Refactoring Scientific Software

Anshu Dubey
Argonne National Laboratory

Better Scientific Software Tutorial, ISS, March 2021

License, Citation and Acknowledgements

License and Citation

- **The requested citation the overall tutorial is: David E. Bernholdt, Anshu Dubey, Rinku K. Gupta, and David M. Rogers, Better Scientific Software tutorial, in Improving Scientific Software conference, online, 2021. DOI: 10.6084/m9.figshare.14256257**
- Individual modules may be cited as *Speaker, Module Title*, in Better Scientific Software tutorial…

Acknowledgements

- Additional contributors include: Mike Heroux, Alicia Klinvex, Mark Miller, Jared O'Neal, Katherine Riley, David Rogers, Deborah Stevens, James Willenbring
- 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.
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
- This work was performed in part at the Los Alamos National Laboratory, which is managed by Triad National Security, LLC for the U.S. Department of Energy under Contract No.89233218CNA000001
- This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and 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.

What is Refactoring

Definition: Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

- Different from development
  - You have a working code
  - You know and understand the behavior
  - You have a baseline that you can use for comparison

IDEAS productivity ECP EXASCALE COMPUTING PROJECT

What is Refactoring

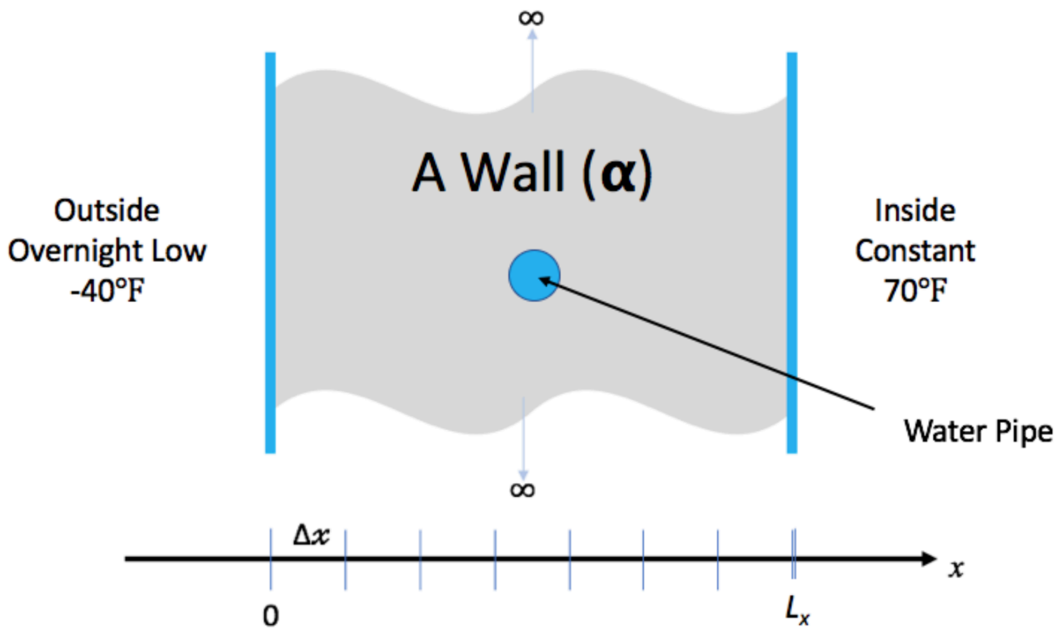
Definition: Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

- Different from development
  - You have a working code
  - You know and understand the behavior
  - You have a baseline that you can use for comparison

- General motivations
  - Modularity enhancement
    - Improve sustainability
  - Release to outside users
    - Easier to use and understand
  - Port to new platforms
    - Performance portability
  - Expand capabilities
    - Structural flexibility

Look at the Running Example

Lets say you live in a house with exterior walls made of a single material of thickness, $$L_x$$. Inside the walls are some water pipes as pictured below.



You keep the inside temperature of the house always at 70 degrees F. But, there is an overnight storm coming. The outside temperature is expected to drop to -40 degrees F for 15.5 hours. Will your pipes freeze before the storm is over?

