# Verification and Refactoring

Better Scientific Software Tutorial

Anshu Dubey

Argonne National Laboratory

ISC High Performance Conference

June 16, 2019

# License, citation, and acknowledgements

## License and Citation




- Requested citation: **Anshu Dubey, Verification and Refactoring, in Better Scientific Software Tutorial, ISC High Performance Conference, Frankfurt, Germany, 2019. DOI: https://doi.org/10.6084/m9.figshare.8242859**


## Acknowledgements


- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357

# Verification

# Verification

- Code verification uses tests
  - It is much more than a collection of tests
- It is the holistic process through which you ensure that
  - Your implementation shows expected behavior,
  - Your implementation is consistent with your model,
  - Science you are trying to do with the code can be done.

# Stages and types of verification

- During initial code development
  - Accuracy and stability
  - Matching the algorithm to the model
  - Interoperability of algorithms
- In later stages
  - While adding new major capabilities or modifying existing capabilities
  - Ongoing maintenance
  - Preparing for production

# Components of Verification

- Testing at various granularity
  - Individual components
  - Interoperability of components
  - Convergence, stability and accuracy
- Validation of individual components
- Testing practices
- Error bars
  - Necessary for differentiating between drift and round-off
- Selection of tests for coverage

# Test Definitions

- Unit tests
  - Test individual functions or classes
- Integration tests
  - Test interaction, build complex hierarchy
- System level tests
  - At the user interaction level

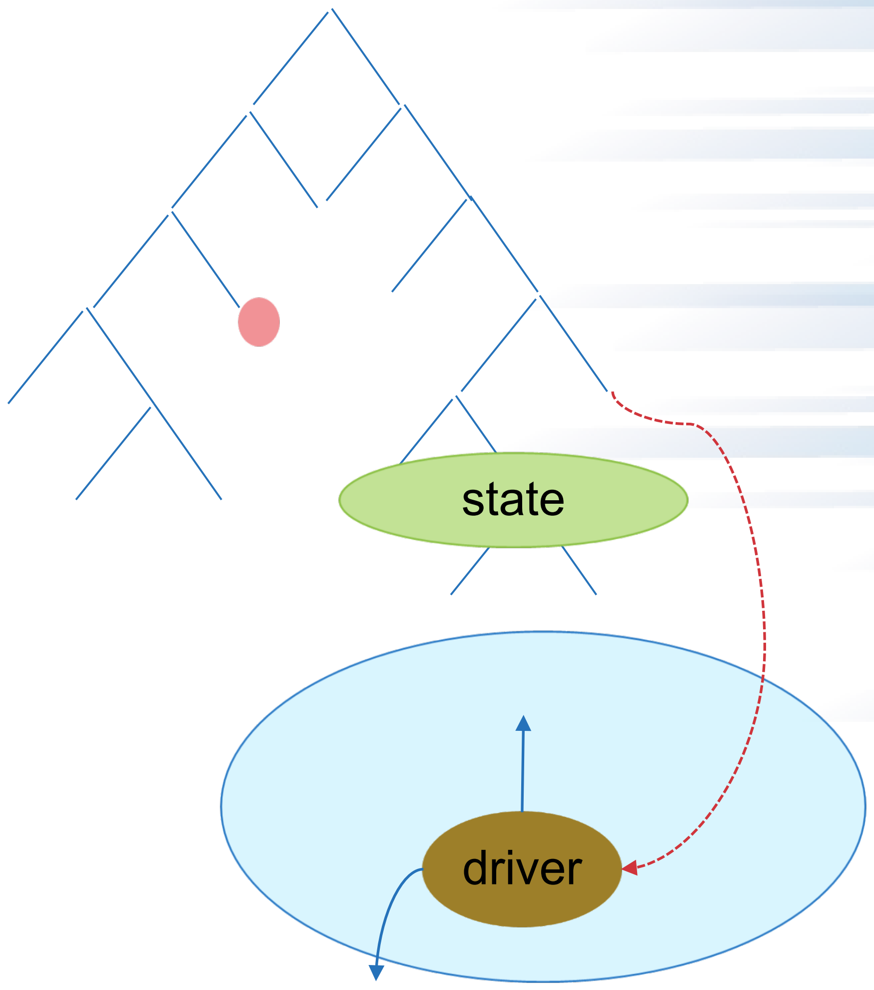
- Restart tests
  - Code starts transparently from a checkpoint
- Regression (no-change) tests
  - Compare current observable output to a gold standard
- Performance tests
  - Focus on the runtime and resource utilization

# Test Development

- Development of tests and diagnostics goes hand-in-hand with code development
  - Non-trivial to devise good tests, but extremely important
  - Compare against simpler analytical or semi-analytical solutions
- When faced with legacy codes with no existing tests
  - More creative approach becomes necessary
- Verify correctness
  - Always inject errors to verify that the test is working

