Spacecraft Flight Software Design Patterns Discovery

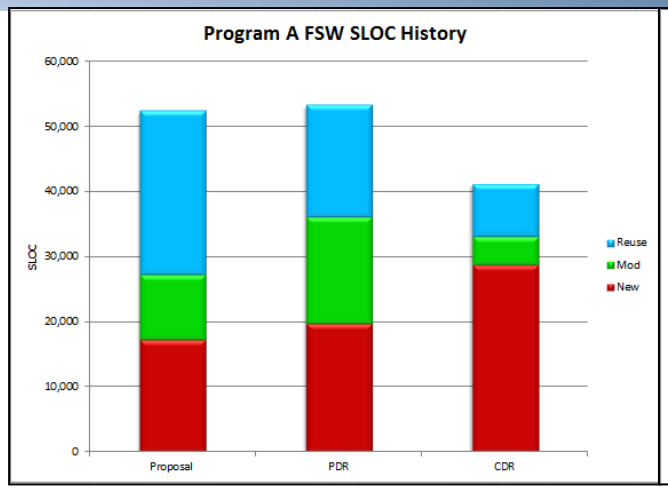
Michael Phillips – Lockheed Martin Fellow, R&D Principal Investigator

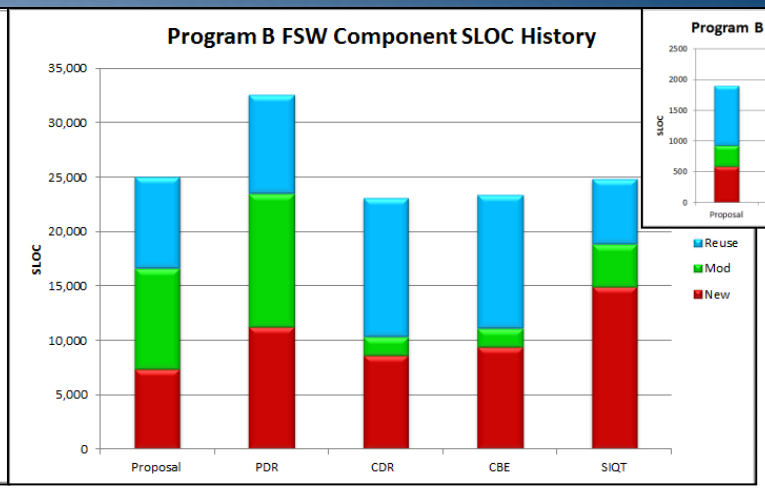
Amy Mok – Lockheed Martin, R&D Technical Lead

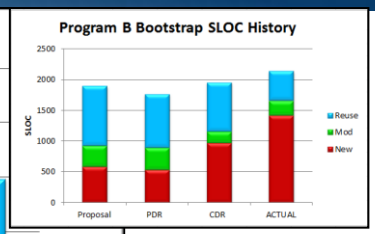
Agenda

- Overview
  - Business Case
  - Introduction of the R&D Project
- COT Tools for Pattern Detection
  - Diagrams
  - Metrics
  - Sample Usage
- Sample Results
- Obstacles & Challenges
- Summary

Business Case for SW Reference Designs







Observations

Reuse **Mod** **New**

1. Software Estimates for NEW code have significant growth from program proposal, through PDR/CDR, and to final implementation
2. Software Estimates for code reuse (Modified + Untouched) migrates rapidly from ReUse (100% untouched) to Modified to New during a program lifecycle

Hypothesis

1. A significant portion of NEW Software developed on programs is very similar in design to other programs
2. Most NEW software (code) is preceded by a corresponding SW Design Activity
3. Programs are efficient at implementing SW to program-unique standards given a mature design

Providing SW Development teams and individual SW Engineers with validated heritage reference designs for common capabilities will lead to improved productivity

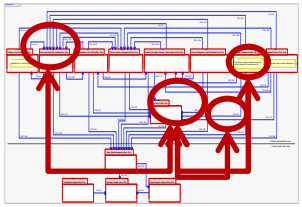
Identifying and proliferating Best Practice reference designs will lead to more commonality across our spacecraft FSW product lines and variants