In the repository there are two versions

- One is a single file with monolithic code
- The other is modularized reusable maintainable code
- If we had only the first version, we would be refactoring to get to the second

Considerations for Refactoring

- Know why you are refactoring
  - Is it necessary
  - Where should the code be after refactoring
- In heat example version 1
  - It is necessary because
    - It is a monolithic code
    - No reusability of any part of the code
    - Devising tests is hard
    - Limited extensibility
  - Where do we want to be after refactoring
    - Closer to the second version
    - More modular, maintainable and extensible

Considerations for Refactoring

- Know the scope of refactoring
  - How deep a change
  - How much code will be affected
- In heat example
  - No capability extension
  - No performance consideration
  - Cleaner, more maintainable code

To convert the monolithic code

- Separate out utilities, generalize interfaces
- Put global definitions in a header file
- Create a general build function
- No new code or intrusive changes

Before Starting

- Know your cost estimates
- Verification
  - Check for coverage provided by existing tests
  - Develop new tests where there are gaps
  - Make sure tests exist at different granularities
    - There should be demanding integration and system level tests

Before Starting

- Know your cost estimates
- Verification
  - Check for coverage provided by existing tests
  - Develop new tests where there are gaps
  - Make sure tests exist at different granularities
    - There should be demanding integration and system level tests
- Know your bounds
  - on acceptable behavior change
  - error bounds
    - bitwise reproduction of results unlikely after transition
- Map from here to there

Incorporate testing overheads into refactoring cost estimates

Exercise: Refactoring the Running Example

- Convert heatAll.C to the cleaner version with reusable code.
  - Though a solution is there in the repo, your solution need not be identical
  - Think about how you want your final product to be and then go through the exercise of refactoring

- Here as an example exercise, I am taking the clean solution and generalizing the update_solution interface
  - Motivation: Do not want to change heat.C for adding another method
  - For this exercise we will use “ftcs” and “upwind15” as alternative options

Preparing for Refactoring – check coverage

- Run ./heat runame="ftcs_results"
- Run gcov heat.C
- Examine heat.C.gcov

- A dash indicates non-executable line
- A number indicated the times the line was called
- ##### indicates line wasn't exercised

```
HDR = Double.H
SRC = heat.C utils.C args.C exact.C ftcs.C upwind15.C crankn.C
OBJ = $(SRC:.C=.o)
GCOV = $(SRC:.C=.C.gcov) $(SRC:.C=.gcda) $(SRC:.C=.gcno) $(HDR:.
H=.H.gcov)
EXE = heat

# Implicit rule for object files
%.o : %.C
	$(CXX) -c -coverage $(CXXFLAGS) $(CPPFLAGS) $< -o $@

# Linking the final heat app
heat: $(OBJ)
	$(CXX) -coverage -o heat $(OBJ) $(LDFLAGS) -lm
```

```
        -:  143:static bool
      500:  144:update_solution()
        -:  145:{
      500:  146:    if (!strcmp(alg, "ftcs"))
      500:  147:        return update_solution_ftcs(Nx, curr, last, alpha, dx, dt, bc0, bc1);
    #####:  148:    else if (!strcmp(alg, "upwind15"))
    #####:  149:        return update_solution_upwind15(Nx, curr, last, alpha, dx, dt, bc0, bc1);
    #####:  150:    else if (!strcmp(alg, "crankn"))
    #####:  151:        return update_solution_crankn(Nx, curr, last, cn_Amat, bc0, bc1);
    #####:  152:    return false;
      500:  153:}
        -:  154:
        -:  155:static Double
      500:  156:update_output_files(int ti)
        -:  157:{
      500:  158:    Double change;
        -:  159:
      500:  160:    if (ti>0 && save)
        -:  161:    {
    #####:  162:        compute_exact_solution(Nx, exact, dx, ic, alpha, ti*dt, bc0, bc1);
    #####:  163:        if (savi && ti%savi==0)
    #####:  164:            write_array(ti, Nx, dx, exact);
    #####:  165:    }
```

