



# Refactoring Scientific Software



David M. Rogers (he/him)  
Oak Ridge National Laboratory

Better Software for Reproducible Science tutorial @ SC23

Contributors: Anshu Dubey (ANL), Mark C. Miller (LLNL), David M. Rogers (ORNL), David E. Bernholdt (ORNL)




See slide 2 for  
license details



# License, Citation and Acknowledgements

## License and Citation

- This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/) (CC BY 4.0). 
- **The requested citation the overall tutorial is:** David E. Bernholdt, Patricia A. Grubel, David M. Rogers, and Gregory R. Watson, Better Software for Reproducible Science tutorial, in The International Conference for High-Performance Computing, Networking, Storage, and Analysis (SC23), Denver, Colorado, 2023. DOI: [10.6084/m9.figshare.24226105](https://doi.org/10.6084/m9.figshare.24226105).
- Individual modules may be cited as *Speaker, Module Title, in Tutorial Title, ...*

## 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.
- 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 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 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.

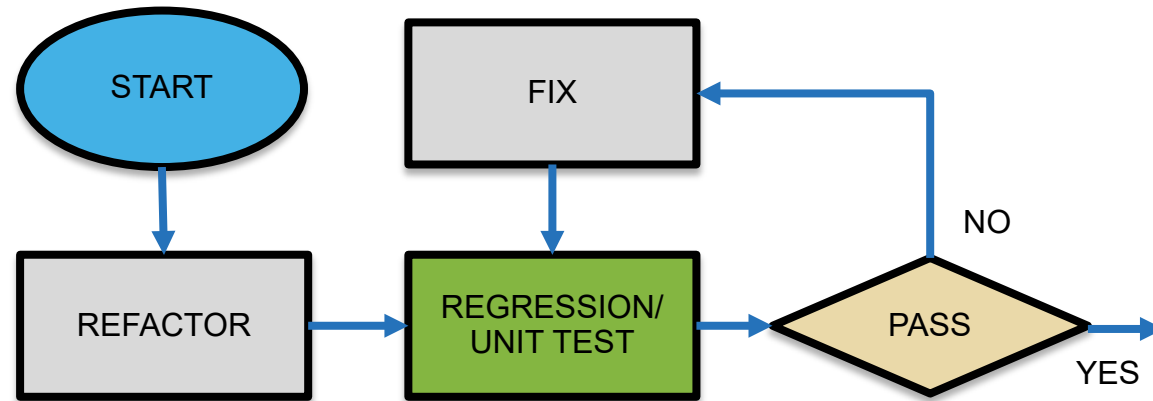
- Different from development
  - You have a working code
  - You know and understand the behavior
  - You have a baseline that you can use for comparison

# What is Refactoring

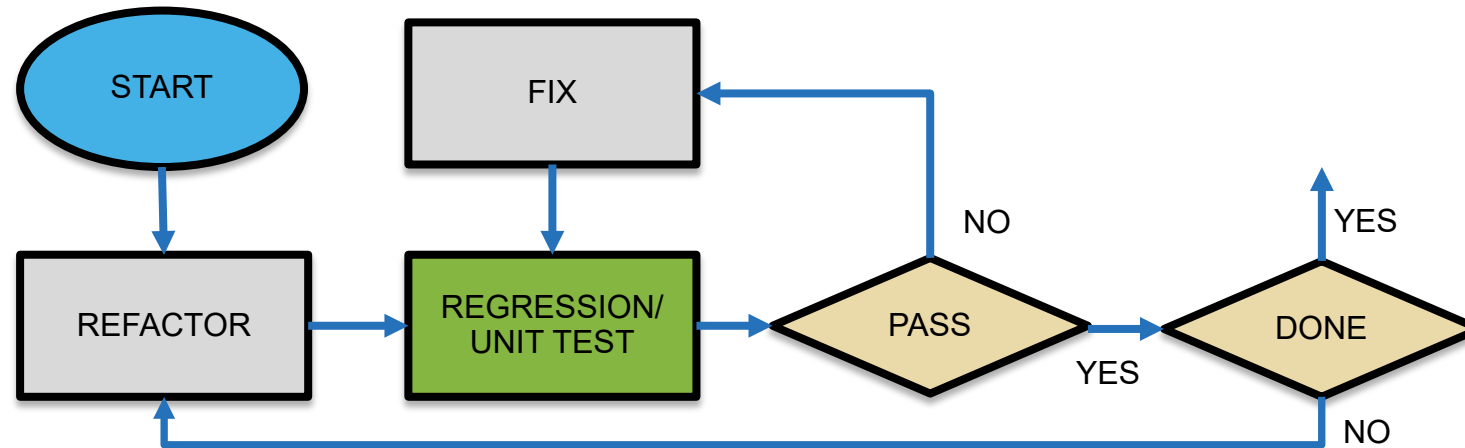
Definition: Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

