Software Refactoring and Documentation

ATPESC 2019

Anshu Dubey
Computer Scientist
Mathematics and Computer Science Division

Q Center, St. Charles, IL (USA)
July 28 – August 9, 2019

License, citation, and acknowledgments

License and Citation

- Requested citation: Anshu Dubey, Software Refactoring and Documentation, in Better Scientific Software Tutorial, Argonne Training Program on Extreme-Scale Computing (ATPESC), St. Charles, IL, 2019. DOI: 10.6084/m9.figshare.9272813.

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 managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357

REFACTORING

About this presentation

- What this lecture is ---
  - Methodology for planning the refactoring process
    - Considerations before and during refactoring
    - Developing a workable process and schedule
    - Possible pitfalls and workarounds
  - Examples from codes that underwent refactoring
    - And lessons learned

- What this lecture is not ---
  - Instructions on detailed process of refactoring
    - It is a difficult process
    - Each project has its own quirks and challenges
    - No one methodology will apply everywhere
  - Tutorial on tools for refactoring
    - There really aren’t that many

Definition

The general definition of refactoring

Refactoring usually applies to object oriented software where the internals of the implementations are "cleaned up" without changing the behavior.

In the context of this lecture

A broad interpretation where any part of the software may change while retaining or enhancing its basic capabilities.

The reason

In context of HPC scientific software the degree of change is motivated by many factors. It may include redesign at a higher level.

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

Reasons for refactoring

The big one these days is the change in platforms

- Once before
  - Vector to risc processors (cpu)
  - Flat memory model to hierarchical memory model

- To heterogeneous
  - Few CPU’s sufficient memory per cpu
  - Several co-existing memory models

- The driving reason for these transitions is performance
  - Performance may drive refactoring even without change in platforms

Reasons for refactoring

There can be other reasons

- Transition of code from research prototype to production
- Imposing architecture and maintainability on an old code
  - Significant change in the code base
    - Change in model or discretization
    - Changes in numerical algorithms
  - Significant change in intended use for the code
    - From a small team to a large team
    - Releasing to wider user base
- Enabling extensibility or configurability
  - Partial common functionality among different usage modes
  - Model refinement
  - Incorporating new insights

Scope of refactoring

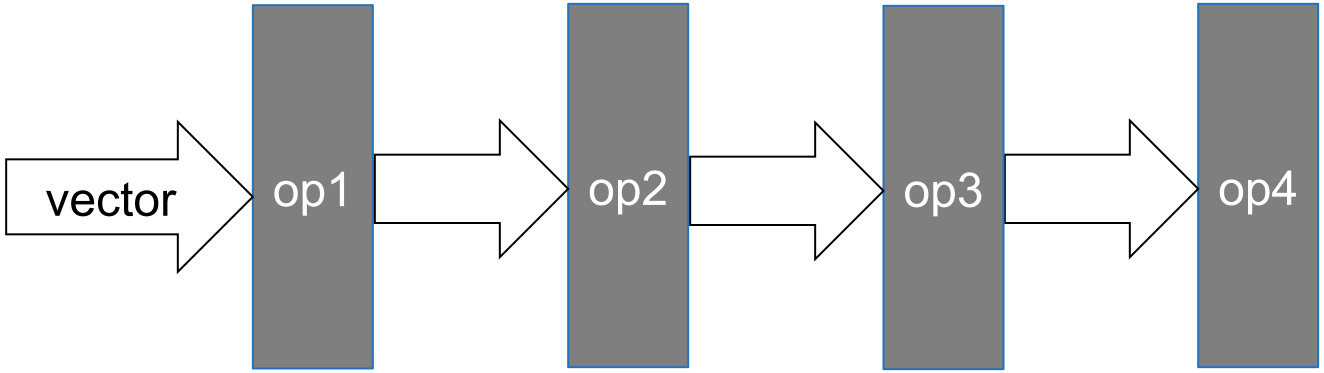
Know where you want the end product to be

- For performance
  - Know the target improvement
    - Very easy to go down the rabbit hole of squeezing the last little bit
    - Almost never worth the effort for obtaining scientific results
- For extensibility
  - Similar to maintainability
  - Greater emphasis on interfaces and encapsulation
- For maintainability
  - Know the boundaries for imposing structure
    - Rewriting the entire code is generally avoidable
    - Kernels for implementing formulae can be left alone ?
    - In general it is possible to stop at higher levels than that