Preparing for Refactoring – get baselines

- Call to upwind15 not exercised
- Run ./heat alg="upwind15" runame="upwind_results

```
      -:  143:static bool
    500:  144:update_solution()
      -:  145:{
    500:  146:    if (!strcmp(alg, "ftcs"))
  #####:  147:        return update_solution_ftcs(Nx, curr, last, alpha, dx, dt, bc0, bc1);
    500:  148:    else if (!strcmp(alg, "upwind15"))
    500:  149:        return update_solution_upwind15(Nx, curr, last, alpha, dx, dt, bc0, bc1);
  #####:  150:    else if (!strcmp(alg, "crankn"))
  #####:  151:        return update_solution_crankn(Nx, curr, last, cn_Amat, bc0, bc1);
  #####:  152:    return false;
    500:  153:}
      -:  154:
```

- We have baselines for ftcs and upwind

```
ahilya:clean dubey$ ls ftcs_results/
clargs.out                         ftcs_results_soln_00000.curve   ftcs_results_soln_final.curve
ahilya:clean dubey$ ls upwind_results/
clargs.out                         upwind_results_soln_00000.curve upwind_results_soln_final.curve
ahilya:clean dubey$
```

Refactoring – The starting code

```
extern bool
update_solution_ftcs(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double bc_0, Double bc_1);

extern bool
update_solution_upwind15(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double bc_0, Double bc_1);

extern bool
update_solution_crankn(int n,
    Double *curr, Double const *last,
    Double const *cn_Amat,
    Double bc_0, Double bc_1);
```

```
if (!strncmp(alg, "crankn", 6))
    initialize_crankn(Nx, alpha, dx, dt, &cn_Amat);
```

- Interfaces are not identical
- crankn has an extra argument
- It also has an extra step in initialization

Refactoring

- Generalize the interface

```
extern bool
update_solution(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double const *cn_Amat,
    Double bc_0, Double bc_1);
```

- Modify the makefile

Refactoring

- Generalize the interface

```
extern bool
update_solution(int n,
     Double *curr, Double const *last,
     Double alpha, Double dx, Double dt,
     Double const *cn_Amat,
     Double bc_0, Double bc_1);
```

```
HDR = Double.H
SRC1 = heat.C utils.C args.C exact.C ftcs.C
SRC2 = heat.C utils.C args.C exact.C upwind15.C
SRC3 = heat.C utils.C args.C exact.C crankn.C
OBJ1 = $(SRC1:.C=.o)
OBJ2 = $(SRC2:.C=.o)
OBJ3 = $(SRC3:.C=.o)

EXE1 = heat1
EXE2 = heat2
EXE3 = heat3
```

- Modify the makefile

Refactoring

- Generalize the interface

```
extern bool
update_solution(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double const *cn_Amat,
    Double bc_0, Double bc_1);
```

```
HDR = Double.H
SRC1 = heat.C utils.C args.C exact.C ftcs.C
SRC2 = heat.C utils.C args.C exact.C upwind15.C
SRC3 = heat.C utils.C args.C exact.C crankn.C
OBJ1 = $(SRC1:.C=.o)
OBJ2 = $(SRC2:.C=.o)
OBJ3 = $(SRC3:.C=.o)

EXE1 = heat1
EXE2 = heat2
EXE3 = heat3
```

- Modify the makefile
- Add null implementations of initialize_crank in ftcs and upwind15

Refactoring

```
void
initialize_crankn(int n,
    Double alpha, Double dx, Double dt,
    Double **_cn_Amat)
{
}

bool
update_solution(int n, Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double const *cn_Amat,
    Double bc_0, Double bc_1)
{
    Double const f2 = 1.0/24;
    Double const f1 = 1.0/6;
    Double const f0 = 1.0/4;
    Double const k = alpha * alpha * dt / (dx * dx);
    Double const k2 = k*k;
```

- make heat1
- Run ./heat runame="ftcs_results"
- Make heat2
- Run ./heat runame="upwind_results"
- Verify against baseline