- 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

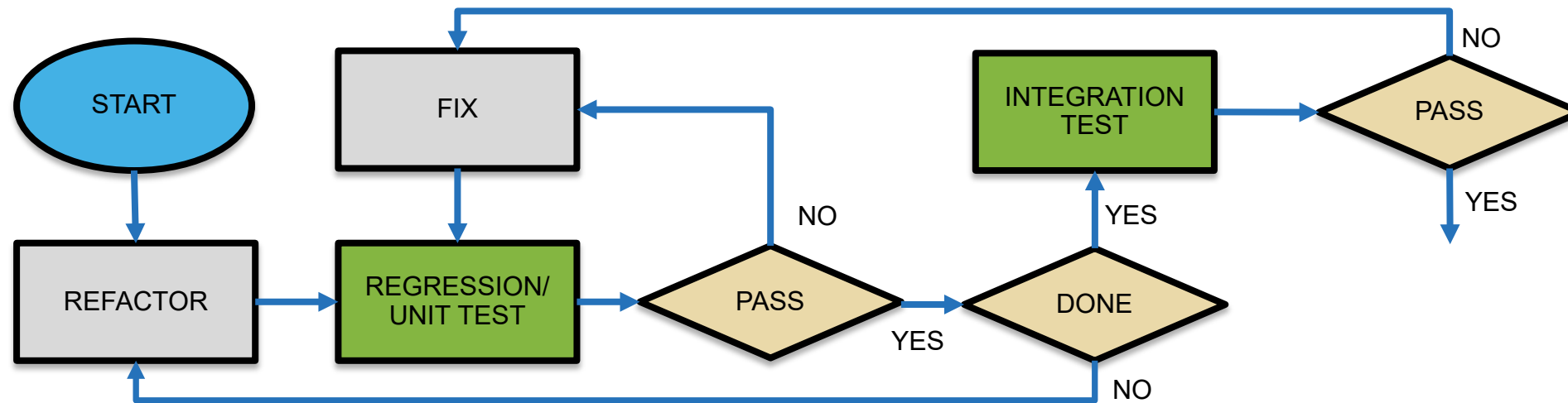
# An Example Workflow



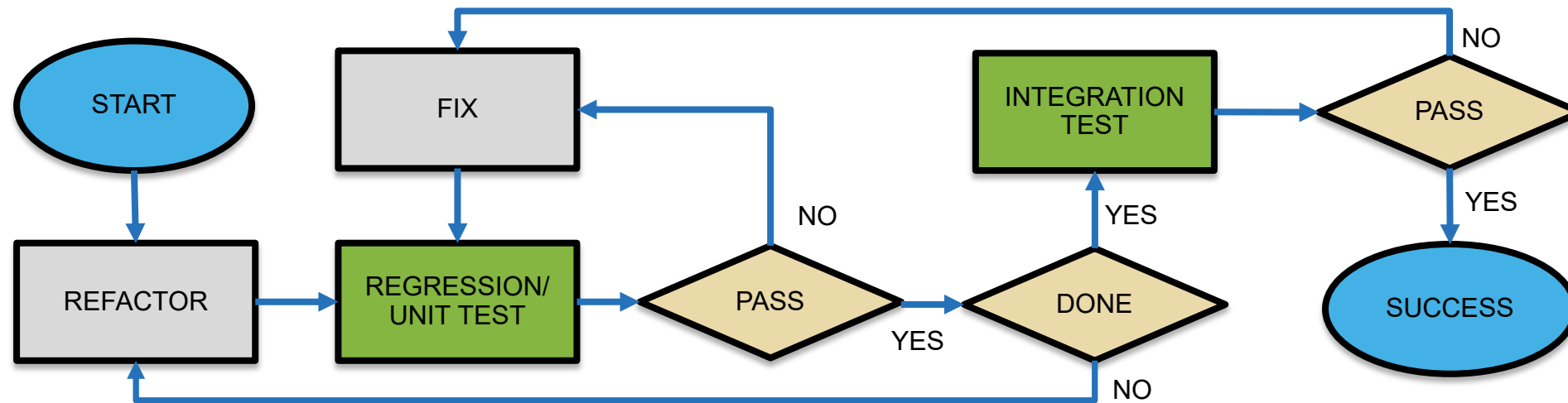
# An Example Workflow



# An Example Workflow



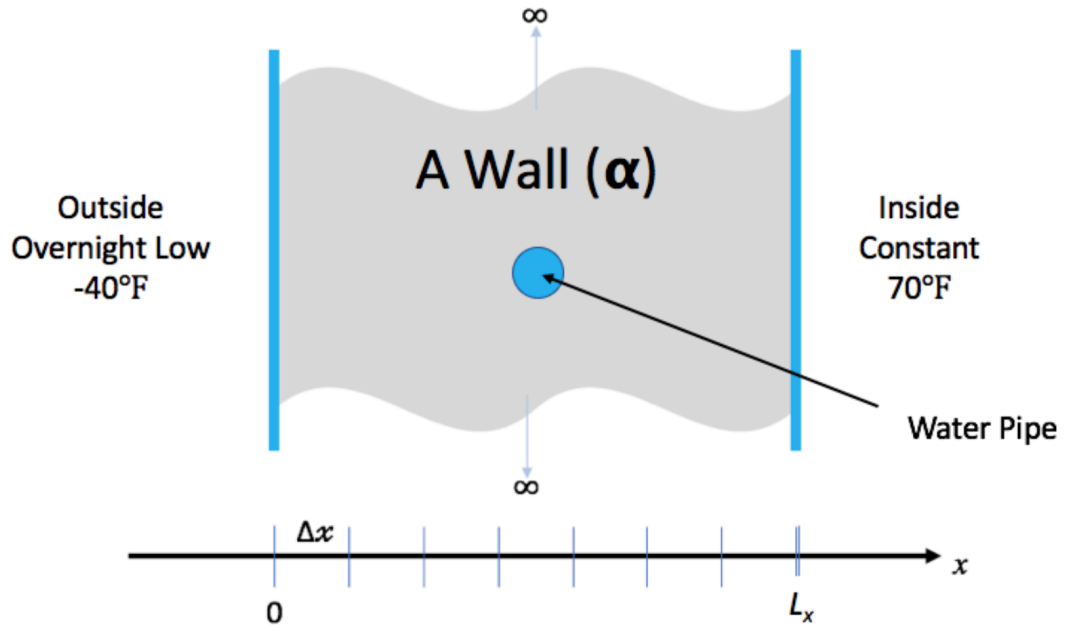
# An Example Workflow





# 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?

Consider two versions of this code...

- 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 modularize 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

# 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 bssw-tutorial/hello-numerical-world

- 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);
```

- Modify the makefile

```
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
```

# 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
- Add null implementations of initialize\_crank in ftcs and upwind15

```
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
```

# Refactoring

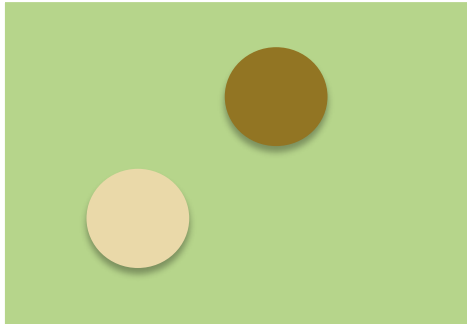
```
void
initialize_crankn(int n,
    Double alpha, Double dx, Double dt,
    Double **_cn_Amat)
{
}