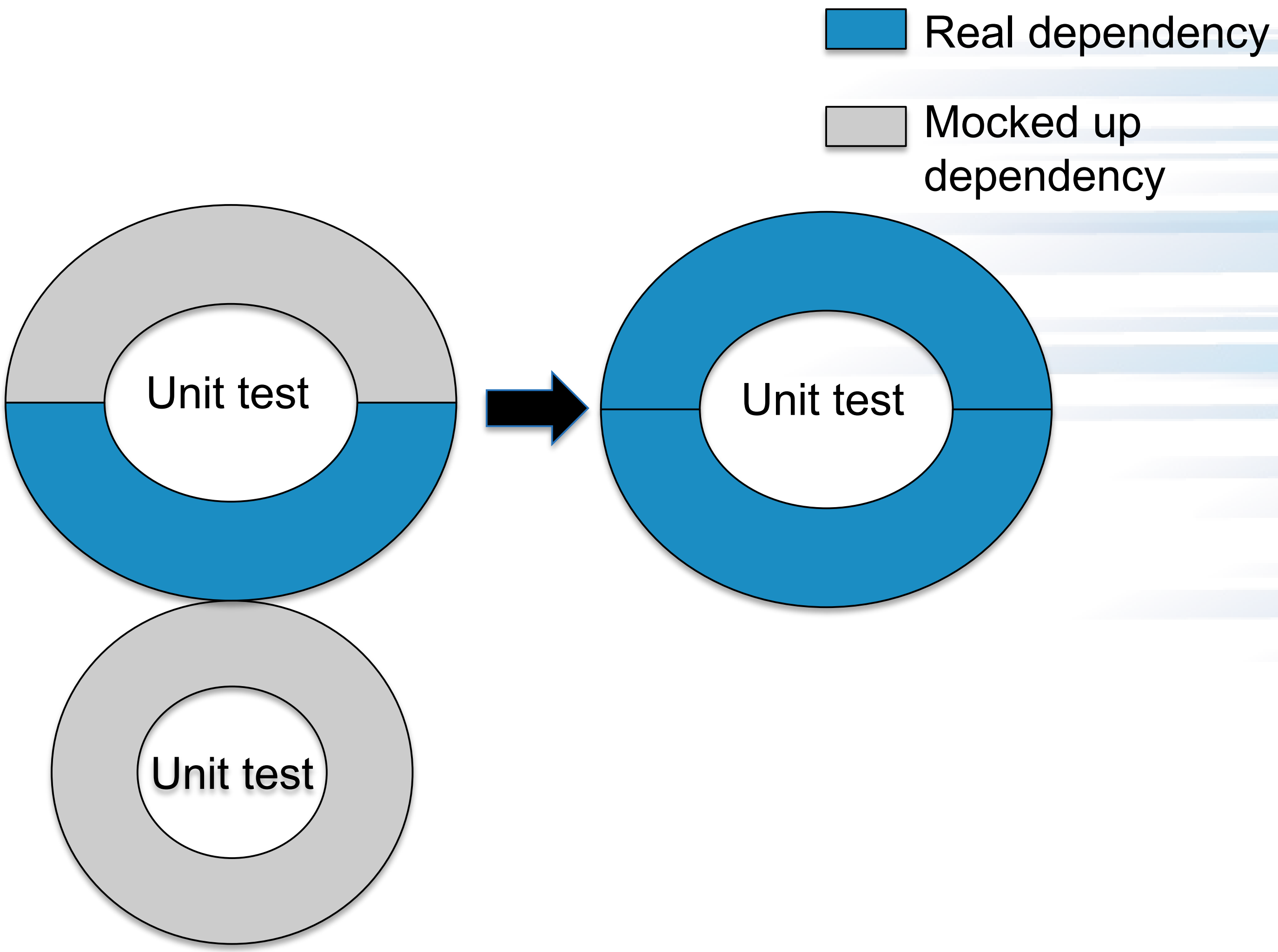
# Example from E3SM

- Isolate a small area of the code
- Dump a useful state snapshot
- Build a test driver
  - Start with only the files in the area
  - Link in dependencies
    - Copy if any customizations needed
- Read in the state snapshot
- Restart from the saved state