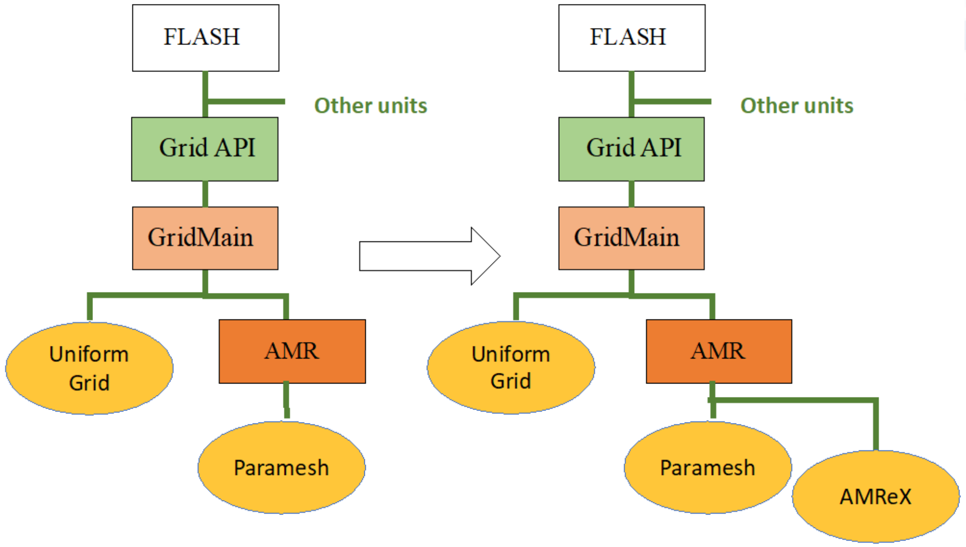
A Real World Example: FLASH

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

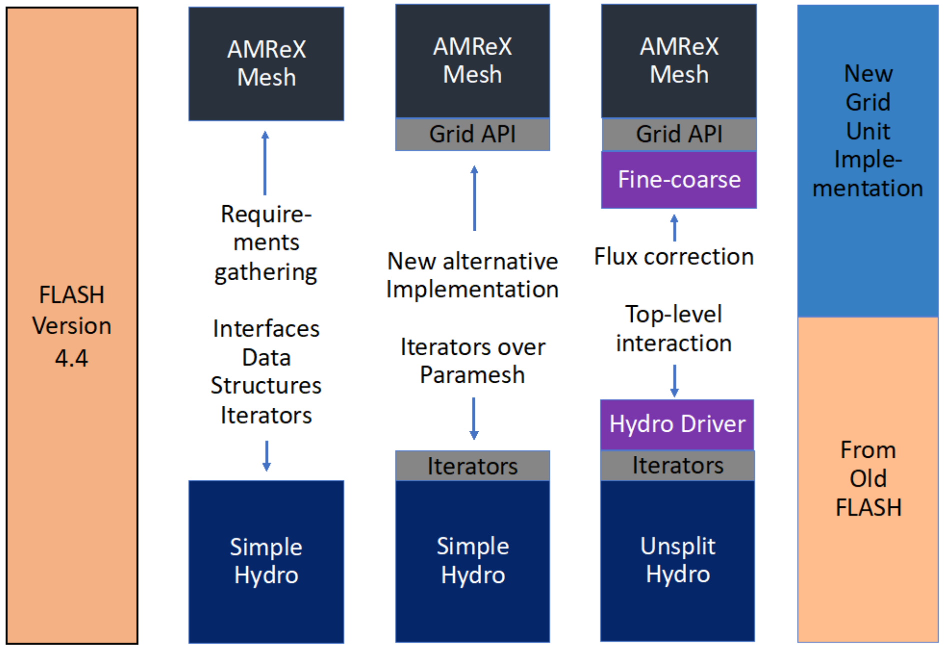
Plan: Getting there from here
- On ramping
- Design
- Intermediate steps
- Realizing the goal