Reasons for refactoring

The big one these days is change in platforms

Transition from vector to risc machines



For vector processors

- Data structures needed to be long vectors
  - Longer => better
- Spatial or temporal locality had no importance
  - Memory access was flat
    - Interleaving banks for better performance

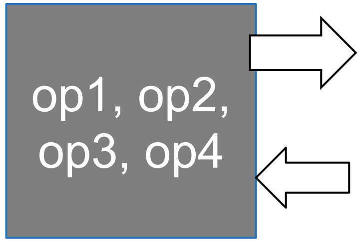
Argonne NATIONAL LABORATORY

EXASCALE COMPUTING PROJECT

Reasons for refactoring

The big one these days is change in platforms

Transition from vector to risc machines



small chunk that could fit
In the cache

For risc processors

- Memory has hierarchy
  - Closer and smaller => faster access
  - Small working sets that can persist in the closest memory preferable
  - Makes spatial and temporal locality important
- Data structures that enable formation of small working sets on which multiple operations can be performed are better

Cost estimation

The biggest potential pitfall

- Can be costly itself if the project is large
- Most projects do a terrible job of estimation
  - Insufficient understanding of code complexity
  - Insufficient provisioning for verification and obstacles
  - Refactoring often overruns in both time and budget

- Factors that can help
  - Knowing the scope and sticking to it
    - If there is change in scope estimate again
  - Plan for all stages of the process with contingency factors built-in
  - Make provision for developing tests and other forms of verification
    - Can be nearly as much or more work than the code change
    - Insufficient verification incurs technical debt

Cost estimation

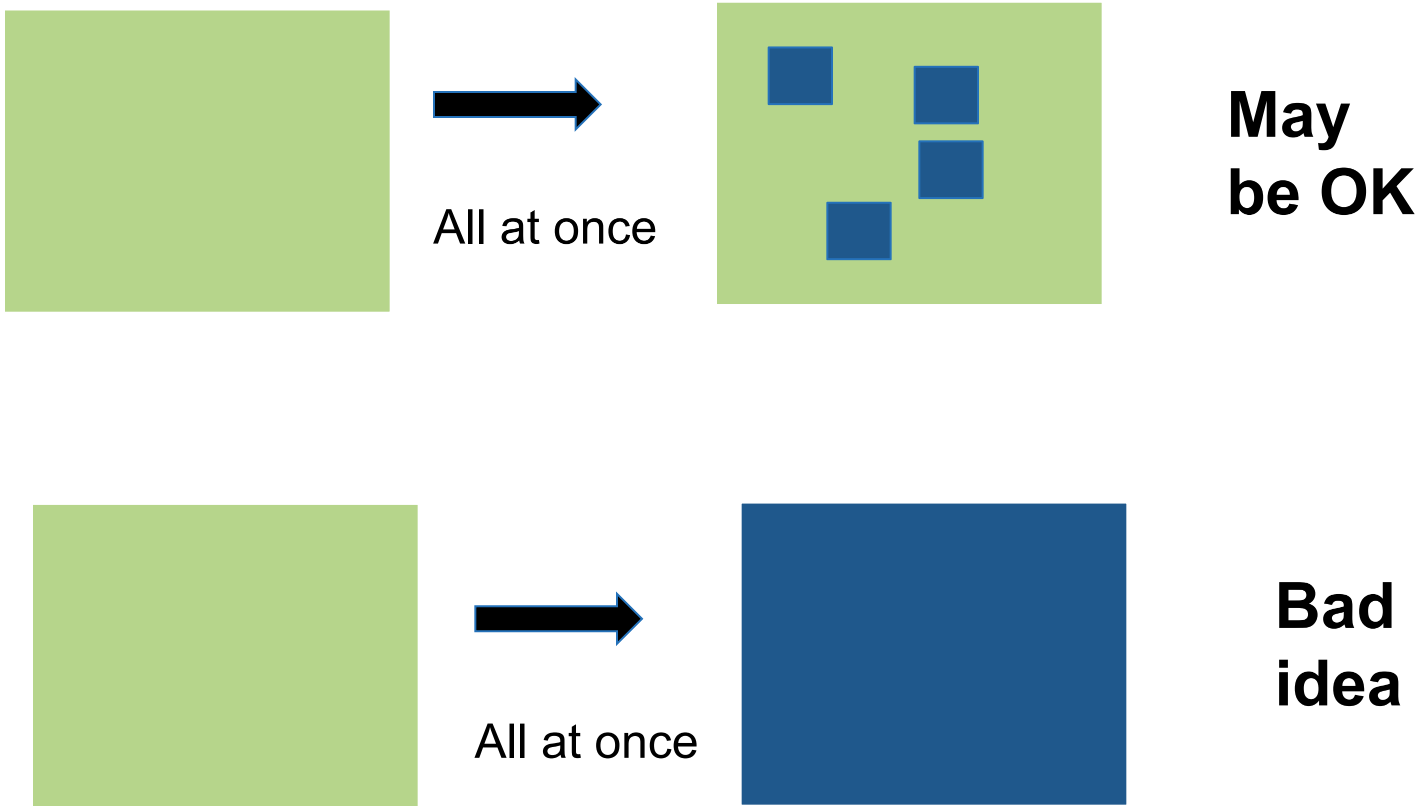
When development and production co-exist

