in Use Case Layer)
- Create a VI Analyzer Test to check for code dependencies pointing to outer Layers



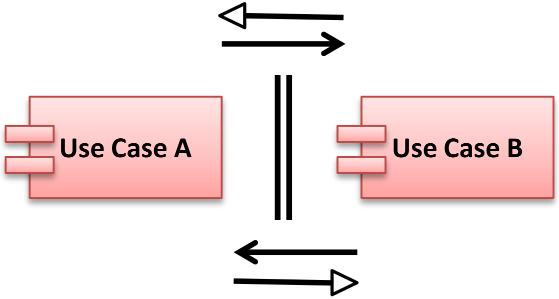
```
Project: Clean_Project.lvproj
  My Computer
    Entities
    Use_Cases
      Setup
      Measurement_Workflow
      Analysis
      Power_Down
    Adapters
      Controllers
        Main_GUI_Controller.lvclass
        Measurement_Controller.lvclass
      Presenters
        Main_GUI_Presenter.lvclass
        Measurement_Presenter.lvclass
      Gateways
        Frame_File_Format.lvclass
          Frame_File_Format.ctl
    Environment
      Views
        Main_GUI_View.lvclass
        Measurement_View.lvclass
      Configuration_Handler
        Configuration_Handler.lvclass
      Devices
        Cameras
          Basler_Ace.lvclass
          Imperx.lvclass
          JAI_GO.lvclass
        Frame_Transfer
          DMA_FIFO_Frame_Transfer.lvclass
          IMAQ_FPGA_IMage_Transfer.lvclass
    Main
      Main
        Main.vi
      Assembler
        Assembler.lvclass
    Tests
      Unit_Tests
      Integration_Tests
      Developer_Tests
    Dependencies
```

Summary: How to Flip Dependency Direction

- Use Dependency Inversion Principle (DIP) to revert direction of a dependency



- Partial (one way) decoupling may be sufficient for Architectural Layer Boundaries



- Full (two way) decoupling is preferred for Use Case Boundaries

References

[1] Clean Architectures, Robert C. Martin, Prentice Hall, 2017

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

[3] Who Needs an Architect ?, Martin Fowler, IEEE Software

[4] GUI Architectures, Martin Fowler, martinFowler.com, 2006

[5] Implementing Clean Architectures – Frameworks vs. Libraries, Plainionist, 2019

[6] Slow and Dirty, Jason Gorman, Agile Wave Blog, 2011

My To Do List

1. **Re-arrange ArTLib classes into Components based on their Architectural Level (especially Error Handling Classes)**
2. **Figure out whether (and why) Actor Framework would create obstacles when being used on projects with Clean Architecture**
3. **Refactor ArTLib to use Native LabVIEW Interfaces instead of “Abstract Class” workaround**

Questions ?

Design Smells – The Odors of Rotting Software

1. **Rigidity** – The system is hard to change because every change forces many other changes to other parts of the system
2. **Fragility** – Changes cause the system to break in places that have no conceptual relationship to the part that was changed
3. **Immobility**– It is hard to disentangle the system into components that can be reused in other systems
4. **Viscosity** – Doing things right is harder that doing things wrong
5. **Needless Complexity** – The design contains infrastructure that adds no direct benefit
6. **Needless Repetition** – The design contains repeating structures that could be unified under a single abstraction
7. **Opacity** – It is hard to read and understand. It does not express its intent well

[1] Robert C. Martin, p.88