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;
```

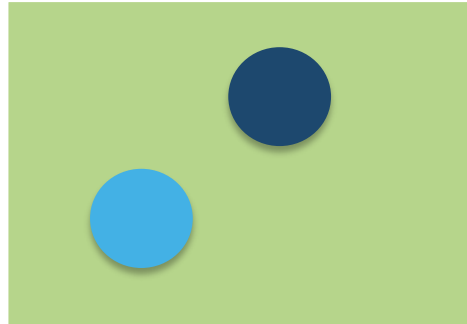
- make heat1
- Run ./heat runame="ftcs\_results"
- Make heat2
- Run ./heat runame="upwind\_results"
- Verify against baselines

# Map from Here to There: On ramp plan

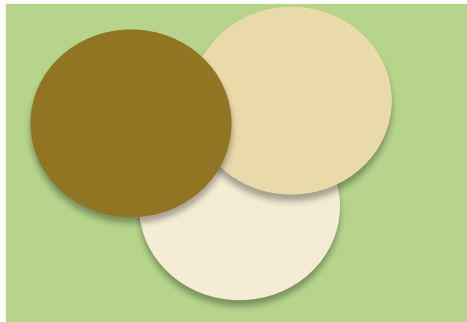
Proportionate to the scope



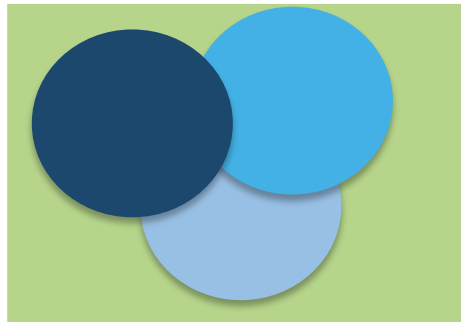
All at once



**Scattered independent changes - May be OK**

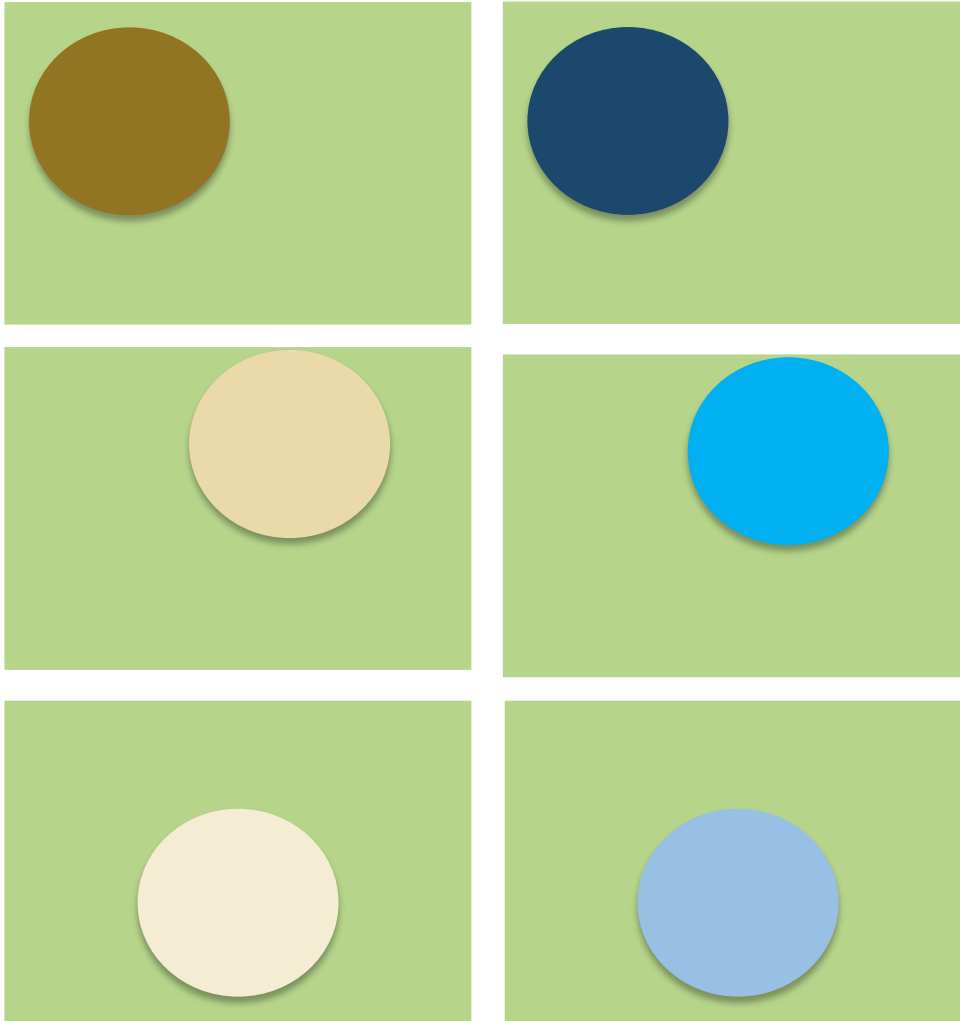


All at once



**Invasive large-scale change in the code - Bad idea**

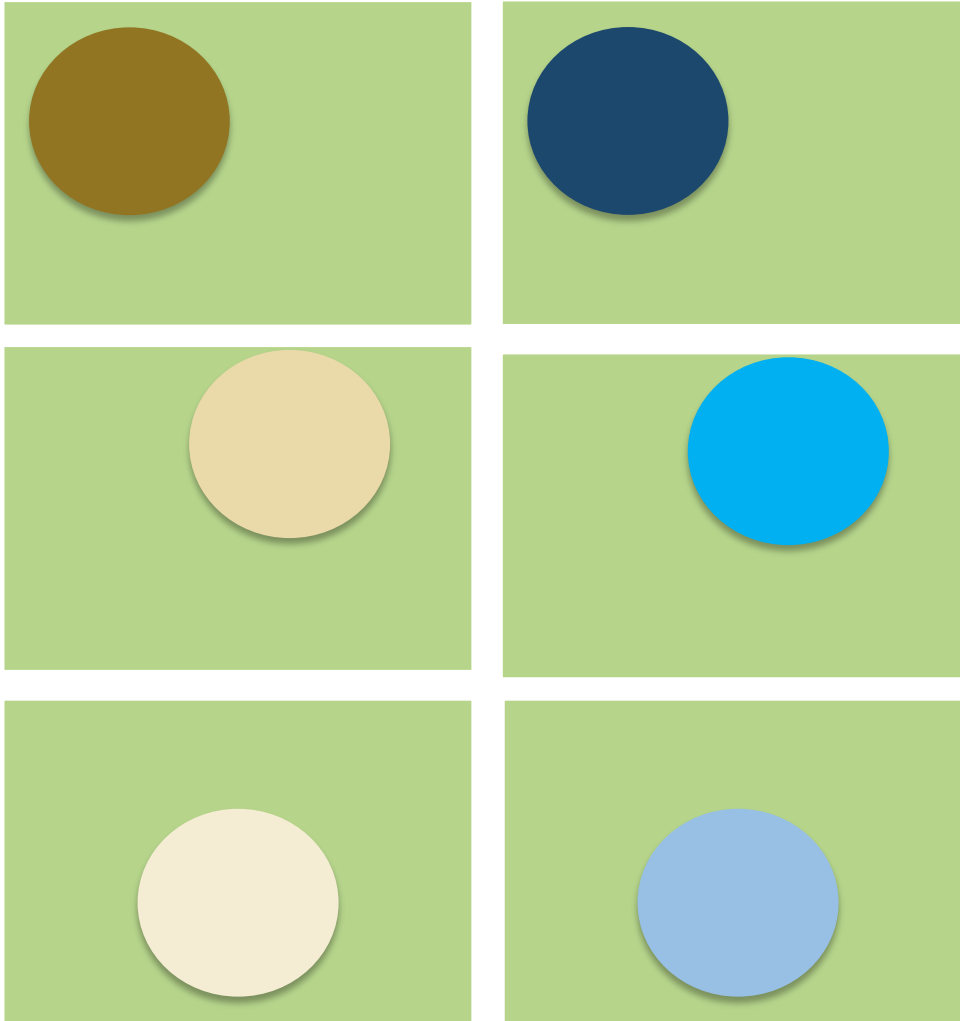
