SAND2022-6848C

# Applying Clean Architecture to Legacy Software Systems

Nate Roehrig - nsroehr@sandia.gov

ASC S³C

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

3 1

# Applying Clean Architecture to Legacy Software Systems

2

1
- what is legacy software
    - applied to which specific system

2
- what is clean architecture
- why use it

3
- effectively apply clean architecture to a legacy system
- results
- can this be reproduced

GOAL

Legacy software systems, under active development, can be incrementally refactored into flexible, well-tested, sustainable software products

*but first, a little background…*

## background

20 years of software development experience at SNL

- 12 years of agile experience
- 13 software products

Process failures, product failures

- second system effect[1]

I have this crazy idea that software development is something I can be proud of because I was part of making it better

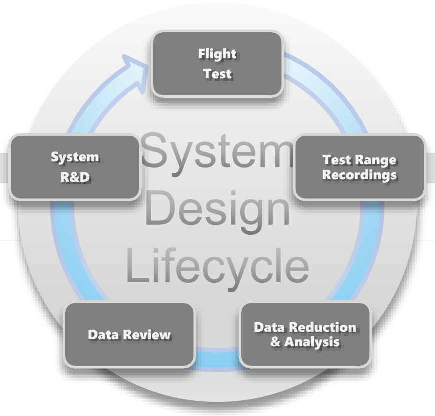
[1]Brooks Jr., Frederick P. *The Mythical Man-Month*

## legacy software – what is it

Deployed software, still in use

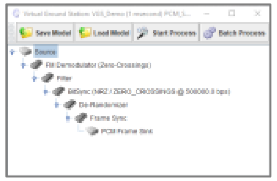
- Code rots - design smells
  - High Fragility – any change no matter how small is a breaking change
  - High Rigidity – any small change cascades into several much larger changes
  - Low Viscosity – it’s easier to do things wrong than it is to do them right
- Little Testing and Low Testability
  - Object under test cannot be created
  - Object cannot be separated from its dependencies
  - Global variables and other temporal couplings change system behavior
  - Abundant private interfaces (no entry points to test)
- It isn’t going away – since it’s being used, it will need to be changed
  - Redoing it from scratch doesn’t work

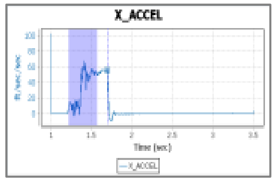
TAVS
Telemetry Analysis & Visualization Suite

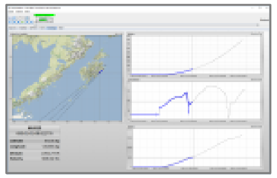


## Comprehensive tool for acquiring, extracting, and analyzing telemetry data throughout the System Design Lifecycle

- Three major components
  - Virtual Ground Station + Analysis & Visualization + Real Time
- Supports stockpile & development systems
  - LANL + LLNL + KCP
  - SMDC + NSWC + MDA + APL + Dynetics + Lockheed + Draper + Kratos
- Underlying capabilities are used in Citadel: DataSEA & SEDS
- 17 year old+ Desktop Application written in Java
  - 330,000 lines of code - 100s of users
  - 2017: Ad hoc software development process, no dedicated team, no CI

