Unifying Workflows with the Strangler Fig Pattern

Brian Weir[1], Frederick Gabelmann[2], Christina Holt[3], Paul Madden[3], Emily Carpenter[3], Naureen Bharwani[3]
[1]Raytheon/EPIC
[2]Element 84/EPIC
[3]CIRES/NOAA GSL

Introduction

- Developing any large software system is complicated, particularly supporting a broad range of users
- Numerical weather prediction systems in particular tend to be tightly coupled and highly labor-intensive
- Implementing changes and updating documentation even among components or apps in a single system is difficult
- Unifying these systems systematically through a shared interface improves the user experience and code maintainability
- A gradual approach is especially important to avoid disruptions to users and other developers

Challenges In Numerical Weather Prediction

Given the sheer breadth of options in Numerical Weather Prediction (NWP), preparing and configuring a model can be a laborious task

It is necessary for the user to choose compile-time options and ensure that required libraries are installed

Parameters must be added or adjusted in any number of ASCII configuration files

Config files must be processed by the user by running additional scripts

The workflow manager if available must be configured by the user

Unifying With Structural Patterns

- Manual user input creates bottlenecks that lower task efficiency, increase risk and affect the time required to learn a new code base
- NWP codes and even apps within a particular code base are similar in terms of user intent, but vary notably in required interaction
- We can reduce the complexity in several ways:
  a. Breaking the steps down into generic tasks
  b. Identifying commonalities among configuration files
  c. Maintaining useful tools with a consistent user experience
  d. Preserving existing functionality without a loss of capability

Why the “Strangler Fig”?

- We could just rewrite existing code
- This entails more overhead than expected to replicate the existing functionality and maintain both systems during the transition
- Instead, we can incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.
- As features from the legacy system are replaced, the new system eventually replaces all of the old system's features, strangling the old system and allowing you to decommission it.

https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig

Strangling Code In Existing Systems

Benefits

1. **Reduces your risk** when you need to update things
2. **Starts immediately** to give you some benefit piece by piece
3. Allows you to push your changes in small modular pieces, **easier for release**
4. Ensures **zero down time**
5. Generally **more agile**
6. Makes your **rollbacks easier**
7. Allows you to **spread your development** on the codebase over a longer period of time

https://developer.ibm.com/articles/cl-strangler-application-pattern-microservices-apps-tr/

Strangling Code In Existing Systems

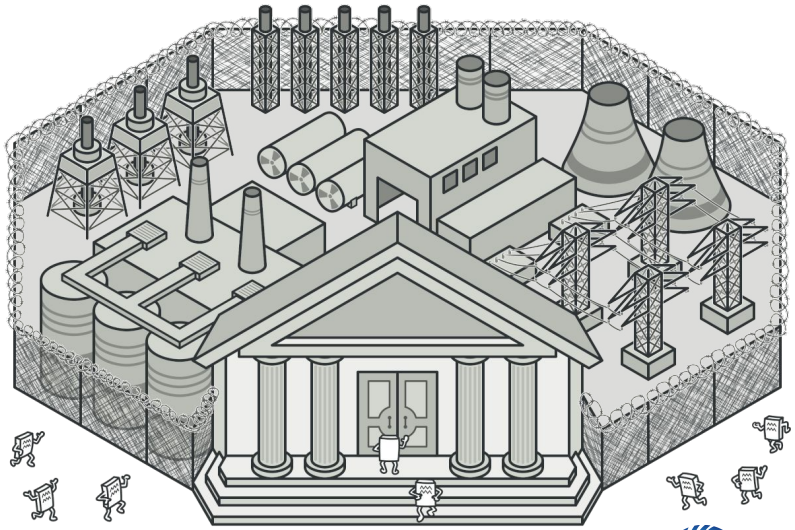
Steps

1. **Transform** — Create a parallel new code base, but based on more modern approaches.
2. **Coexist** — Leave the existing code where it is for a time. Redirect from the existing code to the new one so the functionality is implemented incrementally.
3. **Eliminate** — Remove the old functionality from the existing code (or simply stop maintaining it)

https://developer.ibm.com/articles/cl-strangler-application-pattern-microservices-apps-tro/