# Map from Here to There: On ramp plan1



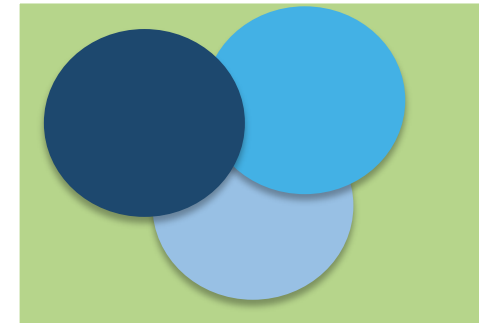
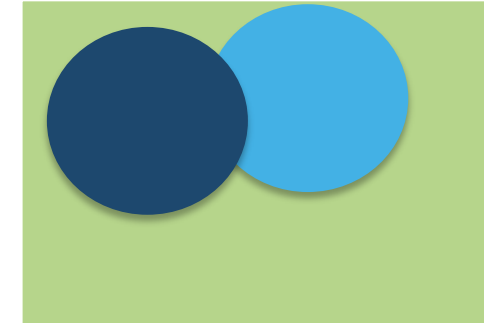
- Turn off all modules except for the one being refactored.
- Have a way of testing in intermediate stages
- Do this for all modules that need refactoring independently



# Map from Here to There: On ramp plan1

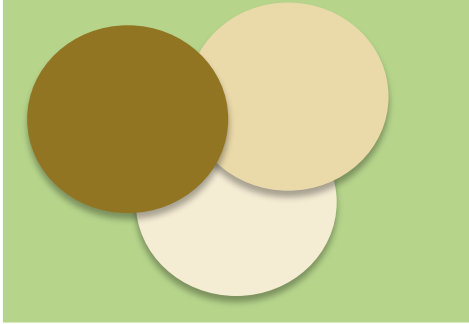


- Turn off all modules except for the one being refactored.
- Have a way of testing in intermediate stages
- Do this for all modules that need refactoring independently



- One by one turn on more than one refactored module

