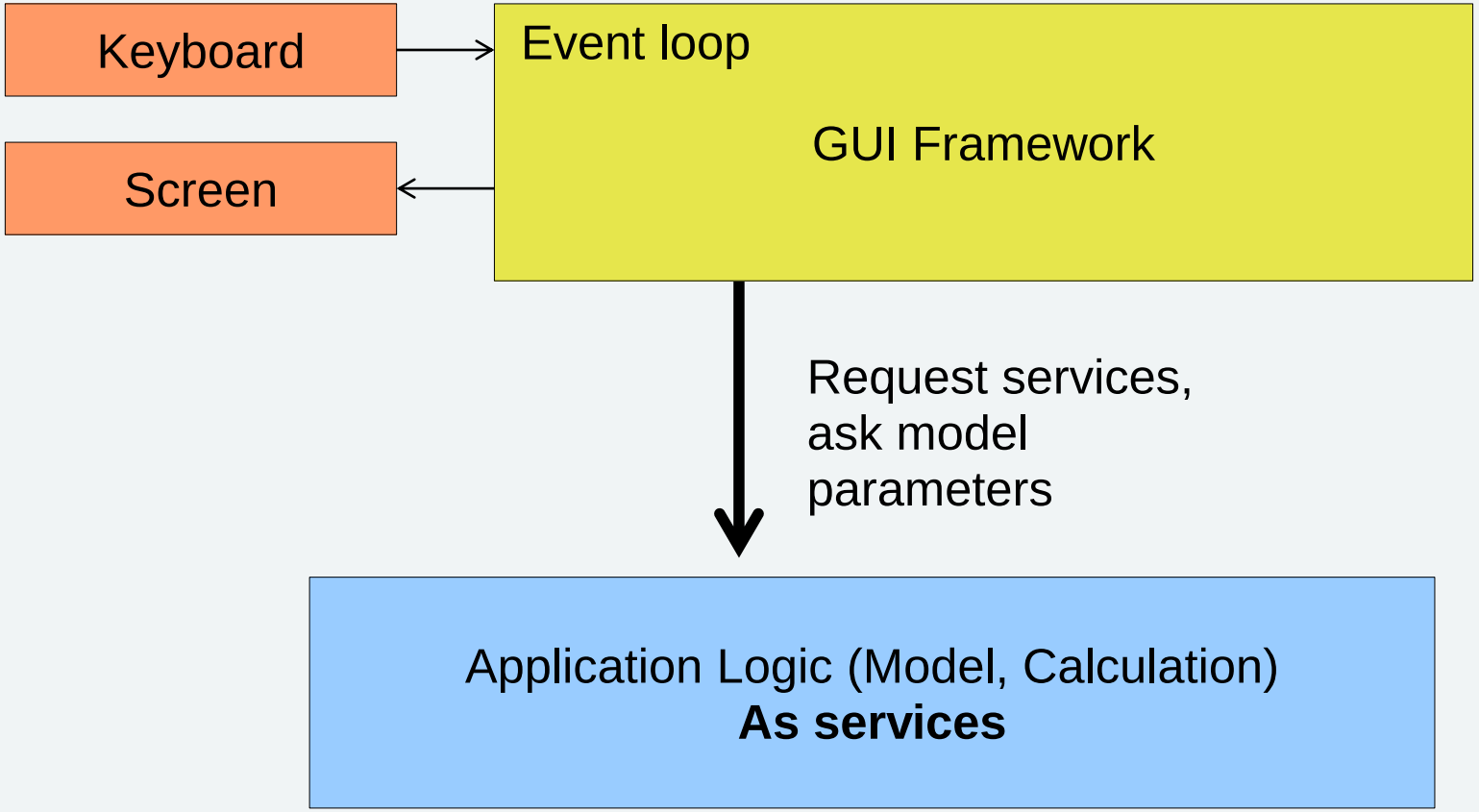
NAPA target architecture

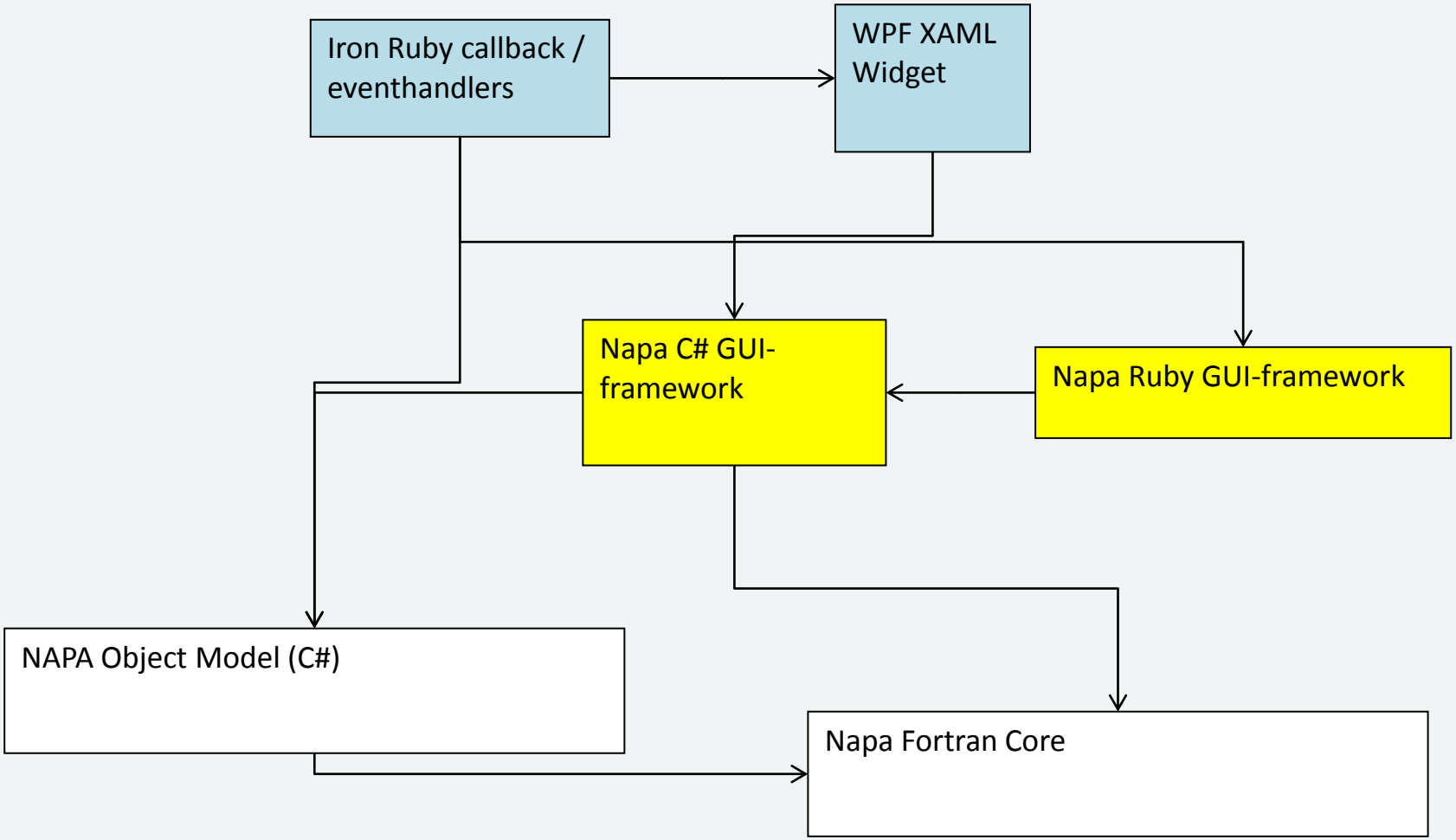
Where we want to go and how it's proceeding

Layering as it has been done in the latest two decades or so



Application logic independent / oblivious of the UI.

The target architecture



Language choices

- Not everything is best done in Fortran ;)
- Have to leverage the legacy but take advantage of newer languages
  - Higher productivity
  - Something fairly mainstream
    - Developers
    - Support
    - Libraries
- Minimize the role of Napa Basic
  - Productivity, libraries, support, documentation

Language choices

- First **Java** was selected but some years later abandoned for **C#**. This has left us with Java legacy we want to get rid of.
- **Iron Ruby** as the GUI callback language
  - Simpler than C# to learn and use for casual developers
  - Easier transformation of present callback codebase
- **XAML** for declarative definition of GUI

NAPA Object Model

- Abstract present functionality with class wrappers to allow OO based access to the functionality
  - Also for implementing new functionality
- OO based systems are not without their share of problems. We'd love to have those problems instead of the ones we have now.

Language interoperability

- Custom(ized) code generators enable easy access between Fortran, C and C# (from any to any) enabling selecting the right language for the job at hand
  - Iron Ruby code can easily call C# (and vice versa)