Reference Designs for future SW product lines using COTS tools for design pattern discovery

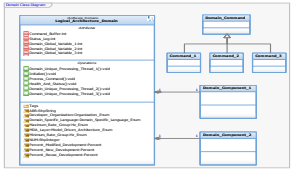
Discover Common Design Patterns across existing TRL9 Spacecraft Flight Software using COTS Tools. Implement, Test, and Deliver TRL 4 FSW Modules for FSW Reference Designs in a few domain areas commonly re-designed and re-implemented during programs.

UML
Unified Modeling Language

Task 1: FSW Design Commonality Discovery



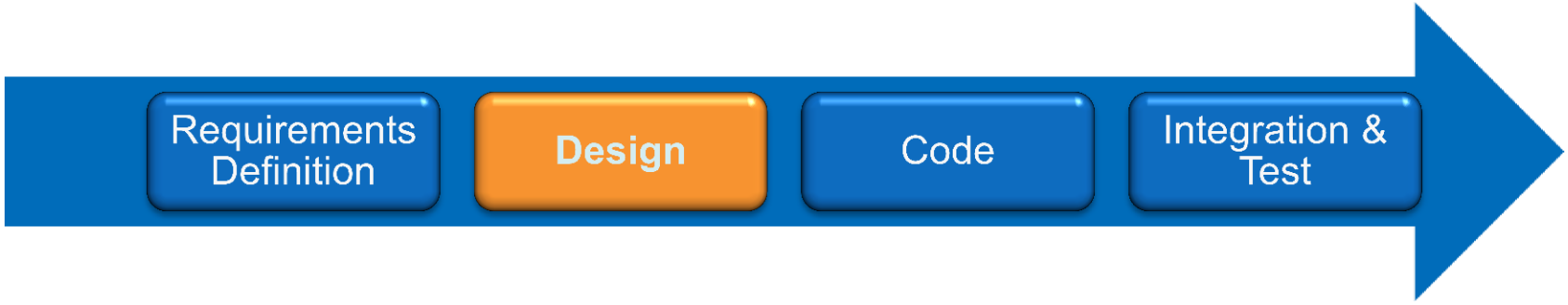
Task 2: Develop Best Practice SW Reference Designs



Task 3: Implement, Test, and Deliver TRL 4 FSW Modules

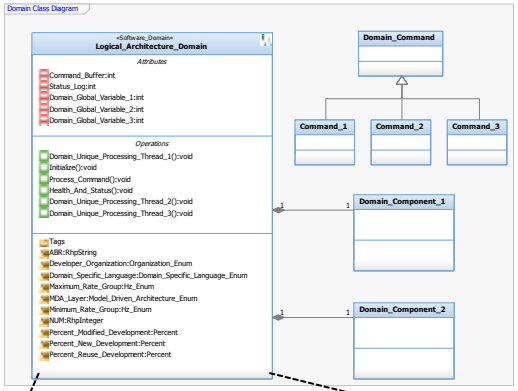
What is a Design Pattern?

- A software pattern is a portion of a end product that is repeated or replicated in multiple SW products, where the software products may be modeling products and or source code products.
- Patterns are architected into software products and or introduced independently during software design when software engineers choose similar or identical solutions to implement similar functionality.