# Map from Here to There: On ramp plan2

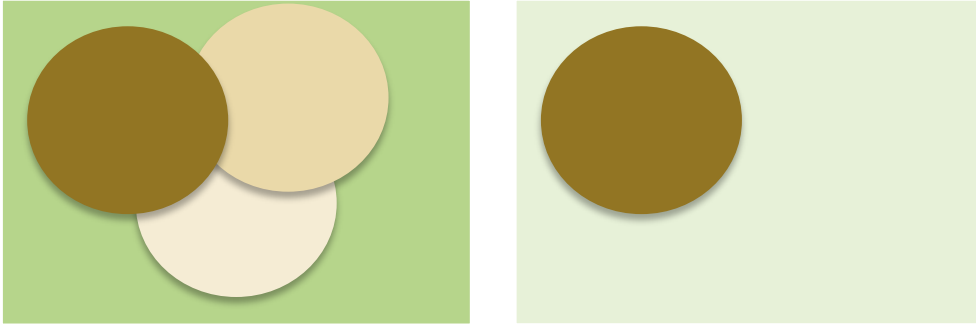


# Map from Here to There: On ramp plan2



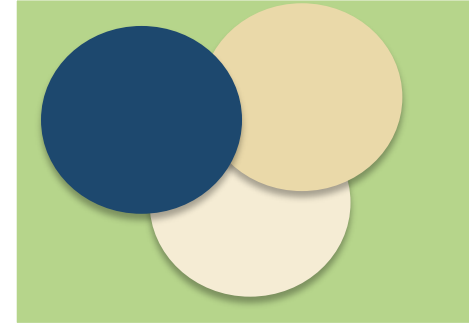
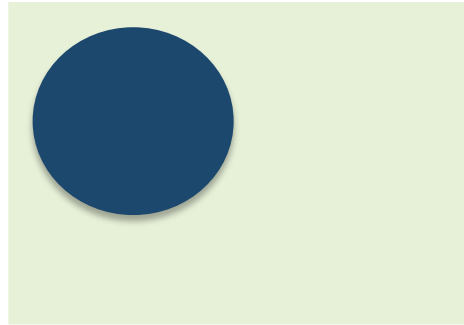
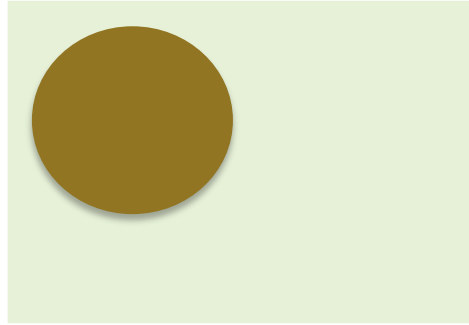
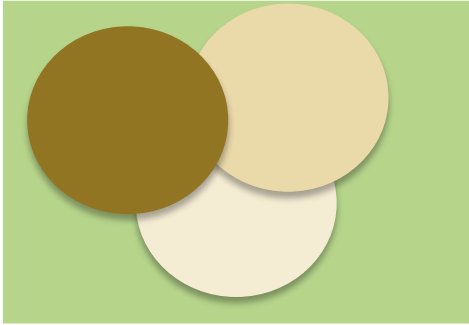
- Build a separate environment for testing refactored module

# Map from Here to There: On ramp plan2



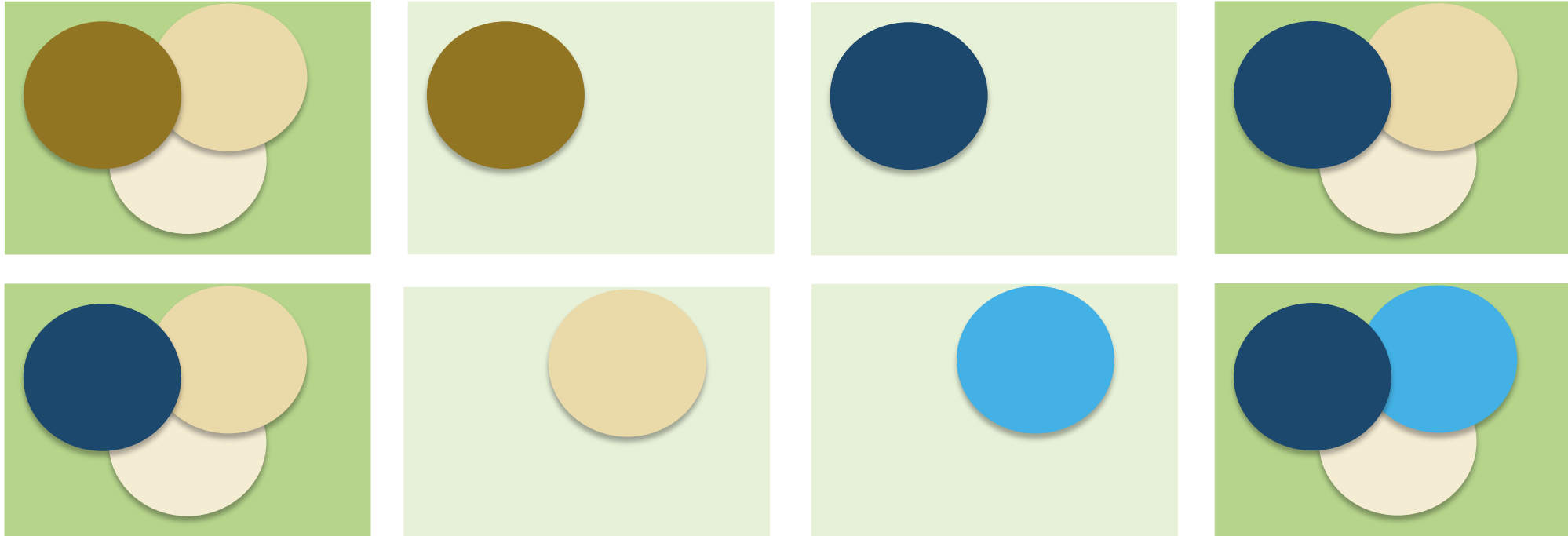
- Build a separate environment for testing refactored module
- Copy over the module in this isolated environment

# Map from Here to There: On ramp plan2



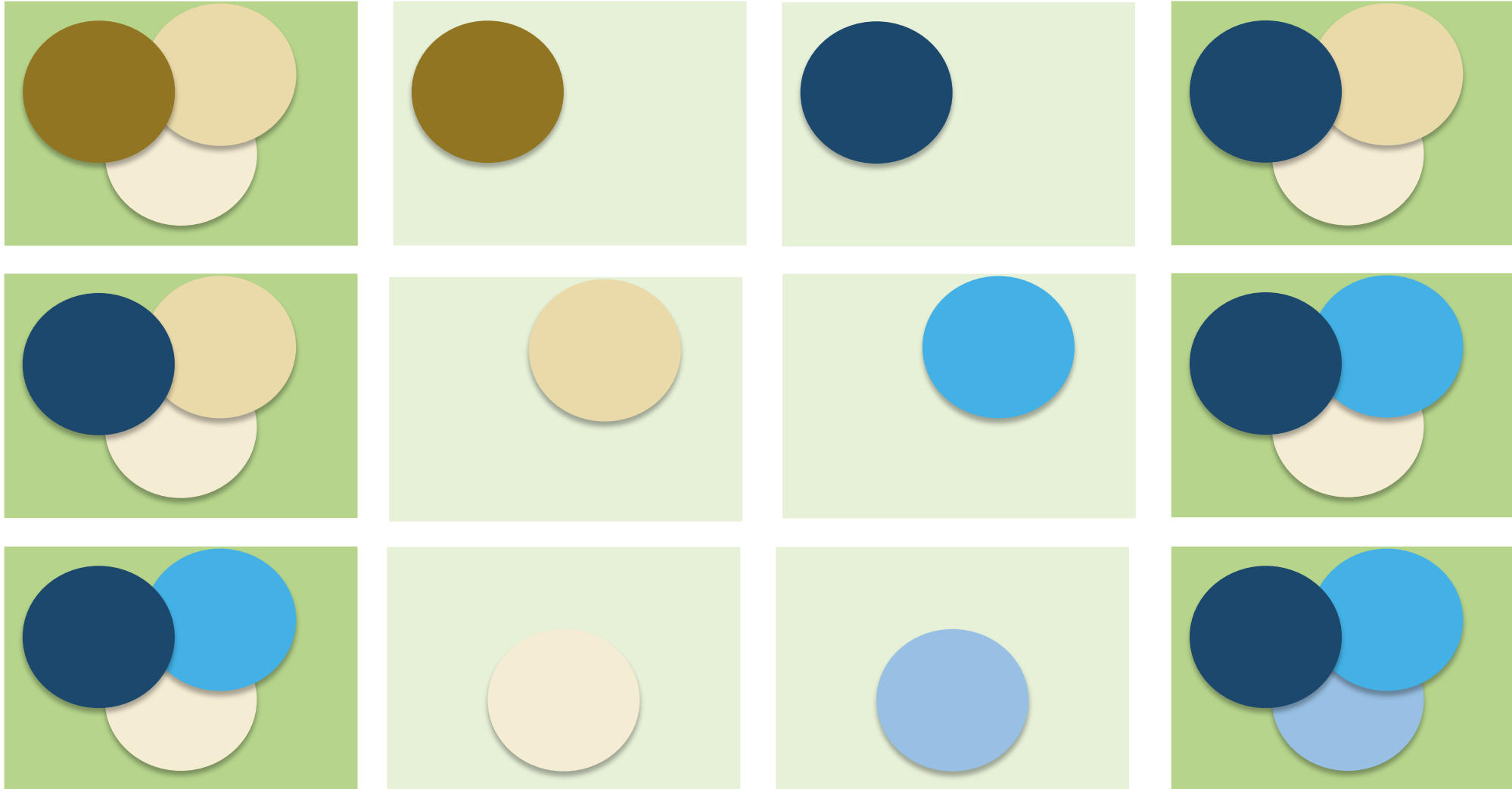