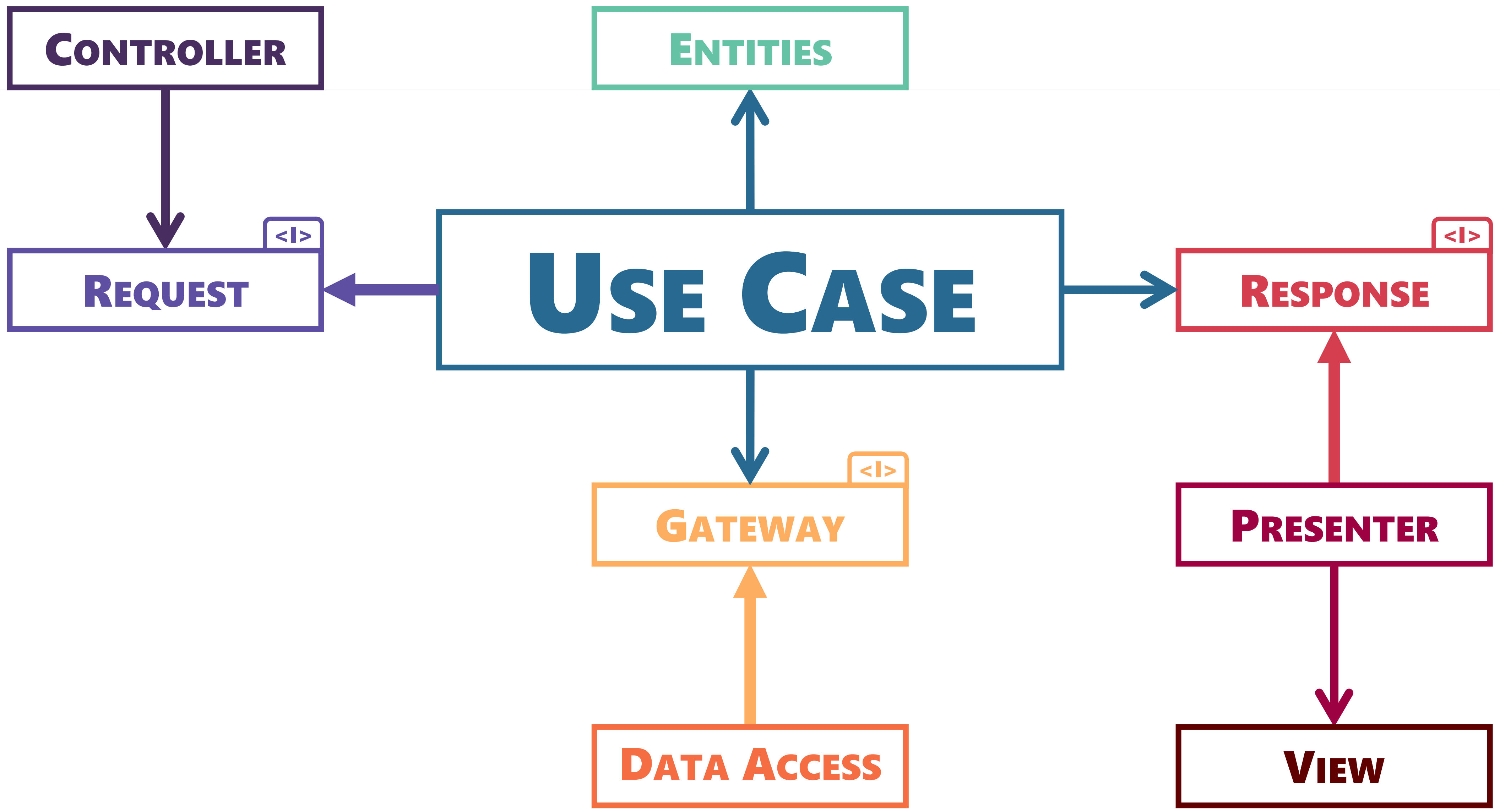






TAVS is a software tool to be used to acquire, visualize and analyze telemetry data.
Telemetry data is of little value until it is plotted and analyzed by flight test engineers.

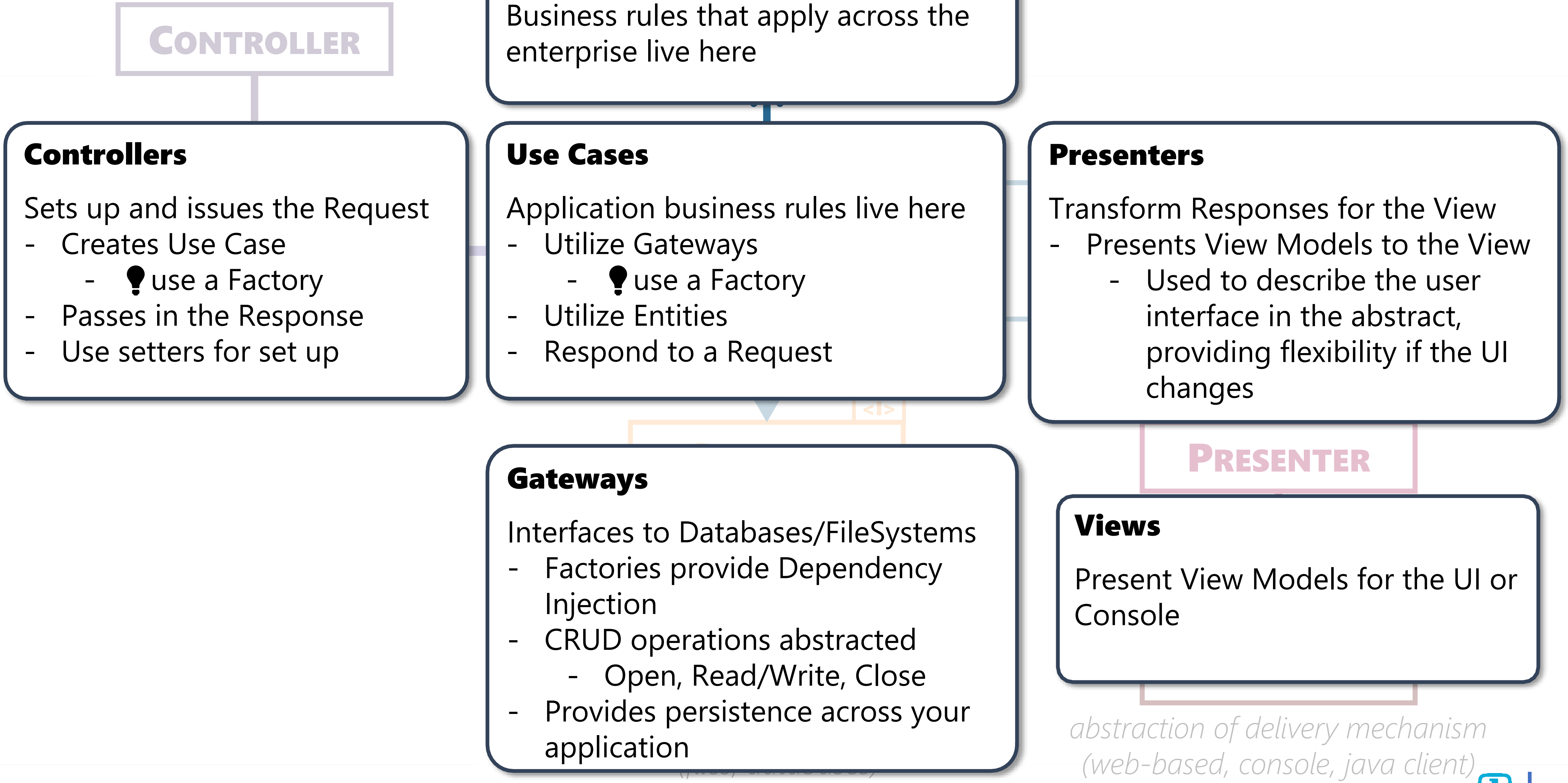
# Clean Architecture
*overview*



*abstraction of data providers (files, databases)*

*abstraction of delivery mechanism (web-based, console, java client)*

## Clean Architecture
### controllers


CONTROLLER
PRESENTER
<|>
abstraction of delivery mechanism
(web-based, console, java client)