- 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

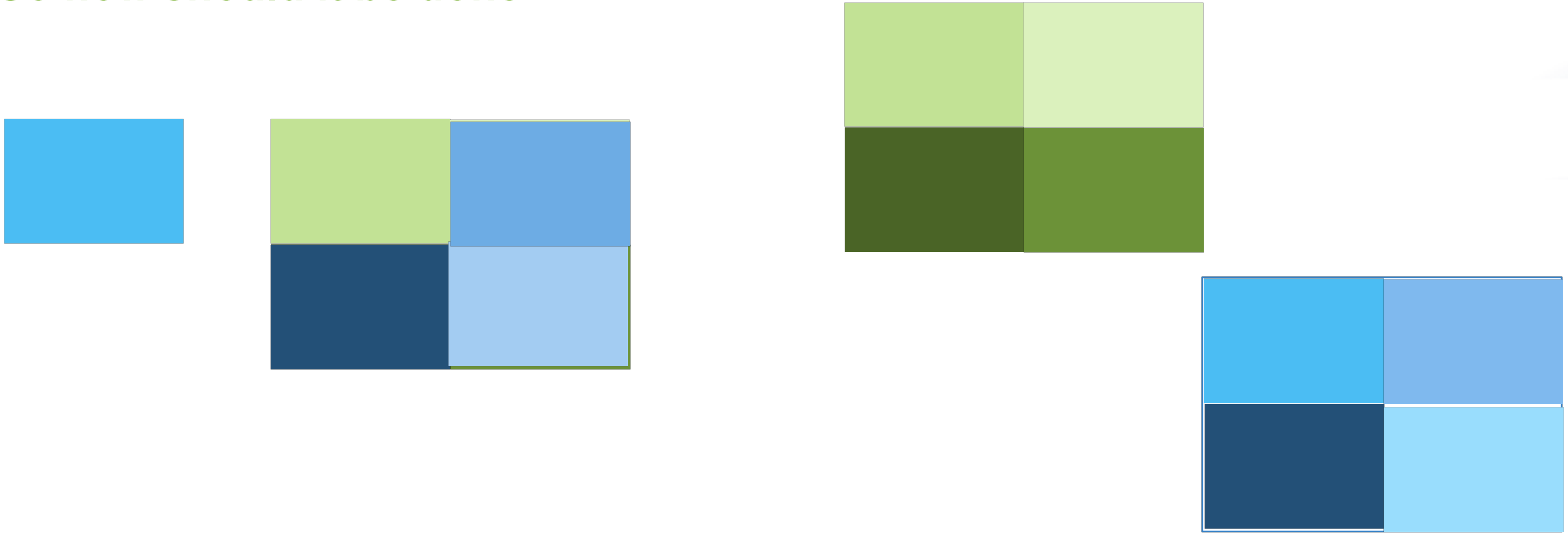
On ramp plan

Proportionate to the scope



On ramp plan

So how should it be done



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

- Alternatively migrate them into new infrastructure

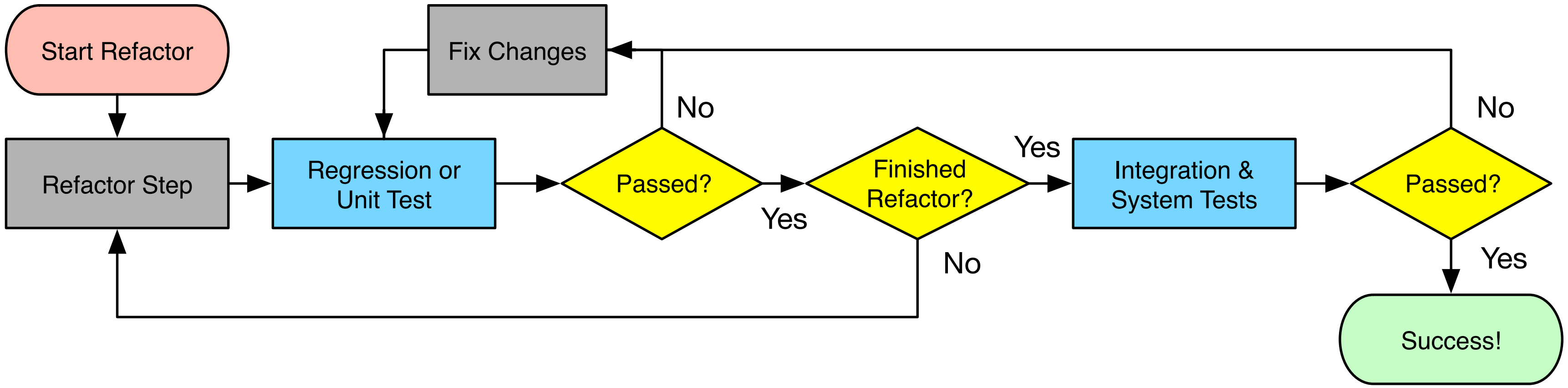
verification

Critical component of refactoring

- Understand the verification needs during transition
- Map from here to there
- Know your error bounds
  - Bitwise reproduction of results unlikely after transition
- Check for coverage provided by existing tests
- Develop new tests where there are gaps