# Workarounds for Granularity

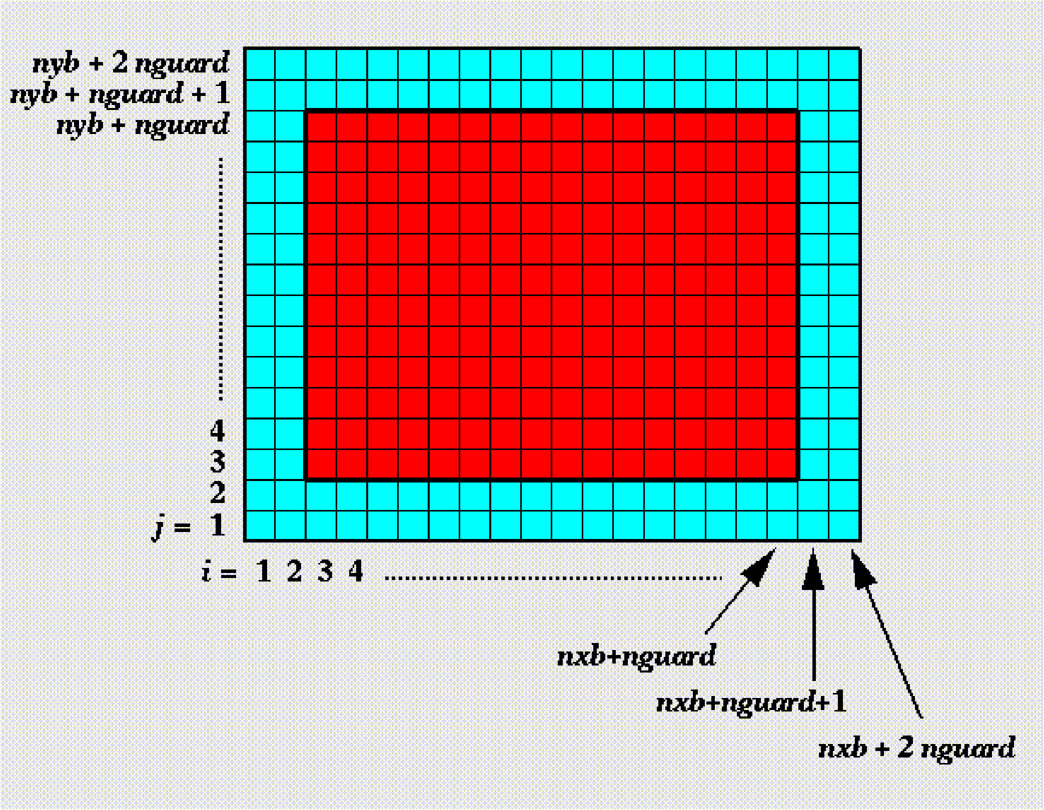
- Approach the problem sideways
  - Components can be exercised against known simpler applications
  - Same applies to combination of components
- Build a scaffolding of verification tests to gain confidence

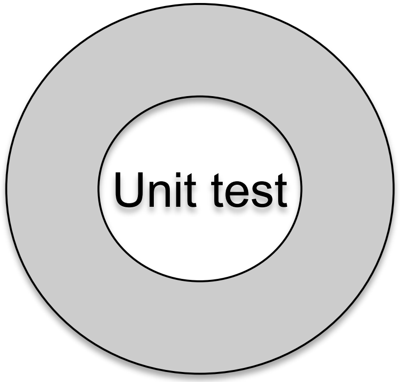


# Example from FLASH

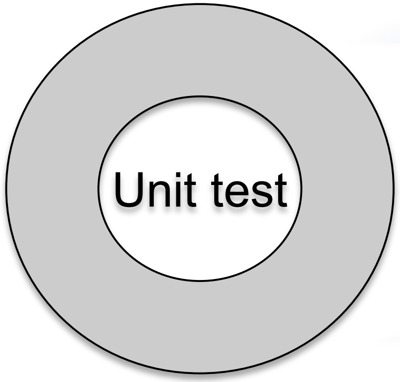
## Unit test for Grid

- Verification of guard cell fill
- Use two variables A & B
- Initialize A in all cells and B only in the interior cells (red)
- Apply guard cell fill to B