**Entities**

Business rules that apply across the enterprise live here

**Controllers**

Sets up and issues the Request
- Creates Use Case
  - use a Factory
- Passes in the Response
- Use setters for set up

**Use Cases**

Application business rules live here
- Utilize Gateways
  - use a Factory
- Utilize Entities
- Respond to a Request

**Presenters**

Transform Responses for the View
- Presents View Models to the View
  - Used to describe the user interface in the abstract, providing flexibility if the UI changes

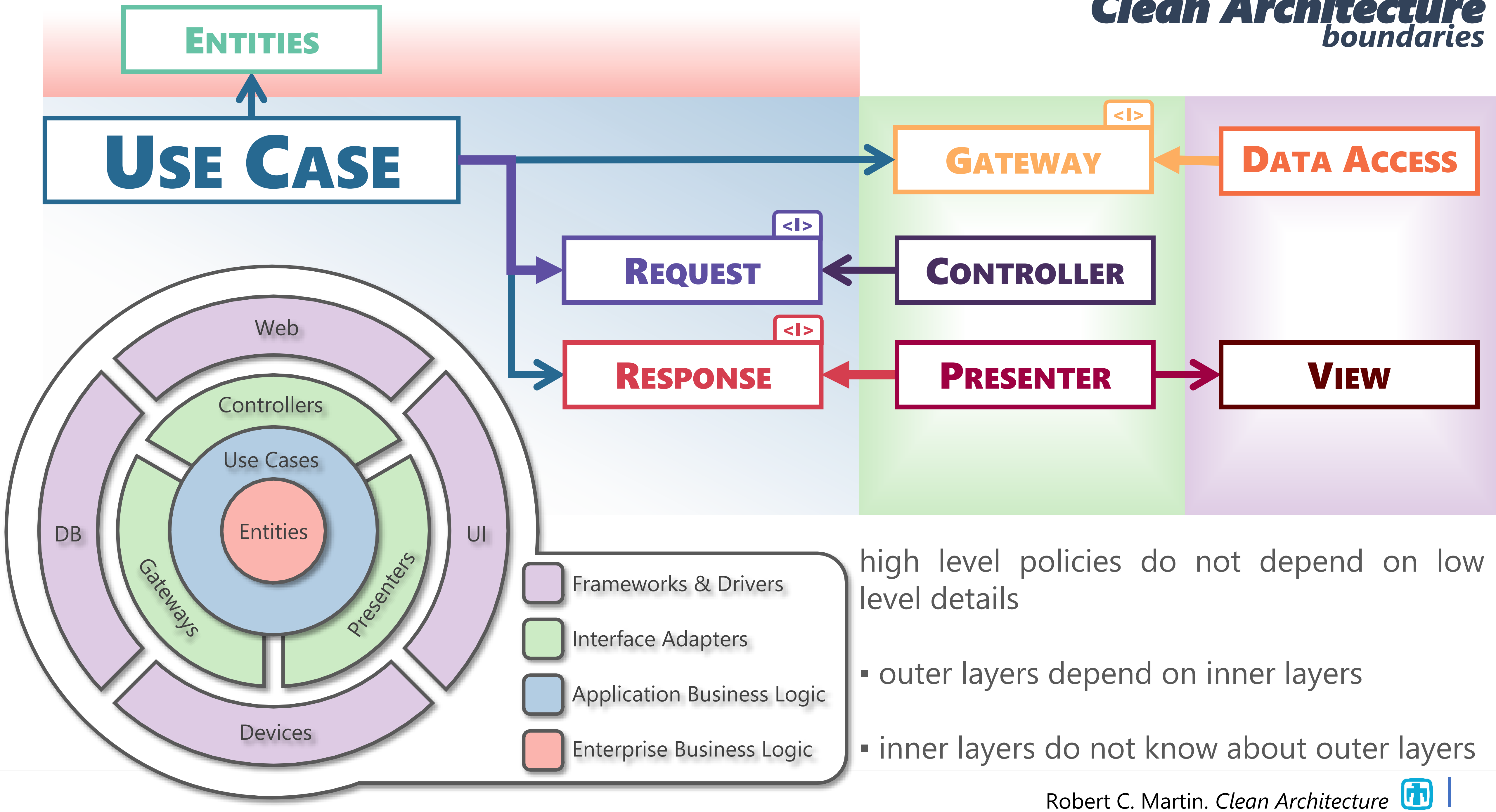
**Gateways**

Interfaces to Databases/FileSystems
- Factories provide Dependency Injection
- CRUD operations abstracted
  - Open, Read/Write, Close
- Provides persistence across your application

**Views**

Present View Models for the UI or Console

## Clean Architecture boundaries



| | |
|---|---|
| 🟪 | Frameworks & Drivers |
| 🟩 | Interface Adapters |
| 🟦 | Application Business Logic |
| 🟥 | Enterprise Business Logic |

high level policies do not depend on low level details

- outer layers depend on inner layers
- inner layers do not know about outer layers