- Make sure tests exist at different granularities
  - There should definitely be demanding integration and system level tests

Refactoring

Workflow with testing



Implementation

Procedures and policies

- Developers (hopefully) know what the end code should be
  - They will do the code implementation

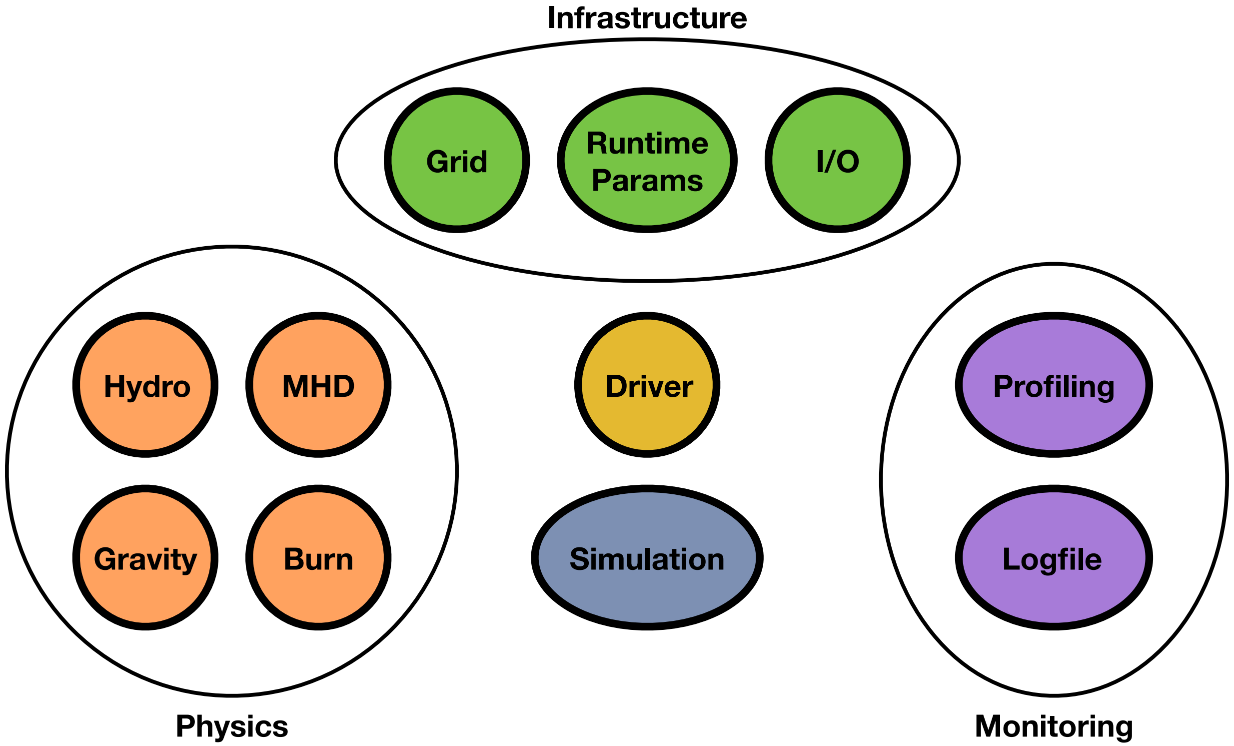
Process and policies are important

- Managing co-existence of production and development
- Managing branch divergence
- Any code pruning
- Schedule of testing
- Schedule of integration and release
  - Release may be external or just to the internal users