Types of Software Patterns

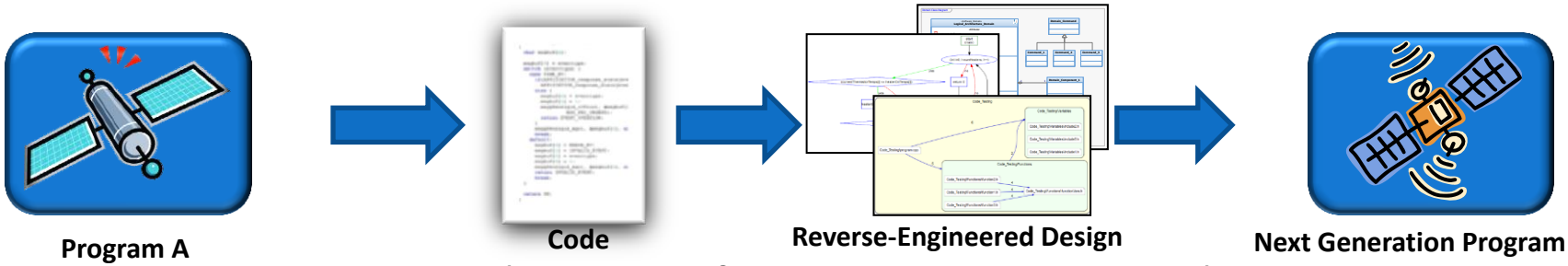
1. Architecture Patterns:
   - Patterns applied across all software domains
     - “Base” classes used or extended by each domain
     - Patterns for creating constants, parameters and variables
     - Patterns for creating command and telemetry messages
     - Patterns for creating initialization and processing threads
     - Patterns for fault management
2. Functional Patterns:
   - Patterns identified within multiple software baselines
     - Requirements common to multiple software designs
     - Functionality common to multiple software designs
3. Logic and Algorithm Patterns:
   - Patterns identified within multiple software baselines
     - Processing threads common to multiple software designs
     - Algorithms common to multiple software designs
     - Equations common to multiple software designs
4. Implementation Patterns :
   - Patterns applied within one or more software domain
     - Modeling or coding solutions to implement logic, algorithms, and equations
     - C++ Templates / Ada Generics
     - Generalization & Extension



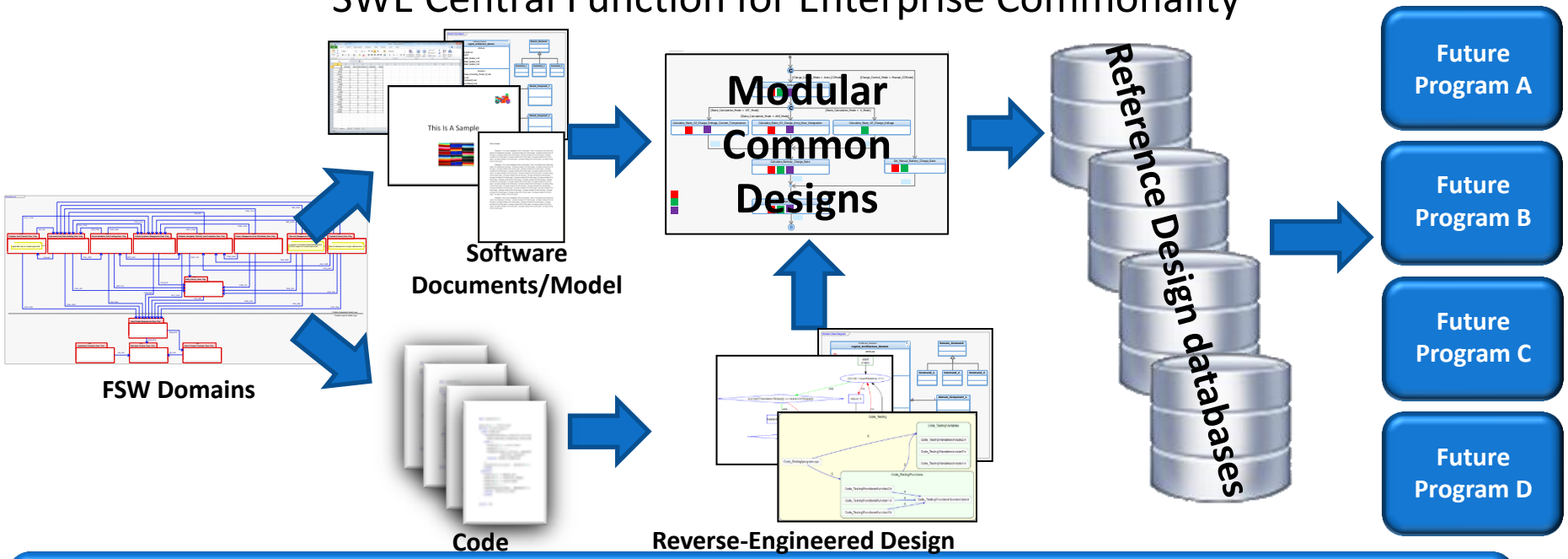
```
void Operation_Name()
  ...
  executable statements
  ...
```