Robert C. Martin. *Clean Architecture*

So why would anyone want to use Clean Architecture

If we consider for a moment…

- The primary value of software is that it remains **soft**
- The secondary value is that it is **correct**

Clean Architecture allows for deferring decisions

- Value is placed on flexibility and agility, not on specific technologies like DataBases or User Interfaces
  - YAGNI
- Focus is placed on the behavior central to our system, our business rules – our business value
  - This is why we're automating, this is why we're creating software

**soft**

Clean Architecture is inherently testable

- tests can easily mock interfaces and verify behaviors

**correct**

# application – approach

**Mindset**

- Context – during Planning
  - Helps establish scope = better estimation
  - Customer needs a new feature
    - New feature extends legacy behavior?
    - New feature can be an isolated use case (mostly non-legacy)
  - Customer found a bug
- Principles
  - Test First Design
  - Vertical slice of capability
    - We want to demo this
  - SOLID[1]
- Toolkit
  - Ping-Pong TDD & BDD
  - Mocking framework
  - Clean Code[1] – coding standards
  - Design patterns

**Adding Features to Existing Legacy Code**

1. Plan
2. Locate & Characterize
3. Refactor towards Use Cases
4. Integrate
5. Verify

**Adding New Use Cases**

1. Use BDD for Use Case (acceptance test)
2. Integrate
3. Verify

**Fixing Bugs**

1. Locate & Characterize
2. Refactor
   1. Keep in Clean
   2. Consider next steps – groom backlog

[1]Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*

## Adding Features to Existing Legacy Code – Detailed Look

1. Plan [🎩PO + SM🎩] - 💡 make sure that the user stories capture acceptance criteria
2. Locate & Characterize
    1. Sprout methods when needed - Goal: 100% coverage (including conditionals)
3. Refactor – now that you've captured existing behavior, advance to making it Clean
    1. Create the new Use Case - taking previous characterization tests form new expected behaviors
        1. Use test doubles (mocks) across your system boundaries
        2. Utilize BDD (double check you story)
            1. make use of Background and Givens to provide linkages
        3. Create needed Gateways - TDD actual gateway implementors
        4. Separate Entities when discovered
    2. Create Controller to supply this new Request - TDD
    3. Create Presenters that handle the new Responses - TDD
        1. Create Views
4. Integrate with the outer-most boundaries of your system
    1. Utilize new Controller to issue the Request
    2. Utilize new Views to present Responses
    3. Bind your Factories in Main
5. Verify
    1. Use Cases (and Entities) are the sole owners of business decisions; is your logic isolated?
        1. View Models should be used to prevent logical from residing in the User Interface
    2. Models should not pass through multiple boundaries (they have different reasons for changing)
        1. Do any of your models violate these conditions?

# results – lessons learn – process

Addressed issues with scope creep (inability to demo working code on Review day)

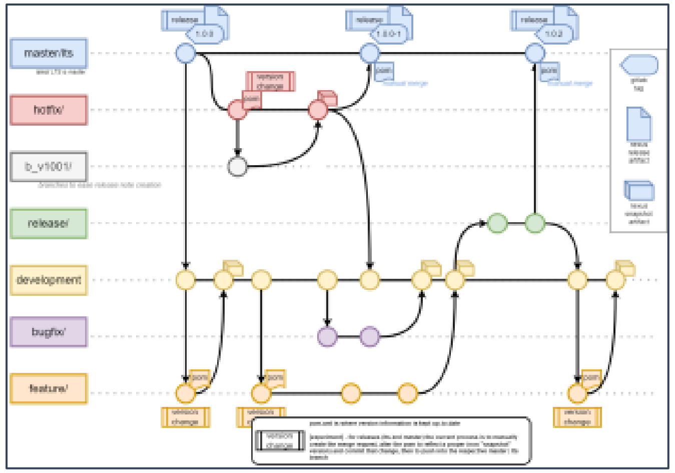
- one week sprints
- less than three point stories for any story on the Sprint Backlog
  - split them but still keep vertical slices

Pairing ⇨ Mobbing to help address issues with gaps in experience

- Later, to keep engagement high
  - Ping-Pong Testing

Release Process - from nothing but an Ant build

- GitFlow-ish
  - Long-Term Support releases were a driver (Hotfixes)
    - Don’t delete…deprecate
  - Gitlab + Maven + Jenkins + SonarQube + Nexus - thanks CEE! ╰(*°▽°*)╯



gitflow for TAVS

Reviews (feedback)

- Only able to demonstrate Done off installation build – a release candidate that is automagically created from an accepted merge commit in gitlab (this is done from ‘development’ branch)
  - Product Owner accepted work is always one step away from a release

# results – lessons learn – clean architecture

Data passed between boundaries (in Models) – should not be used across multiple boundaries