EXPERIENCE – FLASH VERSIONS 1-5

Example FLASH

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



Version 1

- Three independently developed codes smashed together
  - Desire to use the same code for many different applications necessitated some thought to infrastructure and architecture
- Challenges
  - F77 style of programming; Common blocks for data sharing
  - Inconsistent data structures, divergent coding practices and no coding standards
- Solution
  - A setup script and config files
  - Concept of alternative implementations, with a script for some plug and play
  - Inheriting directory structure to emulate object oriented approach
  - Wrapper layer with interfaces

Version 2

- Data inventory and interface formalization
  - Modularize the code and make it extensible
  - Elimination of common blocks
  - Formalization of interfaces
- Objectives partially met
  - Centralized database was built
    - It met the data objectives
    - But got in the way of modularization
    - No data scoping, partial encapsulation
    - Database query overheads
- Scope not fully determined
  - Enforced backward compatibility
    - Precluded needed deep changes
    - Hugely increased developer effort
    - High barrier to entry for a new developer
- Not enough buy-in from users
  - Did not get adopted for production in the center for more than two years
    - Development continued in FLASH1.6, and so had to be brought simultaneously into FLASH2 too

Version 3 : the Current Architecture

- Kept inheriting directory structure, configuration and customization mechanisms from earlier versions
- Defined naming conventions
  - Differentiate between namespace and organizational directories
  - Differentiate between API and non-API functions in a unit
  - Prefixes indicating the source and scope of data items
- Formalized the unit architecture
  - Defined API for each unit with null implementation at the top level
- Resolved data ownership and scope
- Resolved lateral dependencies for encapsulation
- Introduced subunits and built-in unit test framework

Version transition

- Build the framework in isolation
  - Used the second model in the ramp-on slide
- Ramp on was planned
  - scope of change was determined ahead of time
    - Determine data scoping and arbitration
    - Code mostly not altered at the kernel level
    - Base APIs for various units
  - scientists were on-board with the plan
    - Including the depth of changes

The Ramp-on Plan

- Infrastructure units first implemented with a homegrown Uniform Grid.
- Unit tests for infrastructure built before any physics was brought over
- Test-suite started on multiple platforms
- Migrate mature solvers (few likely changes) and freeze them in version 2
- Migrate the remaining solvers one application dependencies at a time
- Scientists in the loop for verification and in prioritizing physics migration

Version 4

- Capability building exercise
- Did not need any change in the architecture
- Few infrastructure changes
  - Mesh replication was easily introduced for multigroup radiation
  - Laser drive
  - Interface with linear algebra libraries
- No or minimal changes to existing code

No explicit version transition methodology

Version 5

Ongoing

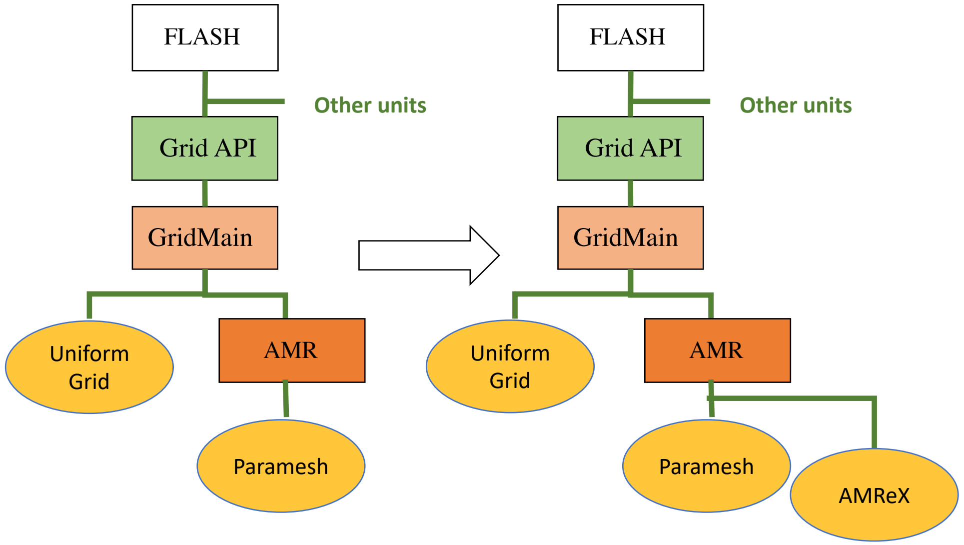
- Objective: prepare for platform and deeper heterogeneity
  - Expected changes in platforms
    - Hierarchical parallelism
    - Remove bulk synchronism
    - Different targets for execution
  - Needed in the code
    - Deeper encapsulation of physics kernels
      - Knowledge of grid
    - Constrained semantics
      - Enable code transformation and optimization