Use Cases & 2013 Pilots

Reverse Engineering to Support Program Needs (Immediate)



SWE Central Function for Enterprise Commonality



The R&D return is achieved through shorter software development cycle and increase of design reuse (i.e., commonality)

Tools Selected to aid in Pattern Detection

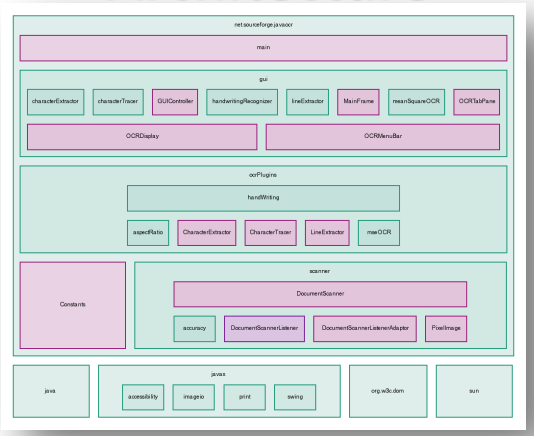
Tool Name	Vendor
Axivion Bauhaus Suite	Axivion
Together	Borland
Imagix 4D	Imagix
LDRA Testbed	LDRA Software Technology
McCabe IQ	McCabe Software
C/C++ Test	Parasoft
Ptidej	FOSS
Sotoarc / Sotograph	Hello2morrow
UMLStudio	Pragsoft
Understand	SciTools
Visual Paradigm	Visual Paradigm
Rhapsody	IBM

Tool Capabilities Matrix

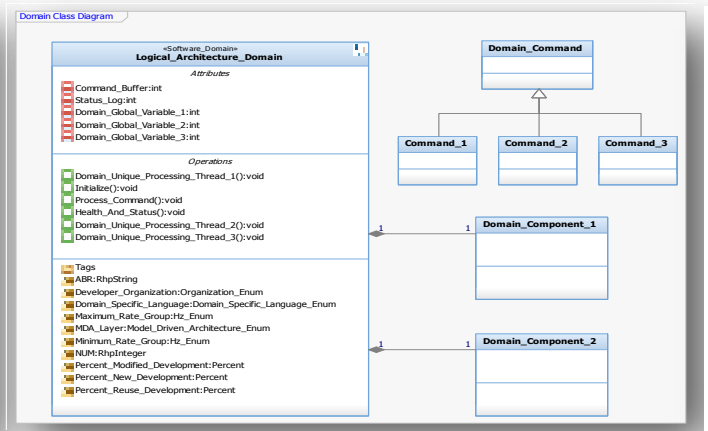
	Pattern Detection		Reverse Engineered Diagrams					Reverse Engineering			Metrics			Supported Languages		
	Architecture	Data Structures	Architecture	Class	Behavior	Dependency	Function Calls	Classes	Structures	Methods	Object Oriented	Complexity	Other	ADA	C	C++
Together	X	X		X				X	X	X						X
Imagix4D	X	X	X	X	X	X	X	X			X	X			X	X
LDRA					X		X				X	X	X	X	X	X
McCabe IQ	X	X			X		X	X	X	X	X	X		X	X	X
C/C++ Test											X	X	X		X	X
UMLStudio	X	X		X			X	X						X		X
Ptidej	X			X					X	X						X
Sotoarc / Sotograph			X			X					X	X	X		X	X
Understand				X	X	X	X				X	X	X	X	X	X
Visual Paradigm	X	X		X	X			X	X	X						X
Rhapsody				X				X	X	X					X	X