# Example from Flash



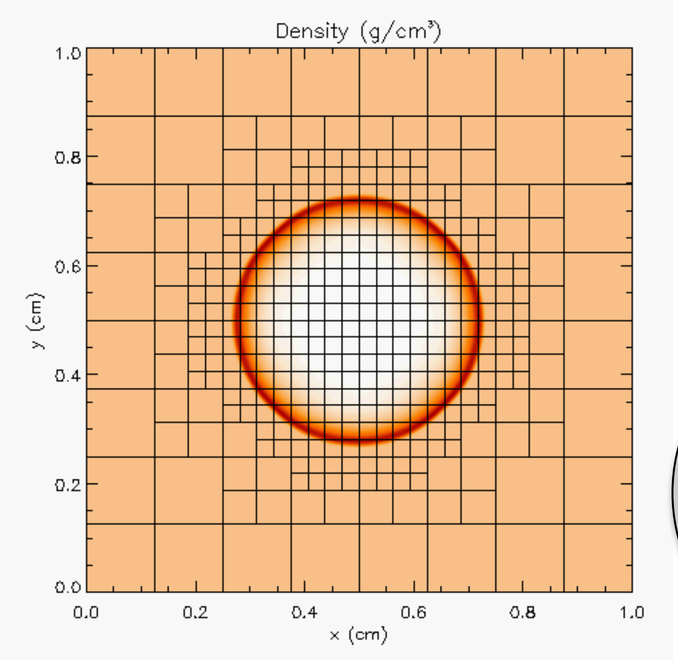

## Unit test for Equation of State (EOS)

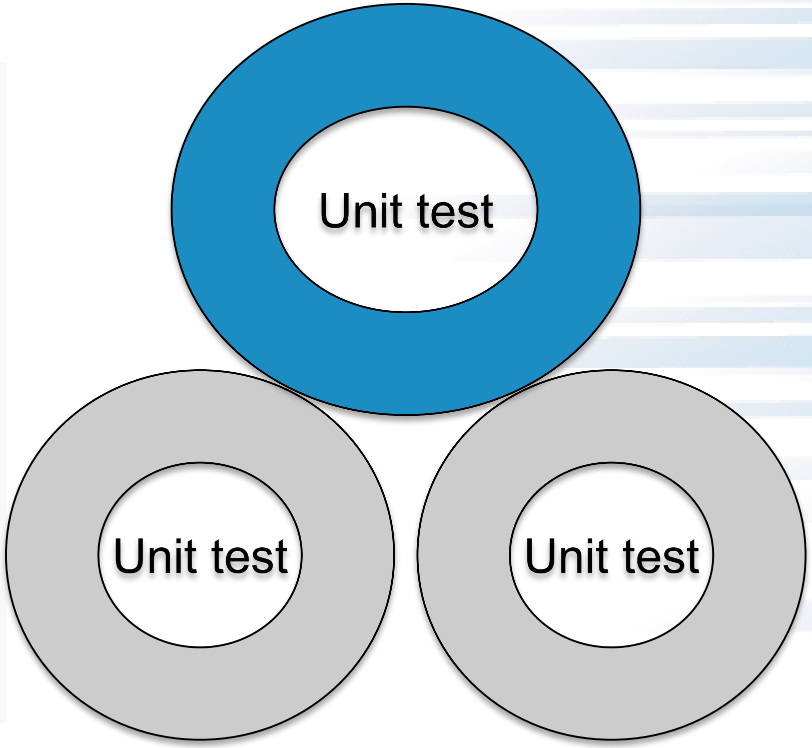
- Three modes for invoking EOS
  - MODE1: Pressure and density as input, internal energy and temperature as output
  - MODE2: Internal energy and density as input temperature and pressure as output
  - MODE3: Temperature and density as input pressure and internal energy as output
- Use initial conditions from a known problem, initialize pressure and density
- Apply EOS in MODE1
- Using internal energy generated in the previous step apply EOS in MODE2
- Using temperature generated in the previous step apply EOS in MODE3
- At the end all variables should be consistent within tolerance

# Example from FLASH

## Unit test for Hydrodynamics

- Sedov blast wave
- High pressure at the center
- Shock moves out spherically
- FLASH with AMR and hydro
- Known analytical solution




Though it exercises mesh, hydro and eos, if mesh and eos are verified first, then this test verifies hydro

**More testing needed for Grid using AMR**
**Flux correction and regridding**

# Example from FLASH

## Reason about correctness for testing Flux correction and regridding

IF Guardcell fill and EOS unit tests passed

- Run Hydro without AMR
  - If failed fault is in Hydro
- Run Hydro with AMR, but no dynamic refinement
  - If failed fault is in flux correction
- Run Hydro with AMR and dynamic refinement
  - If failed fault is in regridding

# Selection of tests

- Two purposes
  - Regression testing
    - May be long running
    - Provide comprehensive coverage
  - Continuous integration
    - Quick diagnosis of error
- A mix of different granularities works well
  - Unit tests for isolating component or sub-component level faults
  - Integration tests with simple to complex configuration and system level
  - Restart tests
- Rules of thumb
  - Simple
  - Enable quick pin-pointing

# Why not always use the most stringent testing?

- Effort spent in devising tests and testing regime are a tax on team resources
- When the tax is too high…
  - Team cannot meet code-use objectives
- When is the tax is too low…
  - Necessary oversight not provided
  - Defects in code sneak through
- Evaluate project needs
  - Objectives: expected use of the code
  - Team: size and degree of heterogeneity
  - Lifecycle stage: new or production or refactoring
  - Lifetime: one off or ongoing production
  - Complexity: modules and their interactions

# Test Selection

- First line of defense – code coverage tools  (demo later)
- Necessary but not sufficient – don't give any information about interoperability

- Build a matrix
  - Physics along rows
  - Infrastructure along columns
  - Alternative implementations, dimensions, geometry
- Mark <i,j> if test covers corresponding features
- Follow the order
  - All unit tests – including full module tests
  - Tests representing ongoing productions
  - Tests sensitive to perturbations
  - Most stringent tests for solvers
  - Least complex test to cover remaining spots