- this data has different reasons to change

Use Cases are Requests…rough start here

- use setters

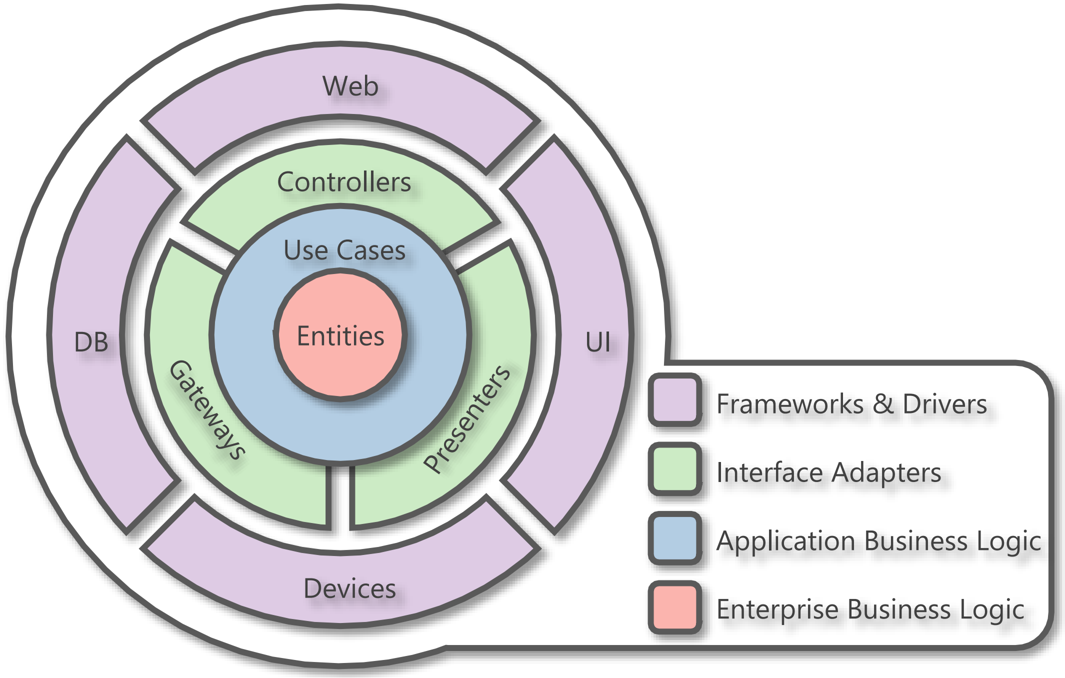
Main binds your Factories*

- your application is a plugin to Main

Gateways provide persistence

Don’t test the User Interface

Entities don’t use Gateways – they rely on Use Cases



# results – lessons learn – test doubles

Dependency inversion plays an important role in reducing the fragility of our systems; interfaces allows for this – mocking gives us the ability to test the expected behavior of these interfaces

Rules of thumb for test doubles (mocks):

- High level policies should not depend on low level objects
  - Use an interface
- Mock interfaces & verify expected behaviors
  - Don't mock everything
    - Objects under test need to be tested directly
    - Plain Old Data objects do not need to be mocked
  - When interfaces do not exist, Spy on the actual object and verify its behavior
- Mock Gateways, Factories, and Responses for your Use Cases
  - Including error handling Response for read/write errors from the Gateways
- Mock Requests for your Controllers - verify the Request is properly configured
- Mock Views for your Presenters

- Avoid directly testing any class further away then the test subject
- There is no need to test language features

## results – lessons learn – acceptance testing

Acceptance tests can be used to help determine Done for Use Stories

Rules of thumb for acceptance testing

- Acceptance criteria maps to the customers language far easier than unit tests
  - Consider using Cucumber or other BDD tools to more clearly demonstrate Done
  - When planning User Stories include Given, When, Then cases as Acceptance Criteria

- Cucumber Features
  - Utilize Backgrounds to provide mocks for Factories and Gateways

- Cucumber Scenarios should map directly to Acceptance Criteria

- Expect some difficulties with non-standard cucumber setups (step definitions that do not reside in a common director)
  - Specific Runners with the connective Glue will need to be applied when step definitions are used outside of the same Package

# results – lessons learn – standards & practices

## Test First Development (use TDD)

- No production code can be written or changed until there is a failing test to necessitate the change
- No-Hop Tests: the class under tests is the only non-mocked class
  - data structures do not need to be mocked

## Utilize Clean Architecture

- All I/O transactions take place behind a Gateway interface
- All business logic resides in Use Cases
- UI classes contain only the necessary functionality to pass user input into a Use Case via a Controller
  - No logic – utilize View Models
  - Implement Views

## Practice Clean Code

- Functions can have no more than three parameters
- Functions should not be longer than 5 lines
- Statements after conditionals or looping structures should not require seagulls: { }

## Deprecations for Public APIs

- @deprecated must include "since" attribute and be marked "forRemoval" = true (JDK 9+)

# results

- 55,201 lines of code added (includes tests, of course)
- 8,152 tests added – 6 tests a day since 2017 (the formation of the team)

jacoco coverage

| branch: | development | classico* | |
|---|---|---|---|
| **instruction** | 22% | 8% | 14% |
| **branch** | 20% | 8% | 12% |
| **complexity** | 24% | 7% | 17% |
| **line** | 22% | 7% | 15% |
| **method** | 31% | 9% | 22% |
| **class** | 35% | 9% | 26% |

sonarqube static analysis

| branch: | development | classico* | |
|---|---|---|---|
| **bugs** | 294 | 613 | -319 |
| **vulnerabilities** | 0 | 24 | -24 |
| **code smells** | 10806 | 14622 | -3816 |
| **coverage** | 22% | 7.3% | 14.7% |
| **duplications** | 5.1% | 8.6% | -3.5% |