- Build a separate environment for testing refactored module
- Copy over the module in this isolated environment
- Put back refactored module

# Map from Here to There: On ramp plan2



- Build a separate environment for testing refactored module
- Copy over the module in this isolated environment
- Put back refactored module

# Map from Here to There: On ramp plan2



- Build a separate environment for testing refactored module
- Copy over the module in this isolated environment
- Put back refactored module

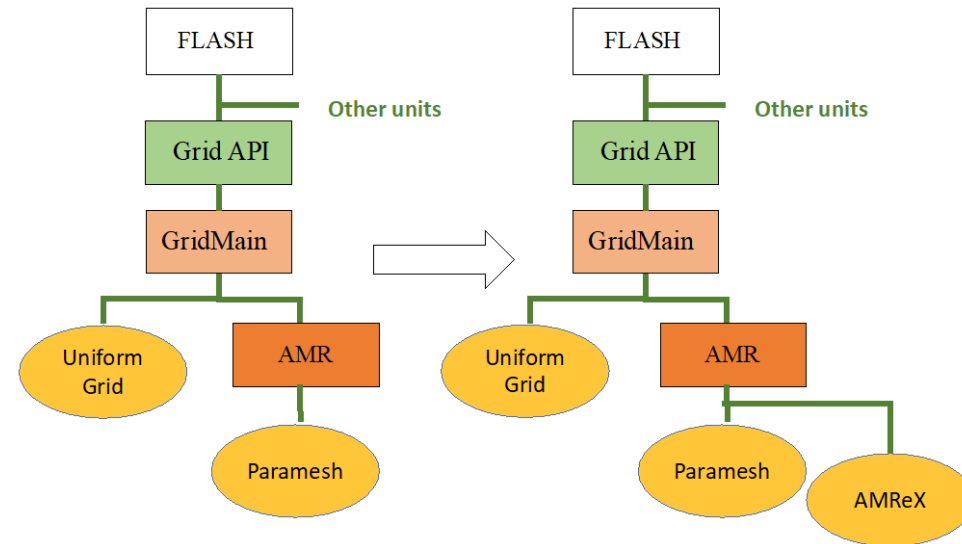
# A Real-World Example: FLASH to Flash-X

## Refactoring to supporting a different AMR library

**Goal:** Replace Paramesh with AMReX

**Plan:** Getting there from here

- On ramping