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

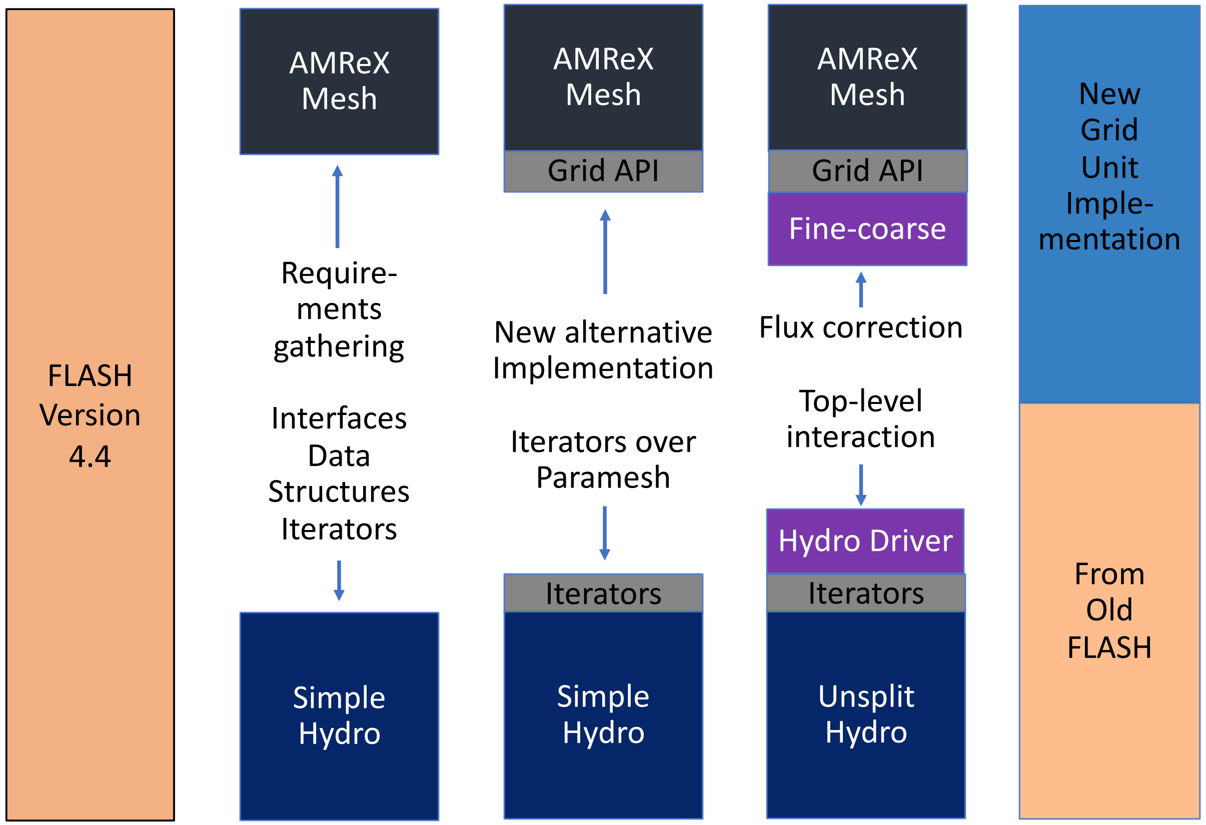
- Degree & scope of change
- Formulate initial requirements