*static analysis has only been considered as qualifying criteria as of 2021*

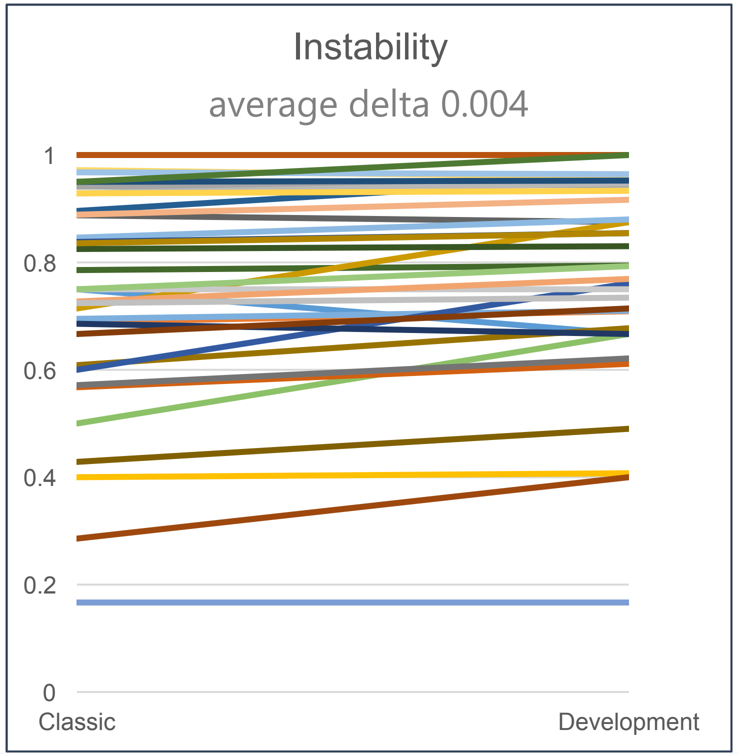
*"classico" was not the same SHA as the other metrics – it was a slightly later maven-ized build

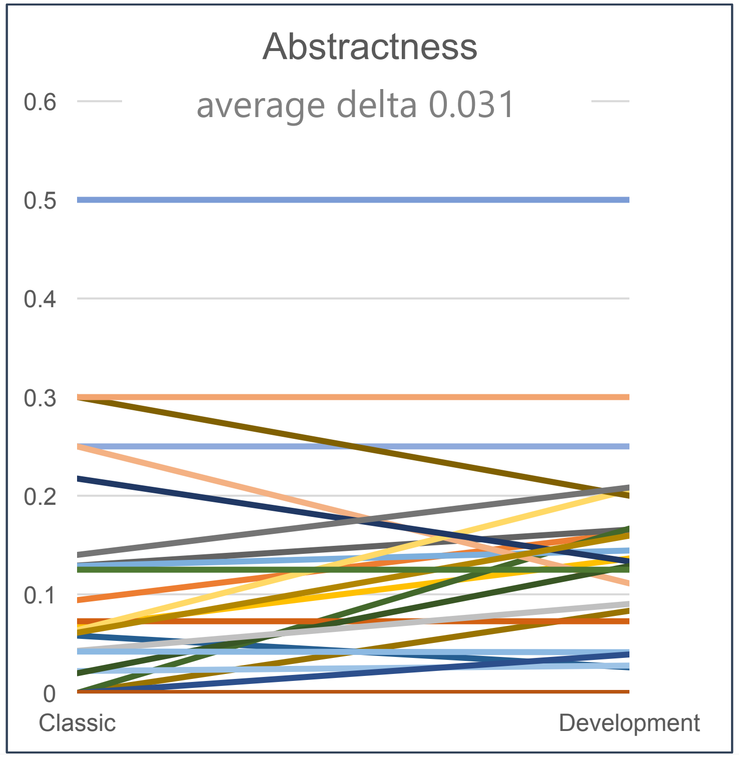
# stability & abstractness metrics

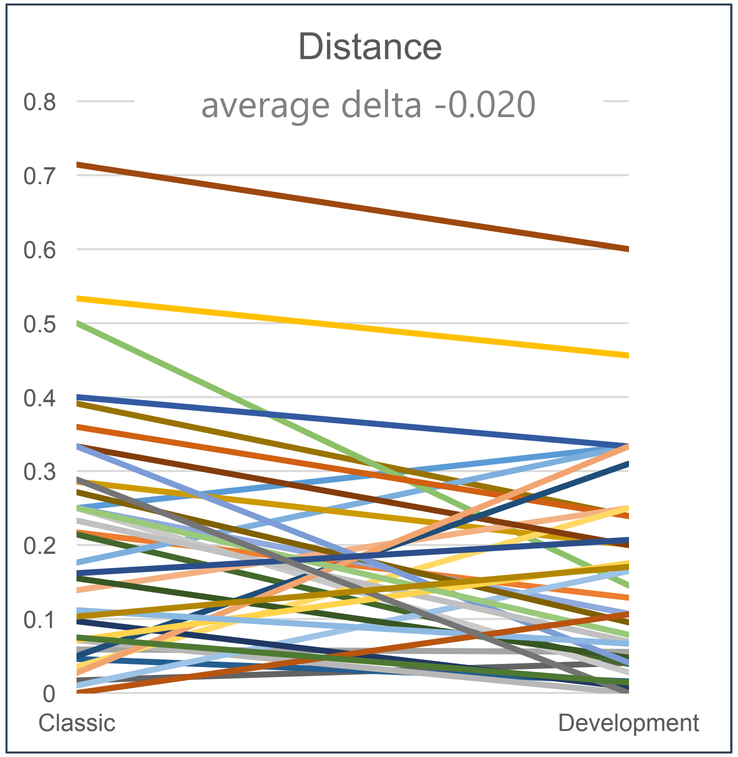
Stable Dependencies Principle – modules that are designed for change (low-level details) should be easiest to change [*instability* below]