Reverse-Engineered Diagrams

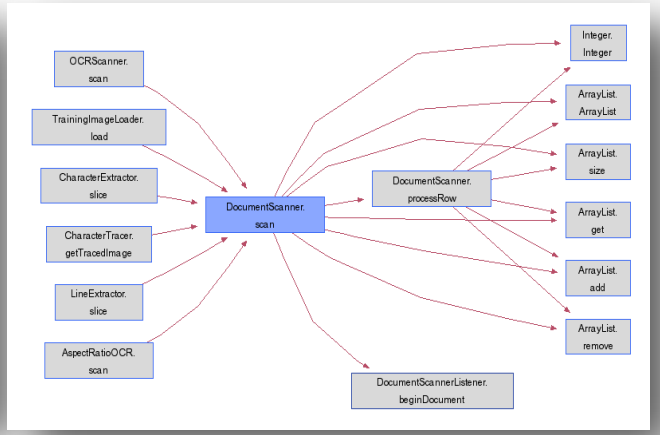
Architecture



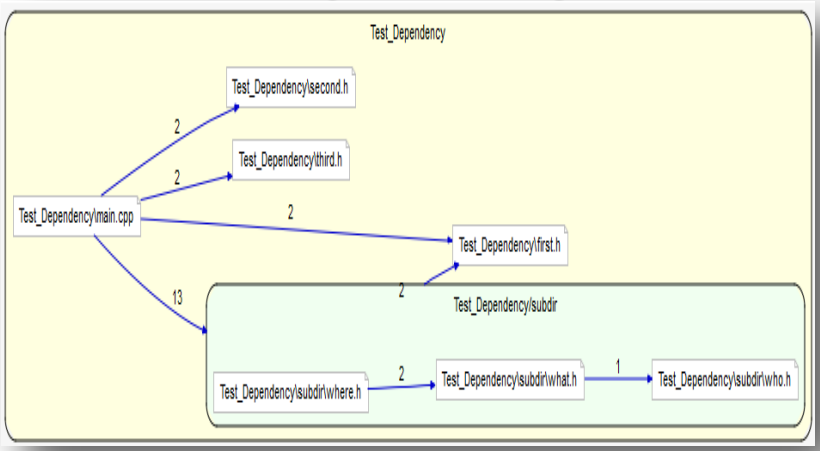
Class



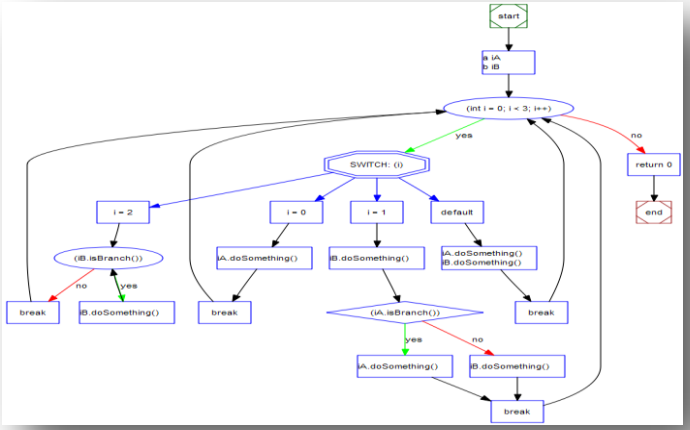
Function Call



Dependency



Logic



Metrics Extracted from Code

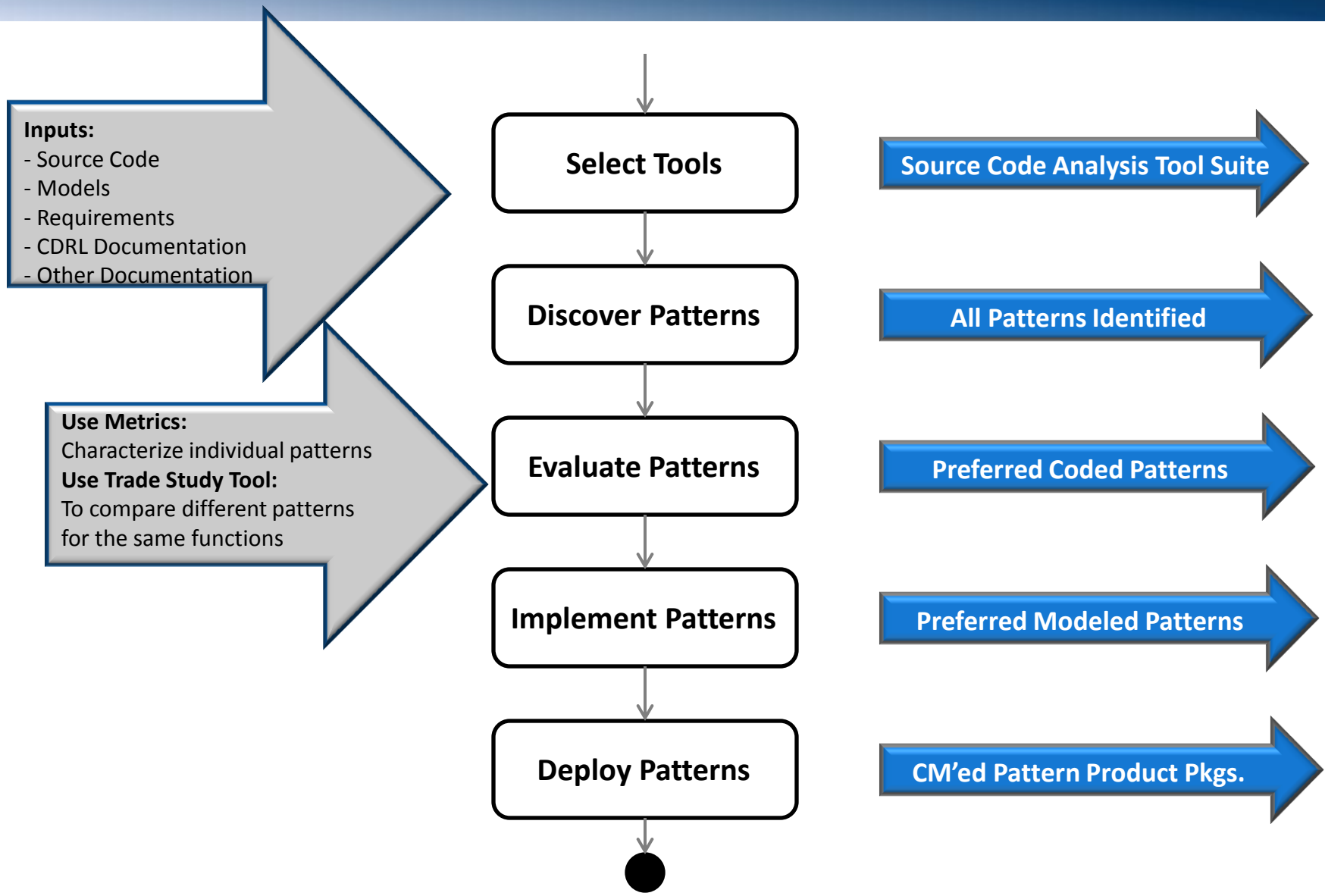
Metrics Show:

- Indication of which code baselines should be used to develop the reference design
- Quantitative differences between the multiple code baselines
- Assurance of the quality of the reference design that will be distributed

Some Examples:

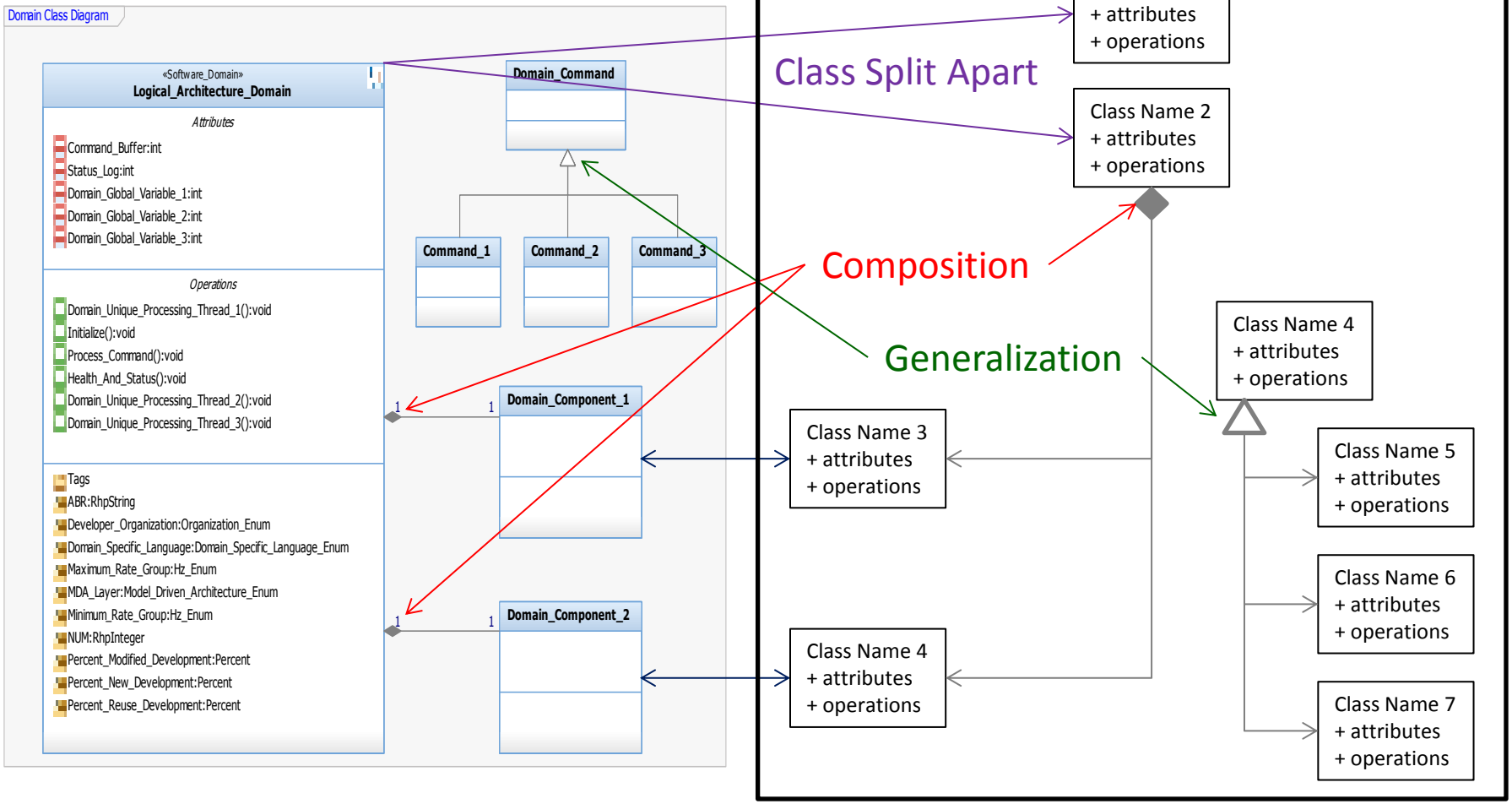
- Complexity (Cyclomatic, Essential, Halstead)
  - Difficulty in understanding, implementing, and testing decision logic
- (Lack of) Cohesion
  - How related the functions and functionality of a module are.
- Coupling
  - Degree to which each module relies on other modules
- Depth of Inheritance
  - How deeply modules inherit from each other
- SLOC
  - Amount of code necessary to implement functionality
- Many more…