- Design
- Intermediate steps
- Realizing the goal





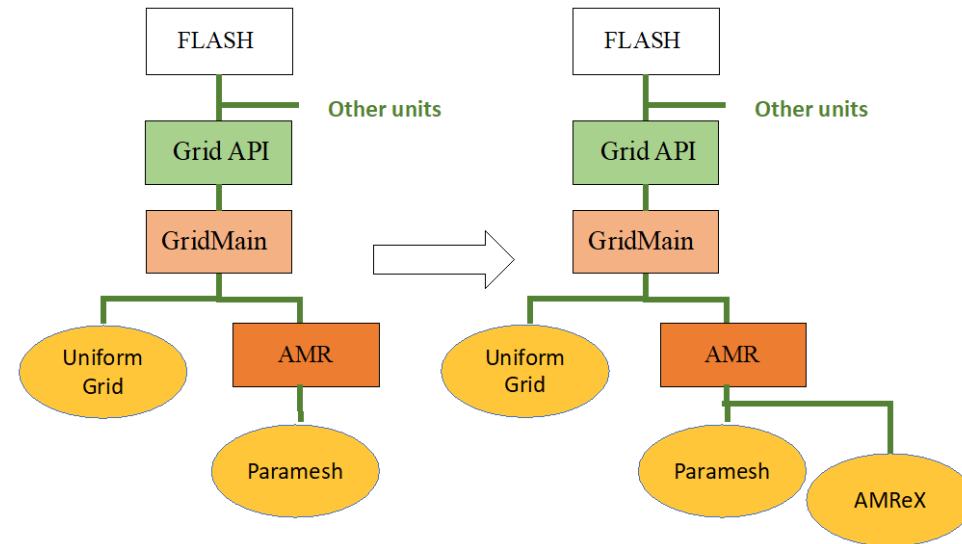
# A Real-World Example: FLASH to Flash-X

## Refactoring to supporting a different AMR library

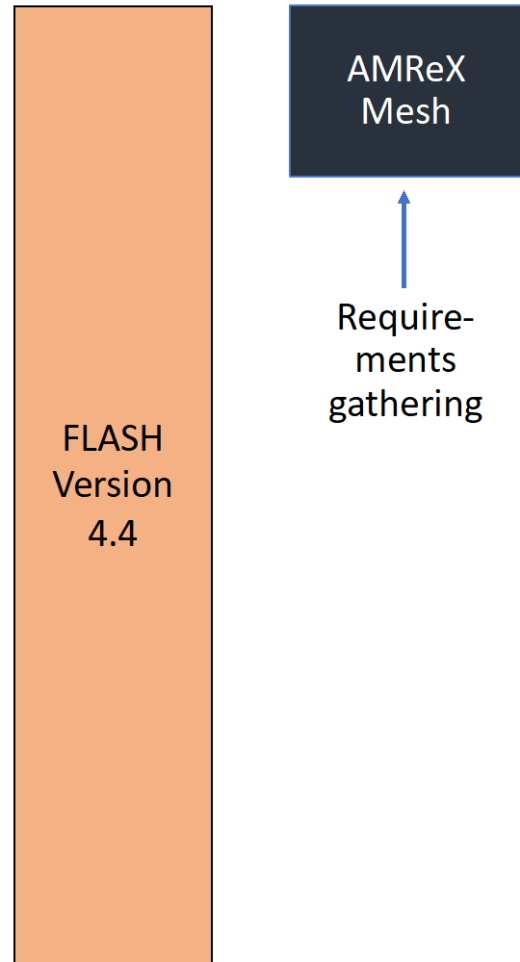
**Goal:** Replace Paramesh with AMReX

**Plan:** Getting there from here

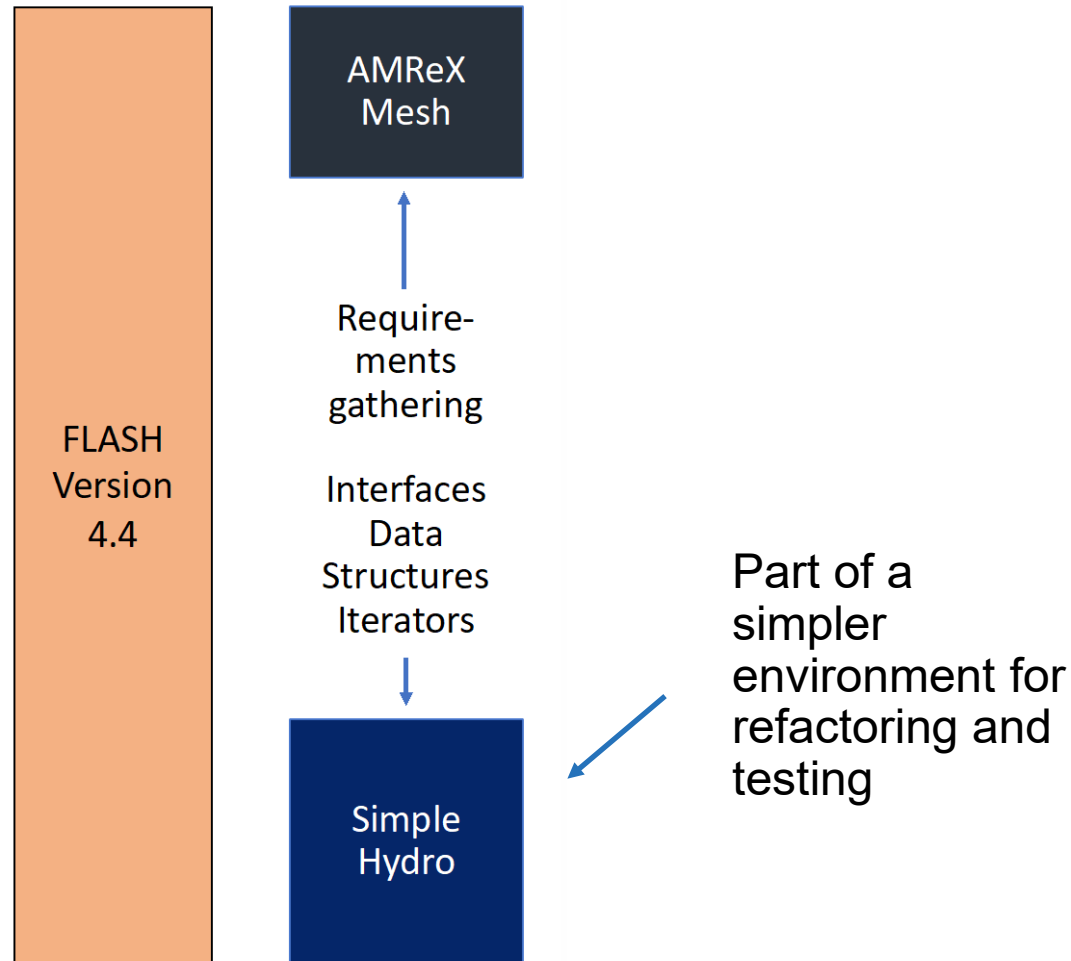
- On ramping
  - Design
  - Intermediate steps
  - Realizing the goal
- 
- Cost estimation
    - Expected developer time
    - Extent of disruption in production schedules