Stable Abstractions Principle – high-level policies should not depend on low-level details [*abstractness* below]

Distance is a indicator of how far away from the ideal “main sequence” the component is: maximally stable and abstract [0,1] or maximally instable and concrete [1,0]







## reproducing these results – prerequisites

Firm understanding of Agile Manifesto and its Guiding Principles[1]

- Standing on the shoulders of giants
  - eXtremePrograming + Scrum
    - Scrum Master, Product Owner, and Developers – a team
    - Team room – a place
    - Inspect and adapt mindset
    - XP development practices
      - user stories, acceptance criteria, continuous integration, sustainable pace…
  - SOLID – development principles

1- https://agilemanifesto.org/principles.html

## reproducing these results

Keep Learning

- Watch Clean Coder Videos
- Utilize your Retrospectives

Keep Practicing

- Get involved
    - form a community
    - practice TDD on simple problems – hold some coding dojos

# gauge

**Doing well**

- The team begins to think in Use Cases
- Acceptance criteria is part of your user stories (given, when, then)
- Iteration cycles start to become shorter
    - But focus is still placed on vertical slices of capability allowing demonstration at Reviews
- Increased re-usability
    - Gateway implementors, Presenters become more abstract

**Caution**

- You start writing production code without a test and push it into the repo
- User Interface starts making decisions again
    - Because it’s just easier, all the data is right there*!*
- You stop pairing and start making excuses, you bring back code reviews
- You start blaming Scrum
    - Instead of using it to highlight the areas you need to improve

# resources

Robert Martin (Uncle Bob)

- *Agile Software Development, Principles, Patterns, and Practices*
- *Clean Code: A Handbook of Agile Software Craftsmanship*
- *Clean Architecture: A Craftsman's Guide to Software Structure and Design*
- Clean Code videos
    - At SNL – TEDS courses for each video (thanks, Manoj! \\(¯︶¯*\\))
    - https://cleancoders.com/

Kent Beck - *Extreme Programming Explained: Embrace Change*

Martin Fowler - *Refactoring: Improving the Design of Existing Code*

Michael Feathers - *Working Effectively with Legacy Code*

Emily Bache – agile coach: coding dojos + code katas

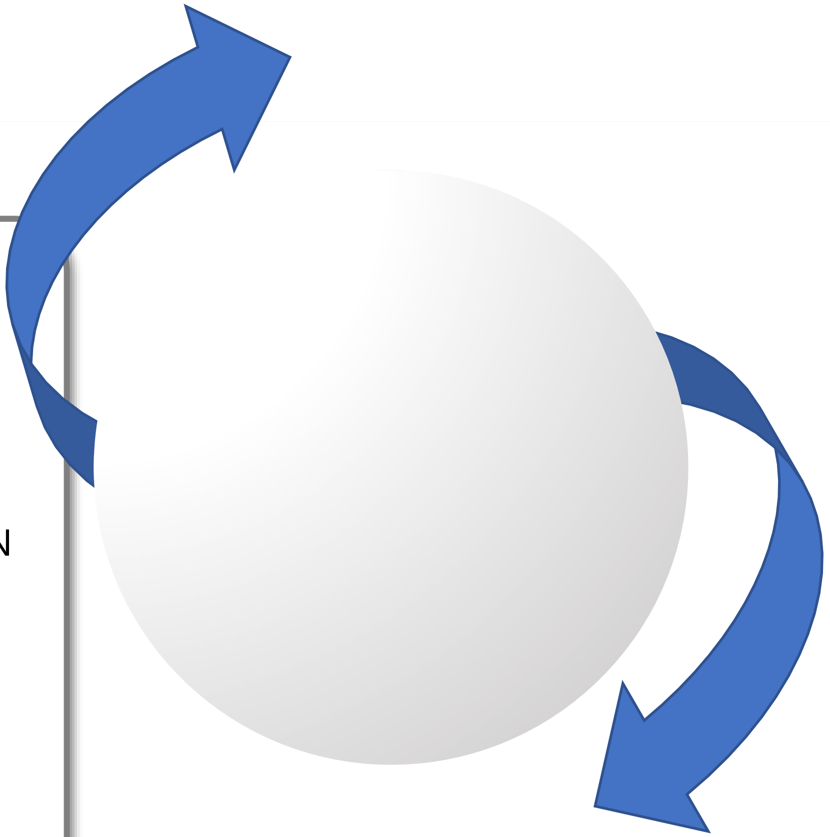
- https://github.com/emilybache

SNL Clean Code Community: cleancode-community@sandia.gov

Scrum Guide: https://www.scrum.org/resources/scrum-guide

## Rules of Ping Pong TDD

1. Write the least amount of code to get a test to fail RED
   A. Commit/push the failing test as Ignored – “Ping”
2. Next developer must:
   A. Get the test to pass with the least amount of production code GREEN
      i. Run all the tests – get all the tests to pass
      ii. When all tests are passing, commit/push – “Pong”
   B. Refactor the code and tests using Design Principles REFACTOR
      i. Run all the tests
      ii. If passing, commit
   C. Repeat, starting at #1🔁