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

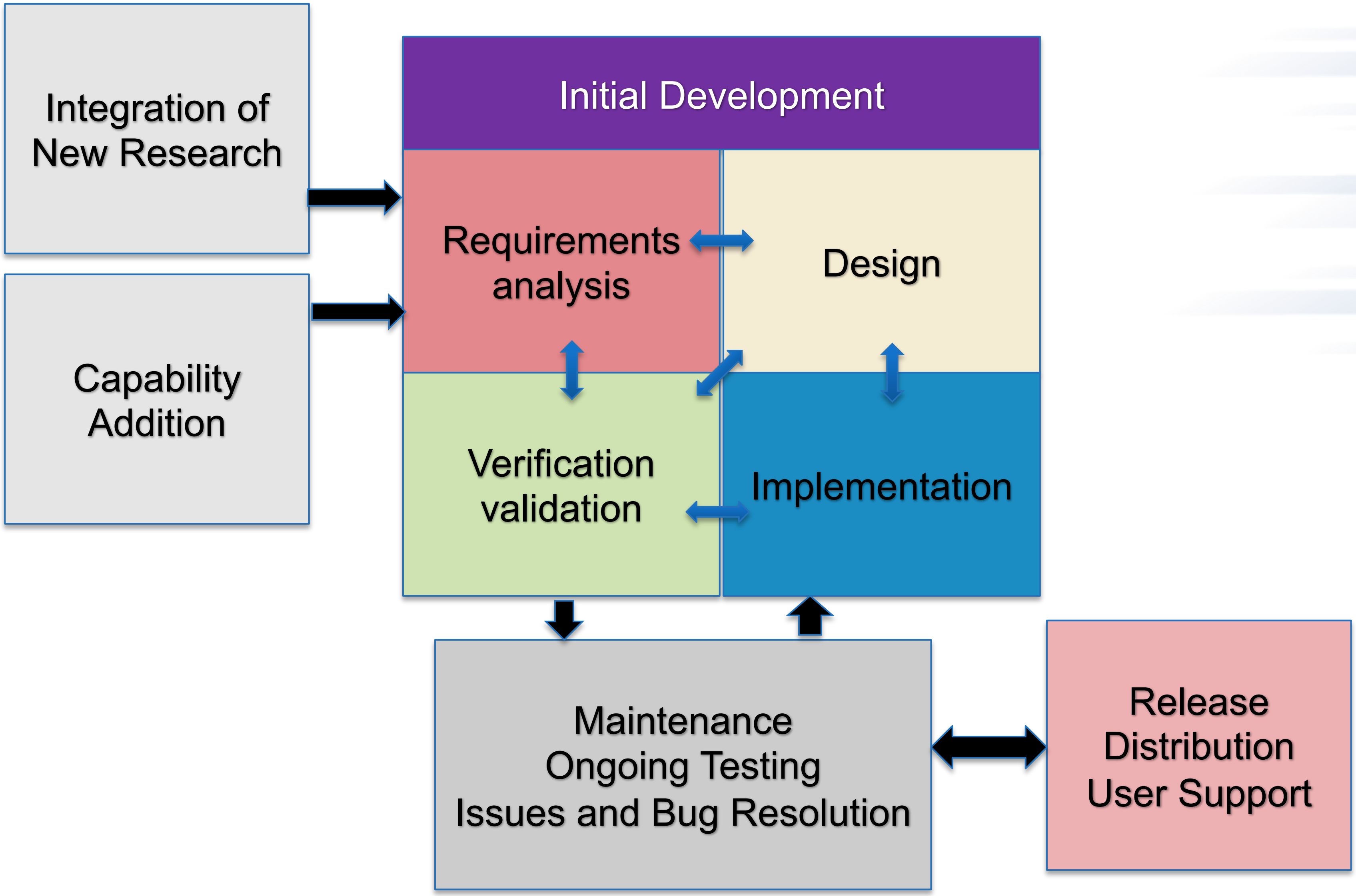
Toward quantifiable success & Continuous Integration

- 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

DOCUMENTATION

Lifecycle



Argonne NATIONAL LABORATORY

ECP EXASCALE COMPUTING PROJECT

Documentation

Initial Development

Requirements analysis

- Expectations from the software
- Capabilities needed
- Solvers needed
- Constraints
- How will they be tested

Example FLASH

- Same code for different applications -> configurability
- Shock Hydro, Degenerate matter EOS, AMR
- Battery of tests

Documentation

Initial Development

Design

- Software overview
- Architecture
- Interfaces
- Coding principles
- Coding standards

Example FLASH

- Design docs → snapshot of discussion
- Online example of unit
- Coding standards

Documentation

Initial Development

- Header – documenting functionality, inputs and outputs and outcomes
- API – tools that autogenerate documentation
  Doxygen, NDoc, Visual Expert, Javadoc, EiffelStudio, Sandcastle, ROBODoc, POD, Twin Text
- Inline documentation
  - Implementation choices