# Example

| | Hydro | EOS | Gravity | Burn | Particles |
|---|---|---|---|---|---|
| AMR | CL | CL | | CL | CL |
| UG | SV | SV | | | SV |
| Multigrid | WD | WD | WD | WD | |
| FFT | | | PT | | |

| Tests | Symbol |
|---|---|
| Sedov | SV |
| Cellular | CL |
| Poisson | PT |
| White Dwarf | WD |

- A test on the same row indicates interoperability between corresponding physics
- Similar logic would apply to tests on the same column for infrastructure
- More goes on, but this is the primary methodology



IDEAS productivity ECP EXASCALE COMPUTING PROJECT

# Regular Testing

- Part of ongoing verification
- Automating is helpful
- Can be just a script
- Or a testing harness

**Jenkins**
**C-dash**
**Custom**
(FlashTest)

- Essential for large code
  - Set up and run tests
  - Evaluate test results
- Easy to execute a logical subset of tests
  - Pre-push
  - Nightly
- Automation of test harness is critical for
  - Long-running test suites
  - Projects that support many platforms

# Refactoring

# Considerations

- Know why you are refactoring
  - Is it necessary
  - Where should the code be after refactoring
- Know the scope of refactoring
  - How deep a change
  - How much code will be affected
- Estimate the cost
  - Expected developer time
  - Extent of disruption in production schedules
- Get a buy-in from the stakeholders
  - That includes the users
  - For both development time and disruption

# Cost estimation

## When development and production co-exist

- Potential for branch divergence
- Policies for code modification
  - Estimate the cost of synchronization
  - Plan synchronization schedule and account for overheads
- Anticipate production disruption
  - From code freeze due to merges
  - Account for resources for quick resolution of merge issues

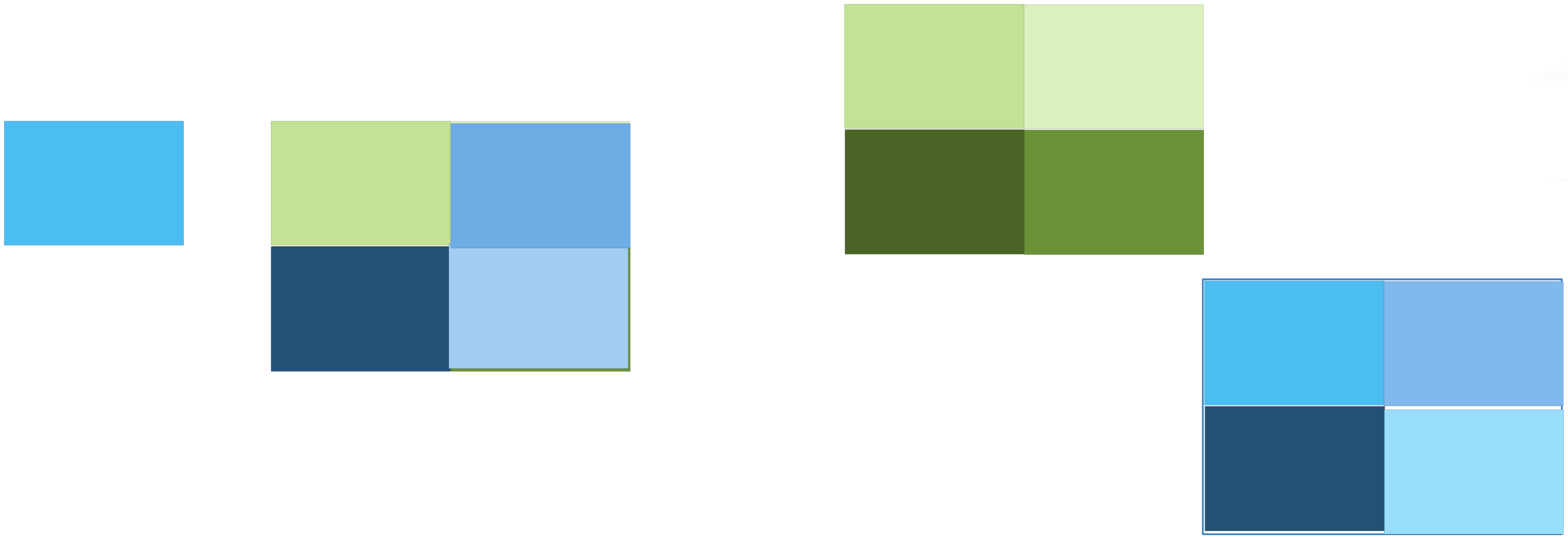
**This is where buy-in from the stake-holders is critical**