  - Get a buy-in from the stakeholders
    - That includes the users
    - For both development time and disruption



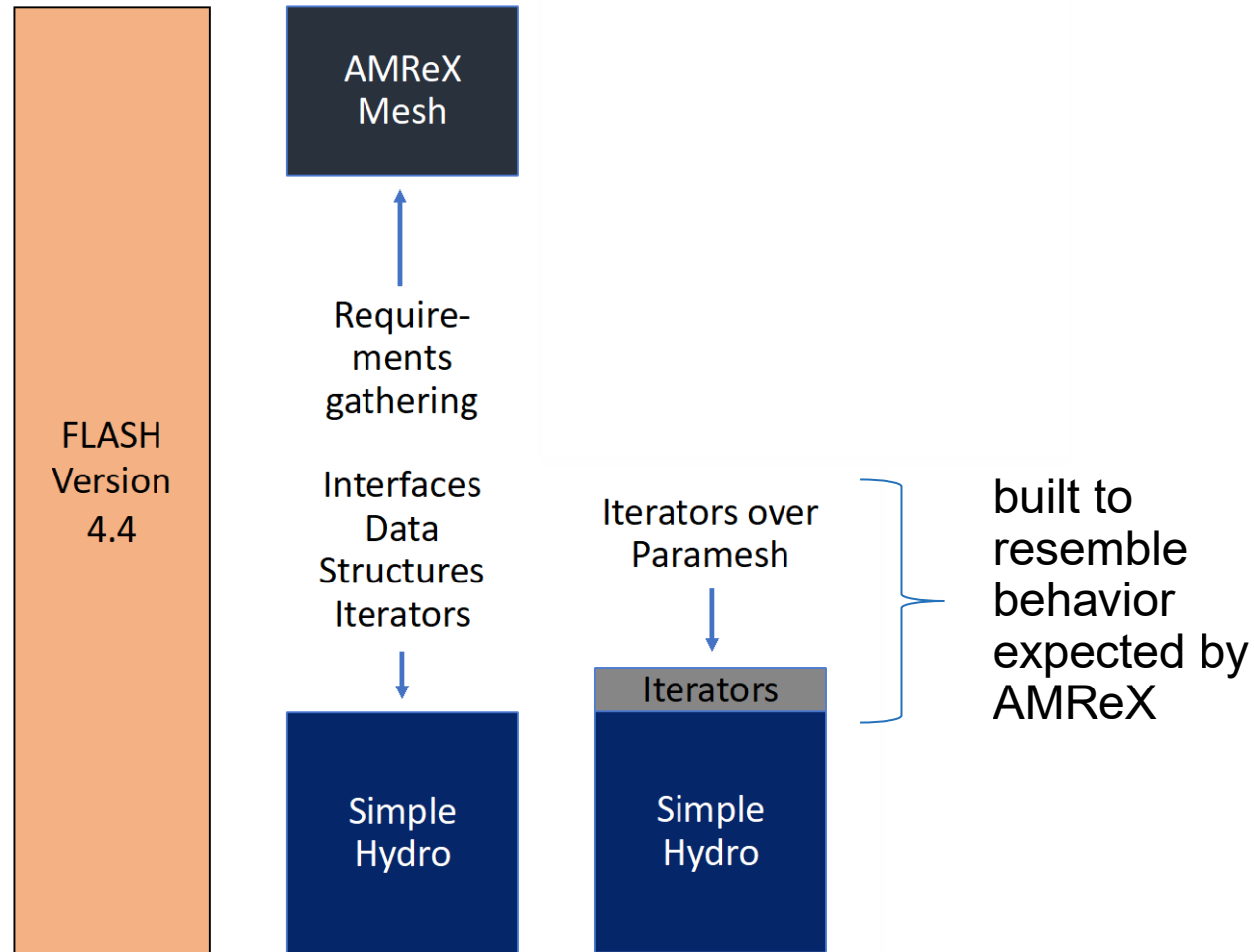
# Steps in the Flash-X Refactoring : a mix of strategies



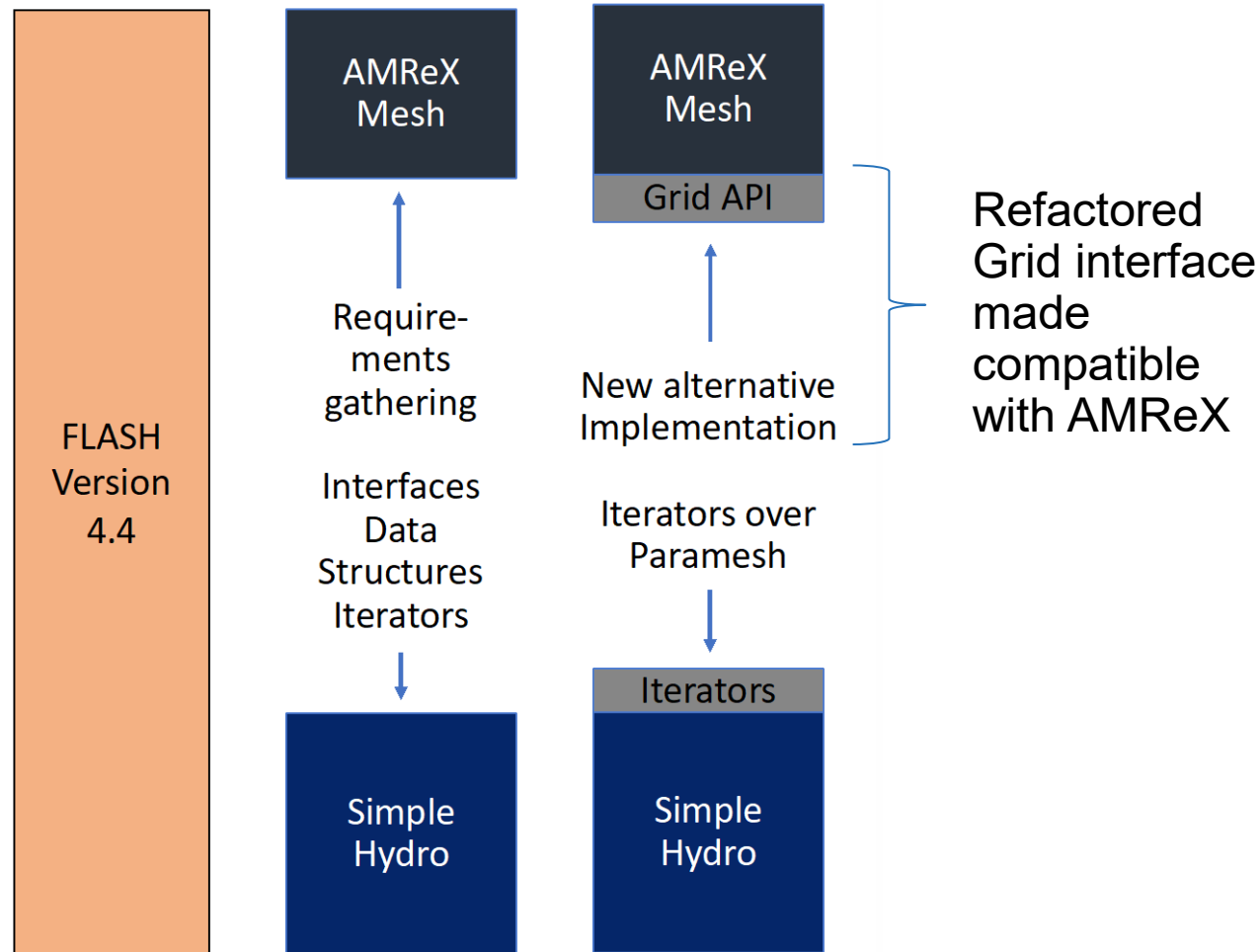
# Steps in the Flash-X Refactoring : a mix of strategies



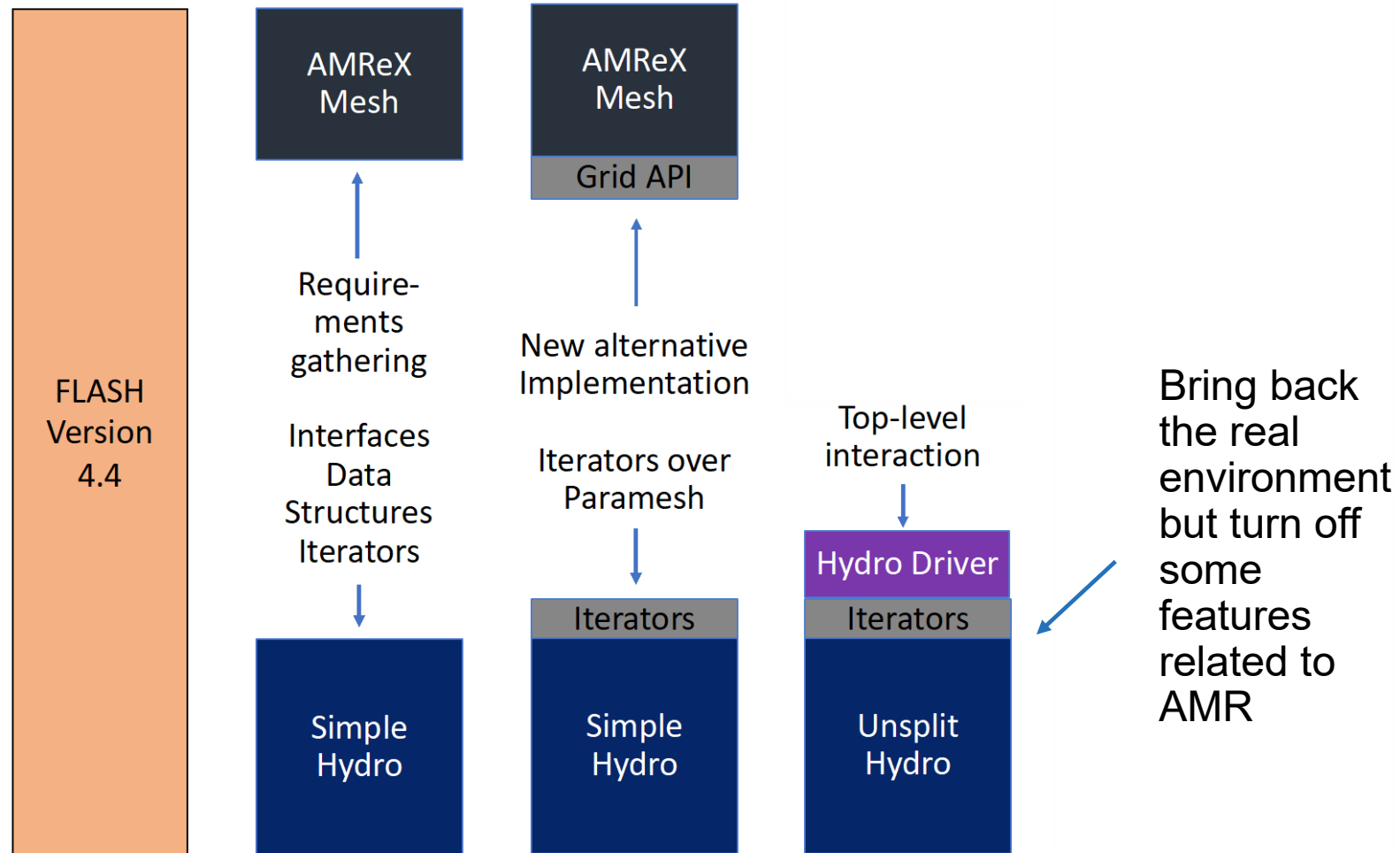
# Steps in the Process



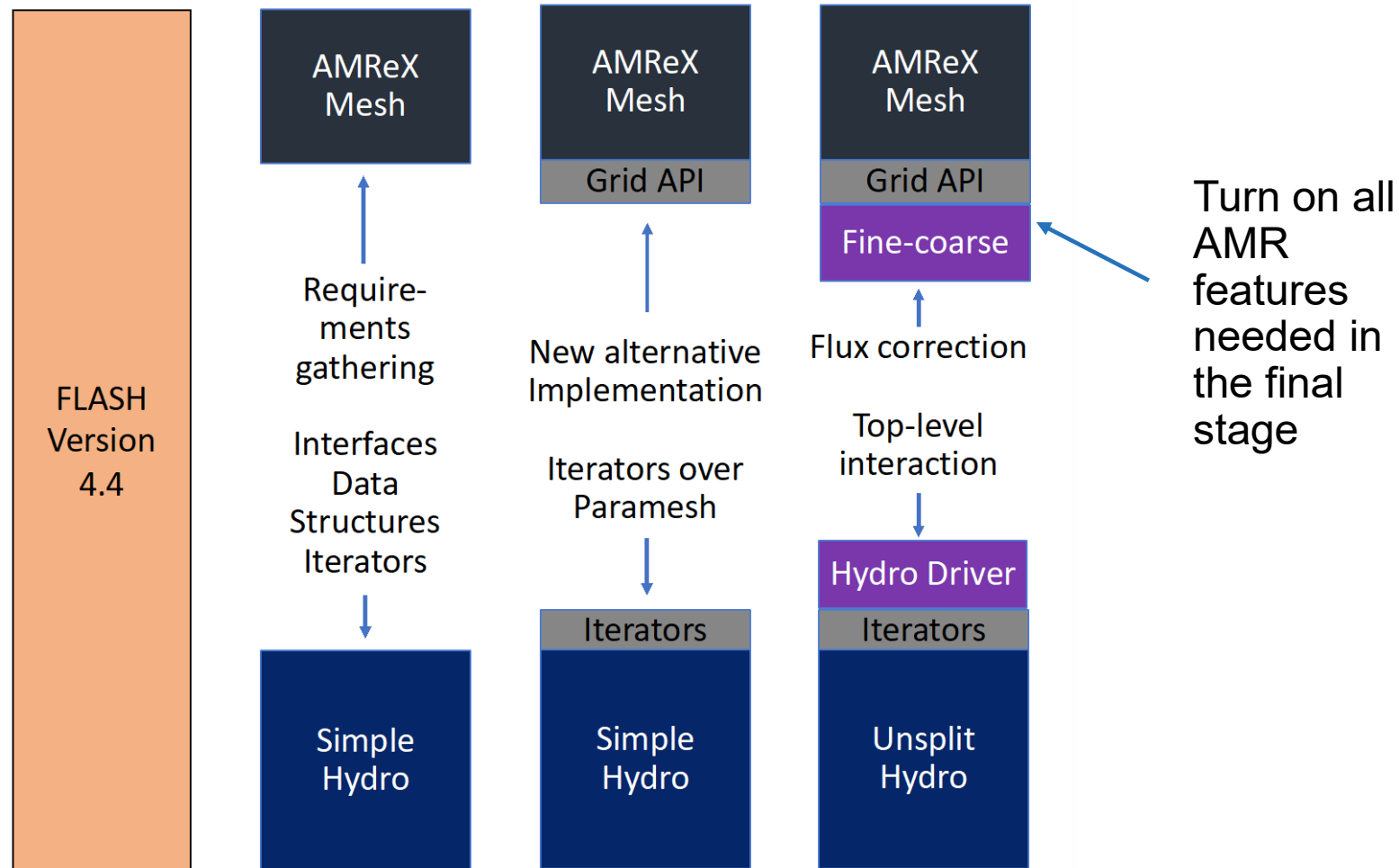
# Steps in the Process



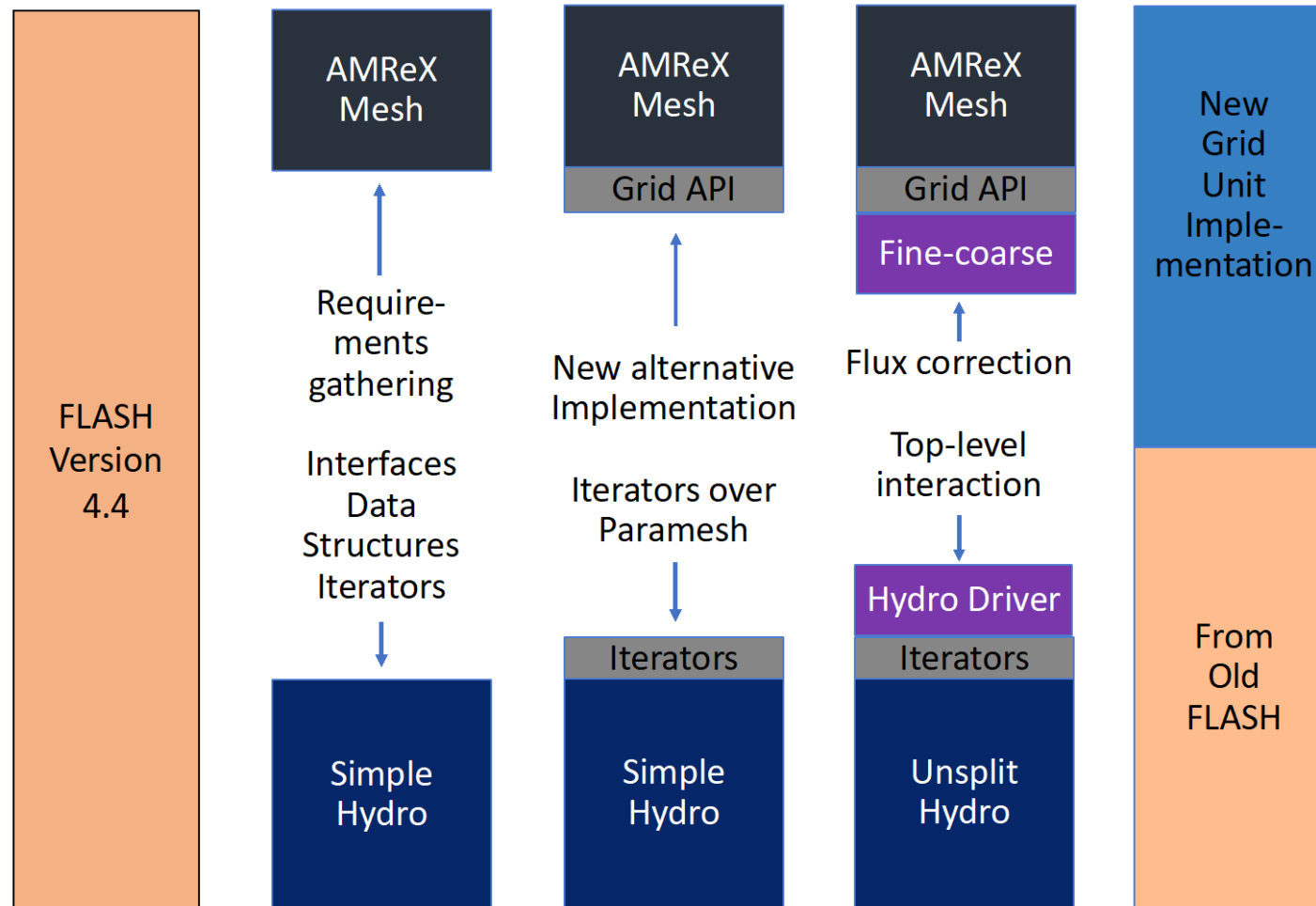
# Steps in the Process



# Steps in the Process



# Steps in the Process





# To Have a 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