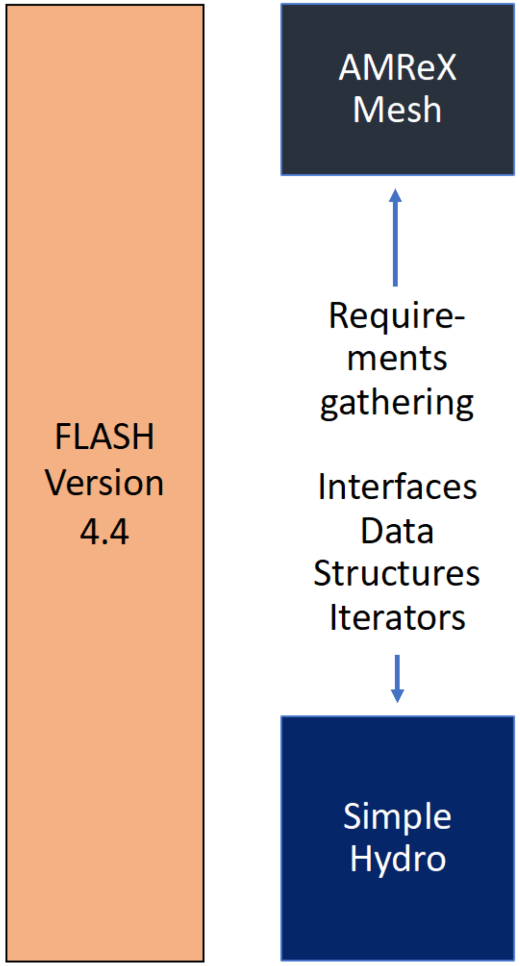
Considerations

- Cost estimation
  - 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

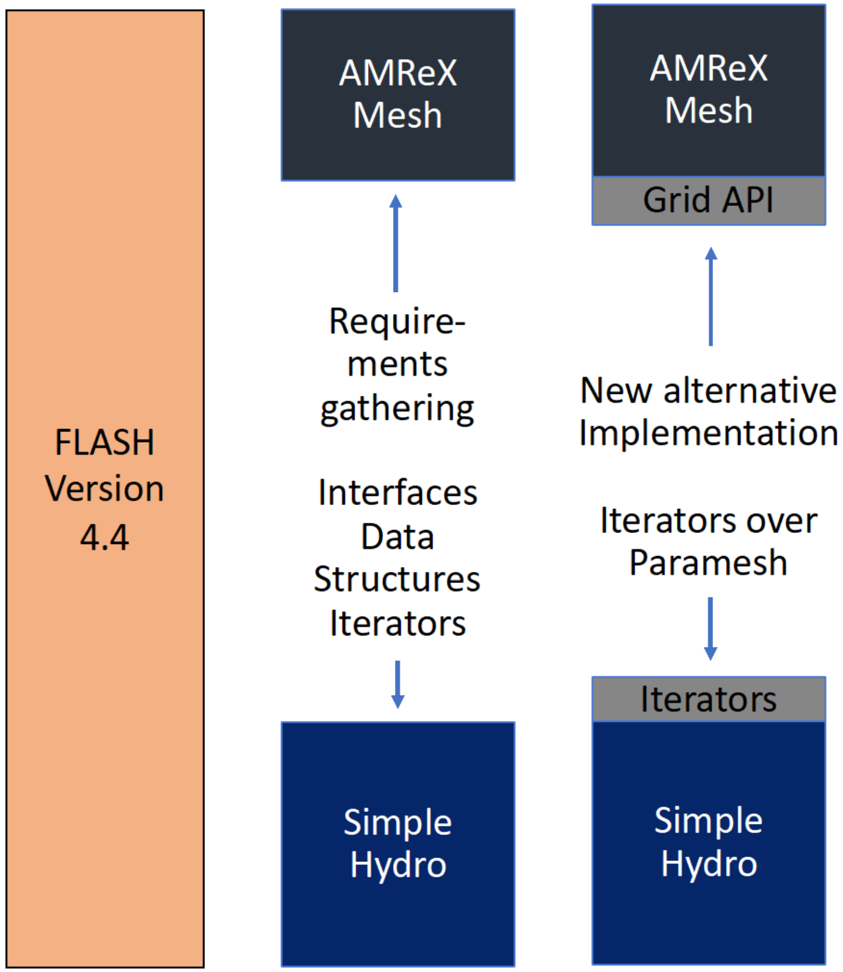
- In FLASH
  - Initial estimate at 6-12 months
  - Took close to 12 months