Implementation

Documentation

- User's guide
- Developer's guide
- Reference manual

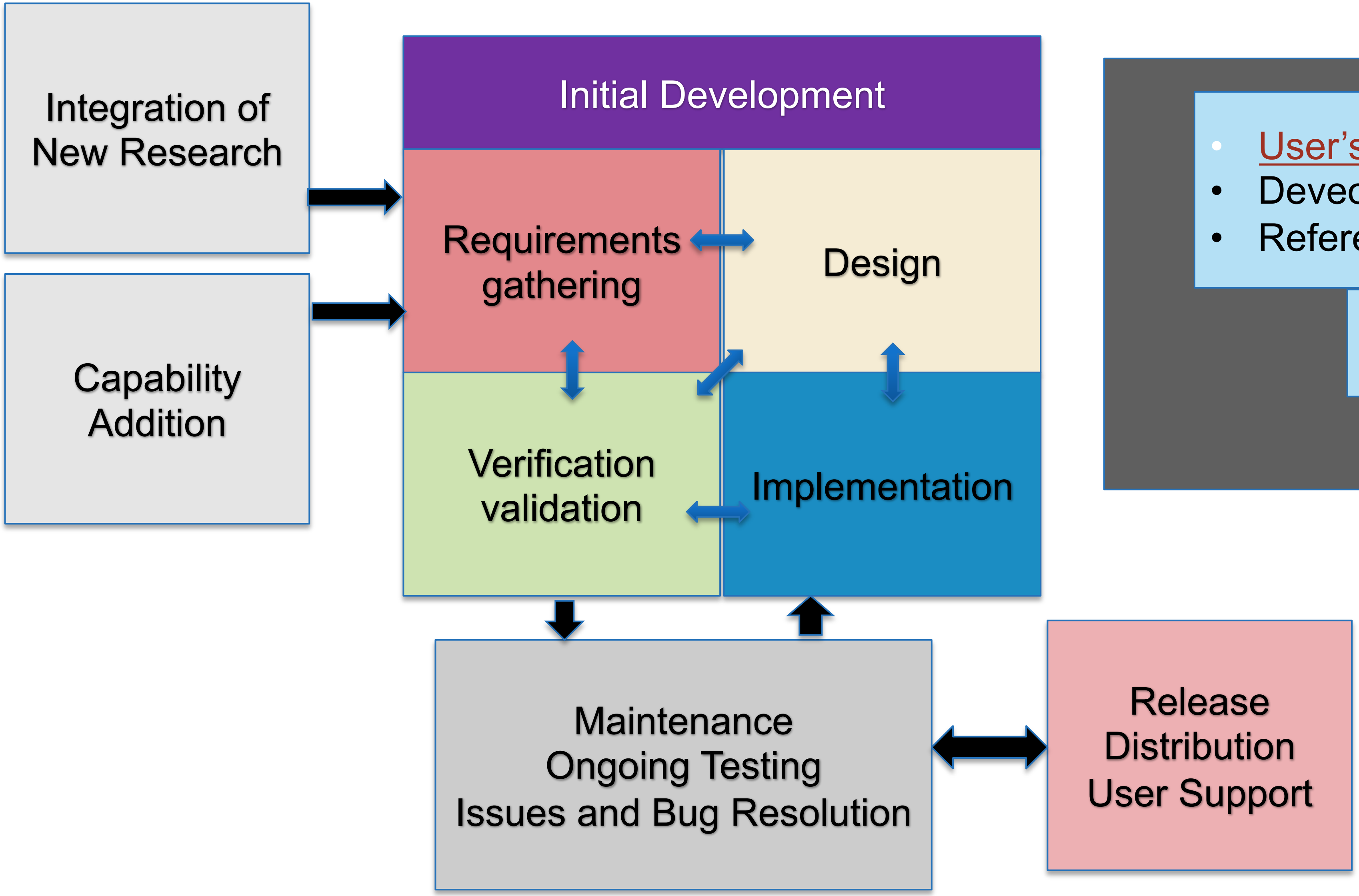
API

Online reference

Maintenance
Ongoing Testing
Issues and Bug Resolution

Release
Distribution
User Support

Documentation



- User’s guide
- Deveoper’s guide
- Reference manual

API

TAKEAWAYS ….

TO HAVE GOOD OUTCOME FROM REFACTORING

KNOW WHY, HOW MUCH, AND COST

PLAN

HAVE STRONG TESTING AND VERIFICATION

GET BUY-IN FROM STAKEHOLDERS

DIFFERENT STAGES OF SOFTWARE NEED DIFFERENT DOCUMENTATION

DOCUMENTING WHY IN THE CODE IS AS IMPORTANT AS HOW