# Before Starting

- Know bounds on acceptable behavior change
- Know your error bounds
  - Bitwise reproduction of results unlikely after transition
- Map from here to there
- Check for coverage provided by existing tests
- Develop new tests where there are gaps

Incorporate testing overheads into refactor cost estimates

# On ramp plan



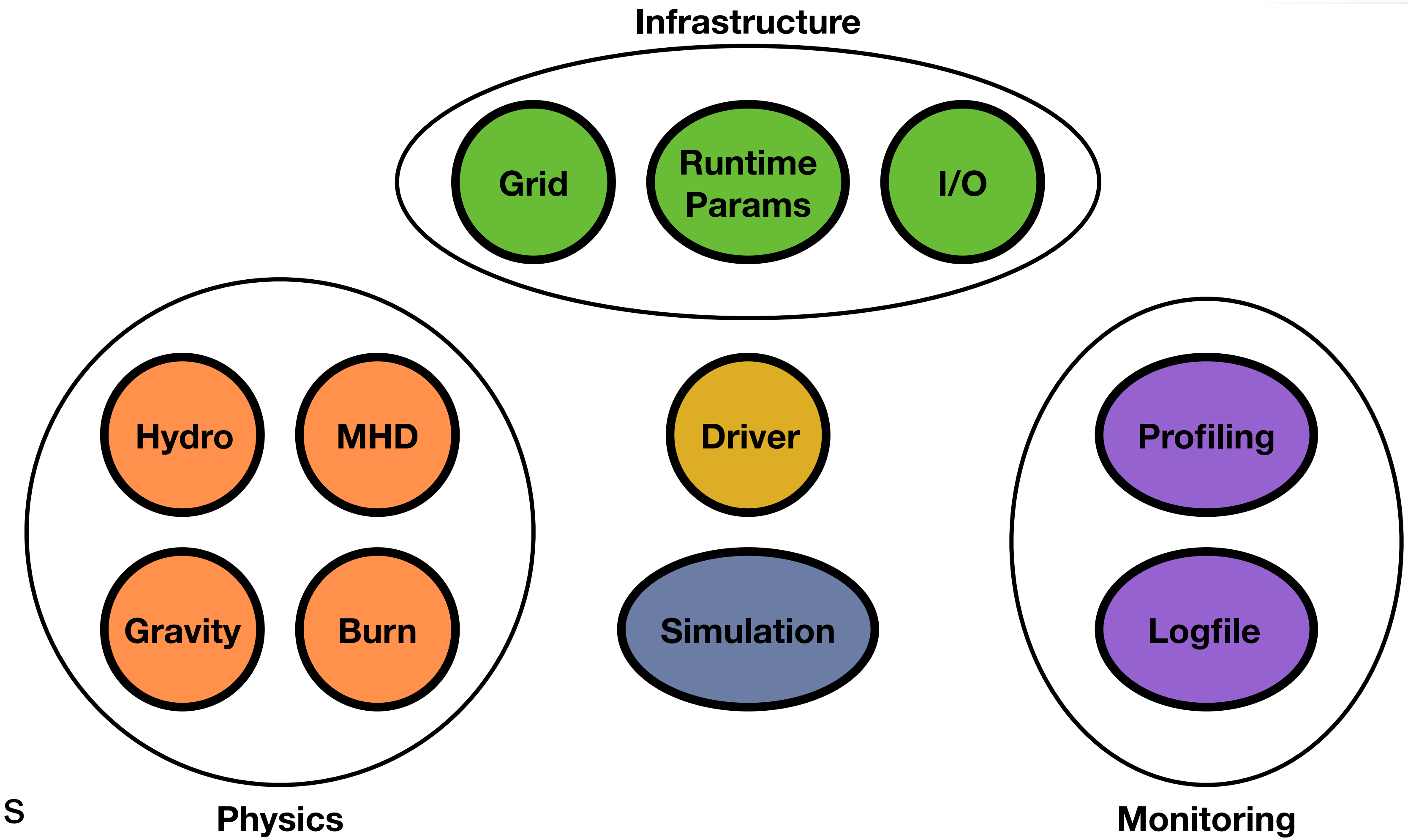
- Incrementally if at all possible
- Small components, verified individually
- Migrated back

- Alternatively migrate them into new infrastructure

IDEAS productivity ECP EXASCALE COMPUTING PROJECT

# Example FLASH

- Grid
  - Manages data
  - Domain discretization
- Hydro
  - `simpleUnsplit`
  - `Unsplit`
- Driver
  - Time-stepping
  - Orchestrates interactions