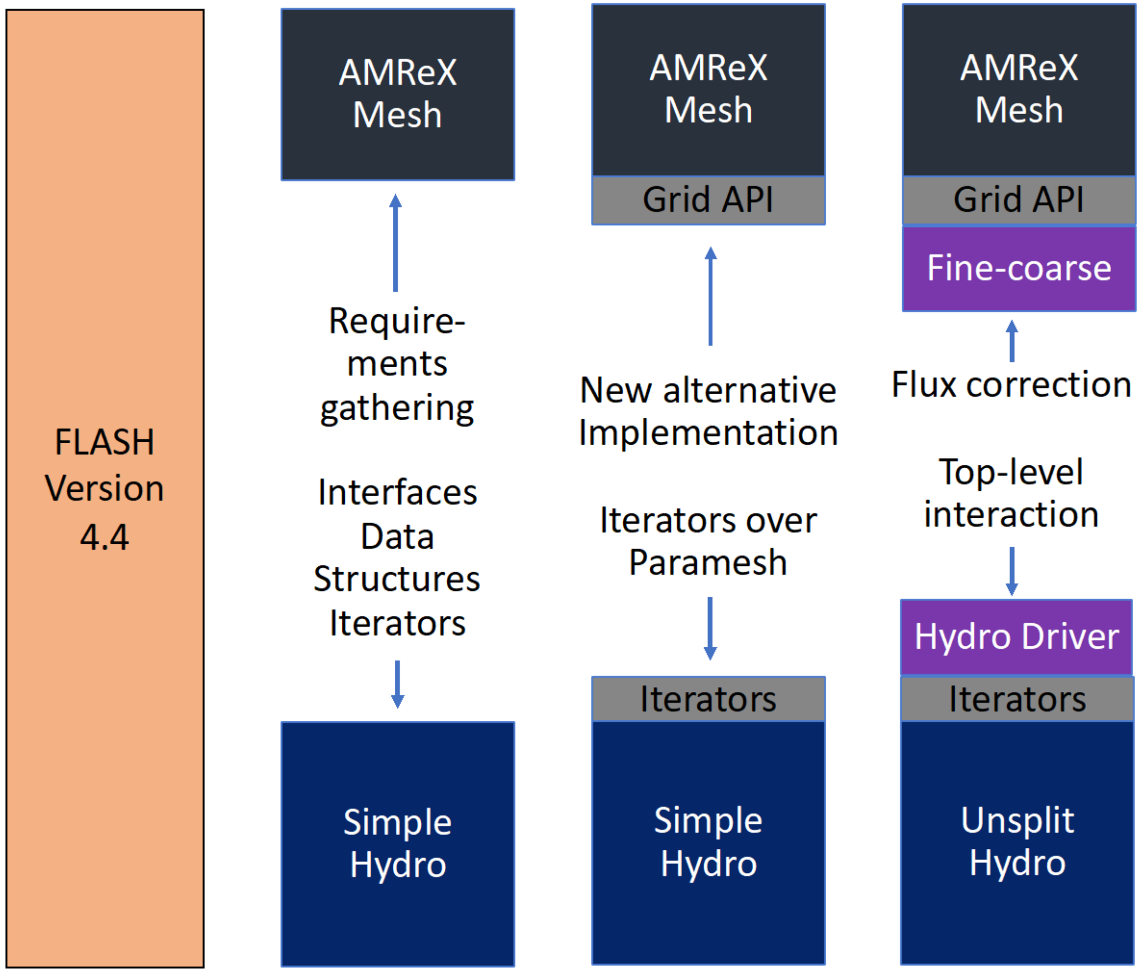
Steps in the Process



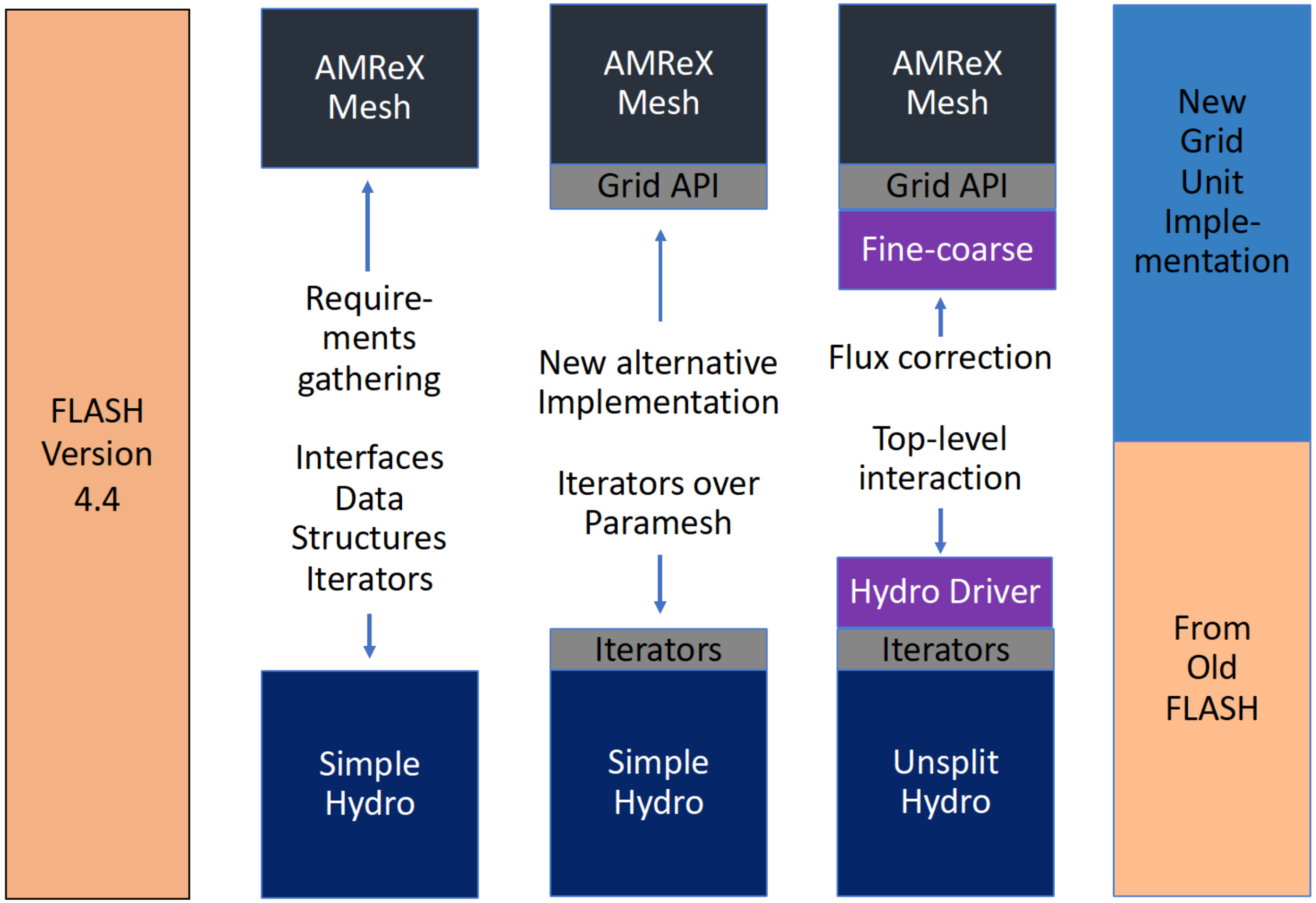
Steps in the Process



Steps in the Process



Steps in the Process



TO HAVE GOOD OUTCOME FROM REFACTORING

1. KNOW WHY
2. KNOW HOW MUCH
3. KNOW THE COST
4. PLAN
5. HAVE STRONG TESTING AND VERIFICATION
6. GET BUY-IN FROM STAKEHOLDERS

Agenda

Time (MDT)	Module	Topic	Speaker
1:00pm-1:05pm	00	Introduction	David E. Bernholdt, ORNL
1:05pm-1:15pm	01	Motivation and Overview of Best Practices in HPC Software Development	David E. Bernholdt, ORNL
1:15pm-1:45pm	02	Agile Methodologies	Rinku K. Gupta, ANL
1:45pm-2:00pm	03	Git Workflows	Rinku K. Gupta, ANL
2:00pm-2:20pm	04	Software Testing 1	David M. Rogers, ORNL
2:20pm-2:40pm		*Break (optional Q&A)*	*All*
2:40pm-3:00pm	05	Software Design	Anshu Dubey, ANL
3:00pm-3:15pm	06	Software Testing 2	David M. Rogers
3:15pm-3:40pm	07	Refactoring	Anshu Dubey, ANL
3:40pm-3:55pm	08	Reproducibility	David E. Bernholdt, ORNL
3:55pm-4:00pm	09	Summary	David E. Bernholdt, ORNL