- Proof of concept implementation of handling Fortran objects from C# / Iron Ruby
  - To enable effortless integration between UI and core

How to (slowly) reach the target state

- Refactoring
  - **The boy scout rule**
    - Leave the camping ground cleaner than it was when you got there
  - Introducing named constants
  - Routine mass renaming
  - Cleaning up control flow (remove GOTOs)
  - Extract routines to make the huge routines smaller
  - ...

How to (slowly) reach the target state

- Architectural refactoring
  - Currently removing the layering violations, i.e. business logic does not ask for more input
- Unit tests
  - In Fortran and Ruby
  - Coverage still low but steadily growing
  - Often hard to write tests for a small piece of code
    - Global state
    - Huge (multi-responsibility) routines
    - High coupling

How to (slowly) reach the target state

- Replace custom solutions with off the shelf ones when feasible
  - E.g. we recently replaced custom memory allocation implemented in Fortran 77 with the one provided by C runtime (POSIX)

How to (slowly) reach the target state

- Technology workshops / internal training
- Communication essential
- Spread knowledge of architectural conventions, best practices etc.

Difficulties with the wetware

- Resistance / nonwillingness to use approaches like
  - Structured types
  - Named constants
    - E.g. `3` vs `STRING_RECORD`
  - Descriptive names
    - E.g. `CH17` vs `CH_UPCASE`
- Resistance to refactoring
  - ”If it’s not broken, don’t fix it”