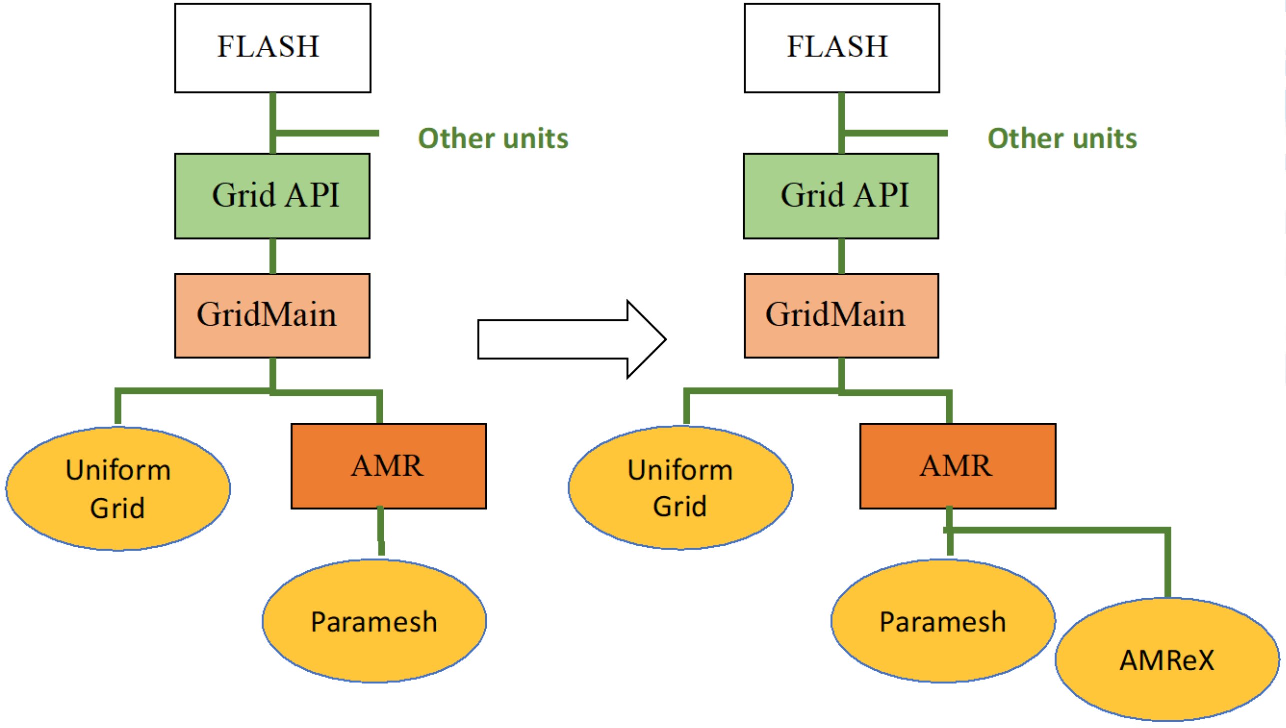
# FLASH5

## Refactoring for Next Generation Hardware

**AMReX -** Lawrence Berkeley National Lab
- Designed for exascale
- Node-level heterogeneity
- Smart iterators hide parallelization

**Goal**: Replace Paramesh with AMReX

**Plan**:
- Paramesh & AMReX coexist
- Adapt interfaces to suit AMReX
- Refactor Paramesh implementation
- Compare AMReX implementation against Paramesh implementation



# Refactoring plan

## Design

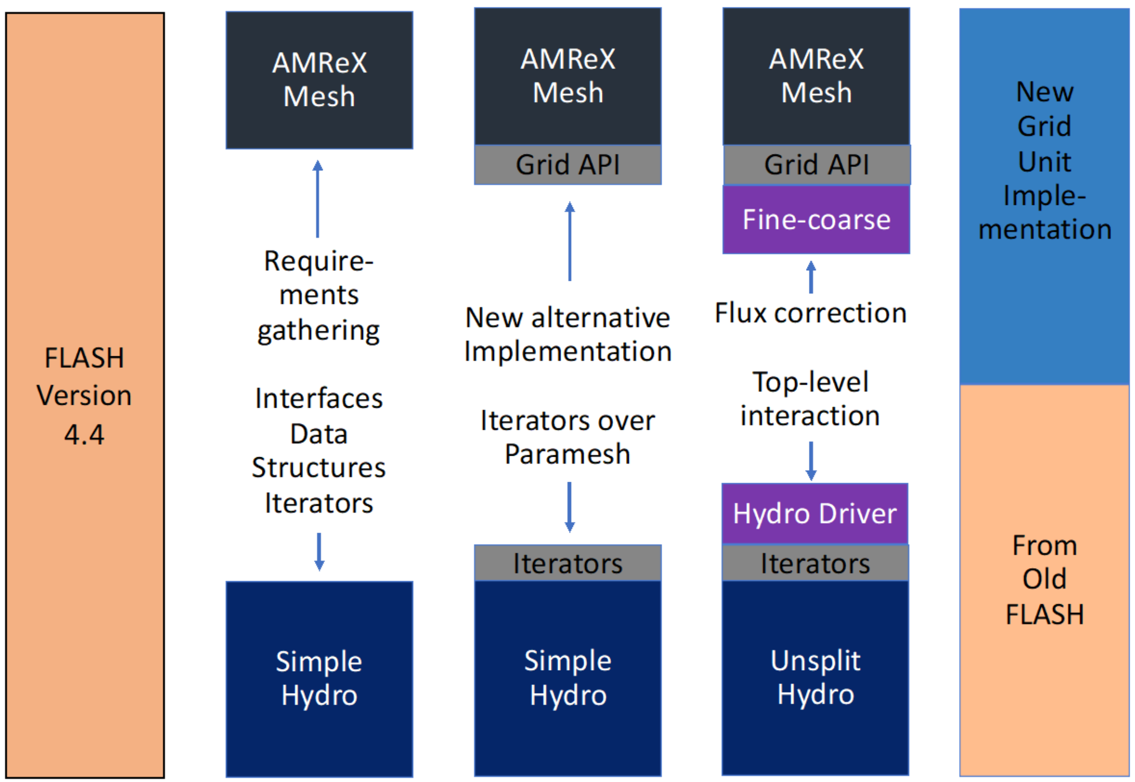
- Degree & scope of change
- Formulate initial requirements