: RED: Ping ↷ GREEN: Pong ↶, REFACTOR: Repeat🔁

## backups

# agile principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

1- https://agilemanifesto.org/principles.html

# agile principles reinforced by

⚖ Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

- ⚙ Scrum - rapid feedback through the Sprint Reviews
- ⚙ Scrum - Product Owner manages (orders) Product Backlog - maximizing value for the customer and prioritizing work
- 💻 We practice 1 week Sprints
- 💻 Utilize Jenkins for Continuous Integration & Automation of Delivery to our Nexus
- 💻 Practice Clean Architecture to help maintain the primary value of software (keep it soft)

⚖ Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

- ⚙ Scrum feedback cycle Planning -> Review
- ⚙ Scrum Team includes the Product Owner - he's always around to facilitate communication about Story scope
- ⚙ Product Owner may cancel a Sprint (Re-Plan and start anew at any point within the Sprint)
- 💻 Product Owner invites the customers to the Review
- 💻 We practice 1 week Sprints - customers only need to wait, at most, a week

⚖ Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

- ⚙ Scrum Sprints help to ensure delivery of demonstrable software (for the Review)
- 💻 Releases can be done, at the end of any Sprint that the PO deems worthy - an accepted Sprint Backlog (releases are automated through maven and jenkins)
- 💻 We practice TDD, specifically ping-pong mobbing, to make sure it is "working"
- 💻 We practice 1 week Sprints

⚖ Business people and developers must work together daily throughout the project.

⚖ Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

- ⚙ Scrum roles include Developers that work together within a shared space - we have a Team Room (under covid: we have "virtual team rooms")
- ⚙ Scrum Sprints and Scrum Cycles help to motivate because the Team is held Accountable
- ⚙ Scrum Developers control the technical approach (it is not dictated by the PO or anyone else)
- 💻 We practice 1 week Sprints, ending in a Review, holding us accountable to the work we added to the Sprint Backlog

⚖ The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

- ⚙ Scrum Planning helps to communicate to the entire team the priority and scope of the work
- 💻 We practice TDD, specifically ping-pong mobbing, the entire team is present (and engaged) when any technical decision is made
- 💻Sometimes we scope work with User Stories...or a Given, When, Then construct (that enables the use of BDD via cucumber for acceptance testing)

⚖ Working software is the primary measure of progress.

- ⚙ Scrum - demonstrable software is ready for Review
- 💻 Demo latest snapshot at the Review
- 💻 TDD, specifically ping-pong mobbing, so we essentially have documented what "working" means (at least to the development team)
- 💻 CI process gives us feedback, including static code analysis quality gates

⚖ Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

- ⚙ the Scrum cycle reinforces the idea of constant pace (work is done and delivered via Sprints)
- ⚙ Scrum - Sprint Velocity helps to determine the amount of work in a given cycle - shining a light on where we are (attempting to defeat last-minute heroics)
- ⚙ Scrum Retrospectives give us a place to analyze how well we are developing software and/or teaming and adapt accordingly
- 💻 Clean Architecture guides our design, keeping the primary value of the software high; that it is soft (tolerates and facilitates on-going change)

⚖ Continuous attention to technical excellence and good design enhances agility.

- ⚙ Scrum enforces the development team dictates technical approach - the people involved in developing and delivering the product (and are accountable)own technical decision space
- 💻 Clean Architecture is our "good design"
- 💻 Clean Code Principles, reflected in our coding standards, help draw attention to low-level technical details
- 💻 CI - static analysis "code smells" (sonarqube) encourages us to follow standardized best practices for the java language (an unbiased 3rd party evaluation)
- 💻 CI - static analysis "vulnerabilities" helps us to better secure our code against possible threats
- 💻 CI - code coverage: 80% coverage (goal 100%) on all "new code" increasing agility by enabling future refactorability
- 💻 legacy code behavior is tested through characterization tests to at least 80% converge (goal 100%) initially to prevent regress but ultimately to facilitate refactoring to a better design

⚖ Simplicity--the art of maximizing the amount of work not done--is essential.

- ⚙ Scrum prioritized Product Backlog
- ⚙ Scrum cycles (Sprints) are too short for any Huge Design Upfront effort
- 💻 TDD principles: Red, Green, Refactor - only write enough of a test to fail, only write enough code to get the test to pass, refactor (reinforcing low complexity)
- 💻 Clean Code principles: single responsibility - classes should only have a single reason to change (reinforcing low complexity and high cohesion)
- 💻 Clean code standard: methods should be five lines or less and have no more than three arguments
- 💻 Clean Architecture allows us to test up to the endpoints of our system - reducing the need for bulky (non-isolating) system-level and integration-level tests
- 💻 CI - static analysis complexity measures - alert us when cyclomatic complexity reaches unacceptably high levels

⚖ The best architectures, requirements, and designs emerge from self-organizing teams.

- ⚙ under Scrum developers must be self-organizing - we organize daily with the Daily Scrum
- ⚙ all developers are treated equally, there are no specific roles for "architect" or "ux designer" or "tester", so the team must decide how to address these responsibilities
- 💻 TDD allows for the architecture to evolve organically instead of being dictated by "the architect"
- 💻 TDD, specifically ping-pong mobbing, is a practice that is designed to promote self-organization by requiring that everyone participate in code development

⚖ At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

- ⚙ Scrum Sprints involve a Retrospective where the team inspects what happened and designs experiments to adapt its practices and processes
- 💻 We hold a Sprint Retrospective every week