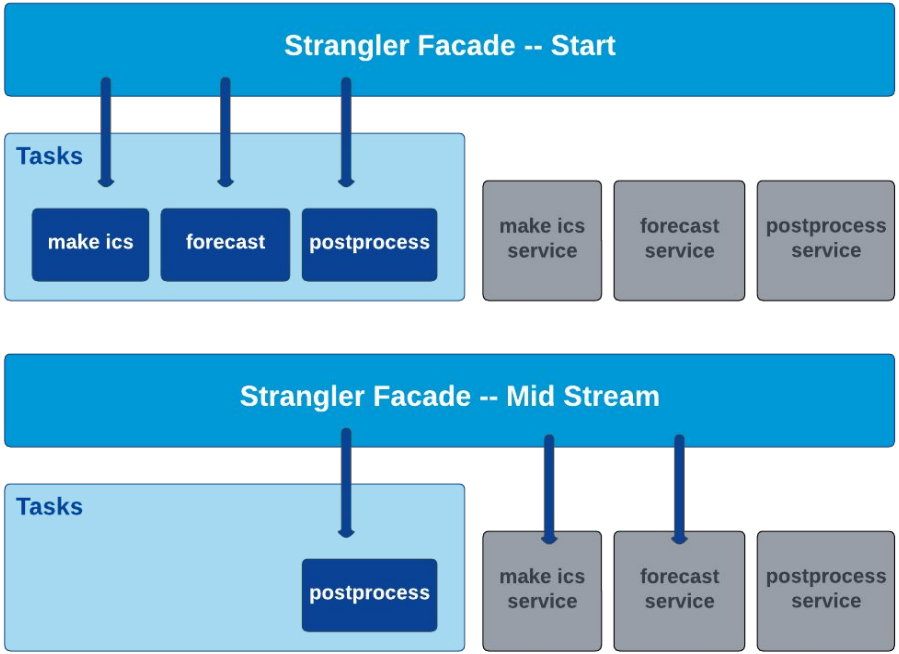
Now why a “Facade”?

- The facade provides a simplified interface to complicated code
- The additional layer allows us to implement the strangler fig pattern without affecting the user experience



https://refactoring.guru/design-patterns/facade

Diagram of a Strangler Facade



https://www.redhat.com/architect/pros-and-cons-strangler-architecture-pattern

An Example of the Strangler Fig in Practice

- One example, simplified here, is using decorator calls to redirect references within the facade
- Here, the decorator allows us to not just switch between old (OldArtifact) and new (NewArtifact) methods, but also run both and compare
- We can change methods without affecting the interface, and roll back if necessary

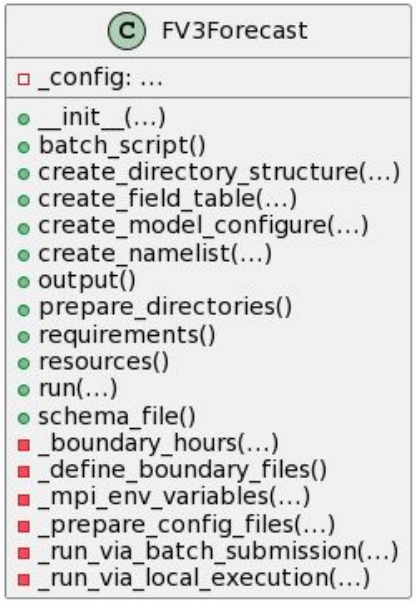
```
from .old.artifact import OldArtifact
from .new.artifact import NewArtifact
from strangler import *

@strangled_method("add_junit_evidence", use=NEW_MAIN)
@strangled_property("created_at", getter=NEW_MAIN)
class Artifact:

  def __init__(self, flow, docs):
    self.old = OldArtifact(flow, docs)
    self.new = NewArtifact(flow, docs)
```

https://www.kosli.com/blog/how-to-strangle-old-code-using-python-decorators/

The Strangler Fig Facade in the Unified Workflow Tools

- The Unified Workflow, following the Strangler Fig Pattern, will comprise three essential subsystems that work together to deliver an end product given user-defined settings
  - Configuration Subsystem
    - Responsible for ensuring proper interfaces to the Workflow Manager and standalone tools to interface with the existing scripts for their configurations
  - Workflow Manager
    - Interface with existing workflow managers to improve compatibility across apps
  - Component Drivers
    - Replace existing run scripts

UW Tools - Component Drivers



- Each step to run a particular model configuration is specified within the driver for that configuration
- The process can then be handled entirely by UW Tools
- The job can be run from the Command Line Interface (CLI) without manual user commands for each step
- Additional configurations will be added later

Summary

- The Strangler Fig Pattern offers a gradual approach to refactoring, reducing risks and enabling continuous delivery.
- The Facade Pattern plays a key role in simplifying integration between legacy and refactored components.
- Embracing these patterns allows for a smoother transition and unification of workflows.
- Both operations and research benefit from the consistency, flexibility and simplicity of this pattern