## Prototyping

- Explore & test design decisions
- Update requirements

## Implementation

- Recover from prototyping
- Expand & implement design decisions



# Phase 1 - design

## Sit, think, hypothesize, & argue

- Derive and understand principal definitions & abstractions
- Collect & understand Paramesh/AMReX constraints
  - Generally useful design due to two sets of constraints?
- Collect & understand physics unit requirements on Grid unit
- Design fundamental data structures & update interface
  - AMReX introduces iterators over blocks/tiles of mesh
  - Package up block/tile index with associated mesh metadata
- Minimal prototyping with no verification

# Phase 2 - prototyping

## Quick, dirty, & light

- Implement new data structures
  - Evolve design/implementation by iterating between Paramesh & AMReX
- Explore Grid/physics unit interface
  - `simpleUnsplit` Hydro unit
- Discover use patterns of data structures and Grid unit interface
- Adjust requirements & interfaces

Verification

- Single `simpleUnsplit` simulation
- Quantitative regression test with Paramesh
- Proof of concept with AMReX *via* qualitative comparison with Paramesh

# Phase 3 - implementation

**Toward quantifiable success & Continuous Integration**

- Derive & implement lessons learned
  - Clean code & inline documentation
- Update `Unsplit` Hydro
- Hybrid FLASH
  - AMReX manages data
  - Paramesh drives AMR
- Fully-functioning simulation with AMReX
- Prune old code

Verification

- Git workflow
- Grow test suite / CI with Jenkins
- Add new feature/test
  - Create Paramesh baseline with FLASH4.4
  - Refactor Paramesh implementation
  - Implement with AMReX & compare against Paramesh baseline

# Other resources

**Software testing levels and definitions:**
http://www.tutorialspoint.com/software_testing/software_testing_levels.htm

**Working Effectively with Legacy Code**, Michael Feathers. The legacy software change algorithm described in this book is very straight-forward and powerful for anyone working on a code that has insufficient testing.

**Code Complete**, Steve McConnell. Includes testing advice.

**Organization dedicated to software testing:** https://www.associationforsoftwaretesting.org/

**Software Carpentry:** http://katyhuff.github.io/python-testing/

**Tutorial from Udacity:** https://www.udacity.com/course/software-testing--cs258

**Papers on testing:**
http://www.sciencedirect.com/science/article/pii/S0950584914001232
https://www.researchgate.net/publication/264697060_Ongoing_verification_of_a_multiphysics_community_code_FLASH

**Resources for Trilinos testing:**
Trilinos testing policy: https://github.com/trilinos/Trilinos/wiki/Trilinos-Testing-Policy
Trilinos test harness: https://github.com/trilinos/Trilinos/wiki/Policies--%7C-Testing

# Agenda

| Time | Module | Topic | Speaker |
|---|---|---|---|
| 2:00pm-2:40pm | 01 | Overview of Best Practices in HPC Software Development | Anshu Dubey, ANL |
| 2:40pm-3:20pm | 02 | Better (Small) Scientific Software Teams | David E. Bernholdt, ORNL |
| 3:20pm-4:00pm | 03 | Improving Reproducibility through Better Software Practices | David E. Bernholdt, ORNL |
| *4:00pm-4:30pm* | | *Break* | |
| 4:30pm-5:15pm | 04 | Verification & Refactoring | Anshu Dubey, ANL |
| 5:15pm-6:00pm | 05 | Git Workflow & Continuous Integration | Jared O'Neal, ANL |