End-To-End Process



Sample Detailed Design – Class Diagrams

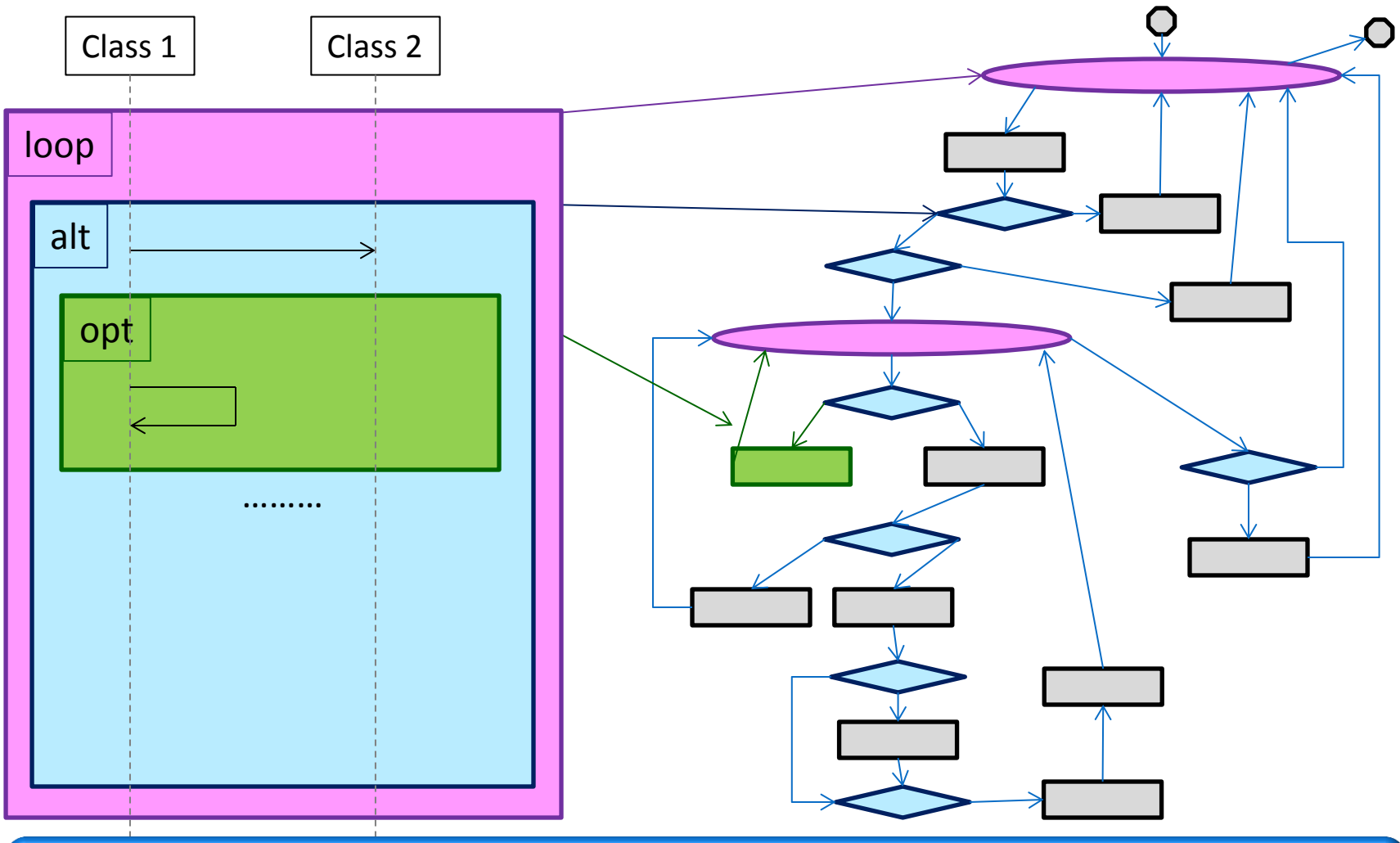
Detailed Design ***Reverse Engineered Design***



End User Artifacts: SW Architecture Diagrams - Differences between planned and actual can be discovered providing users with correct design for review and implementation

Sample Detailed Design – Behavior Diagrams

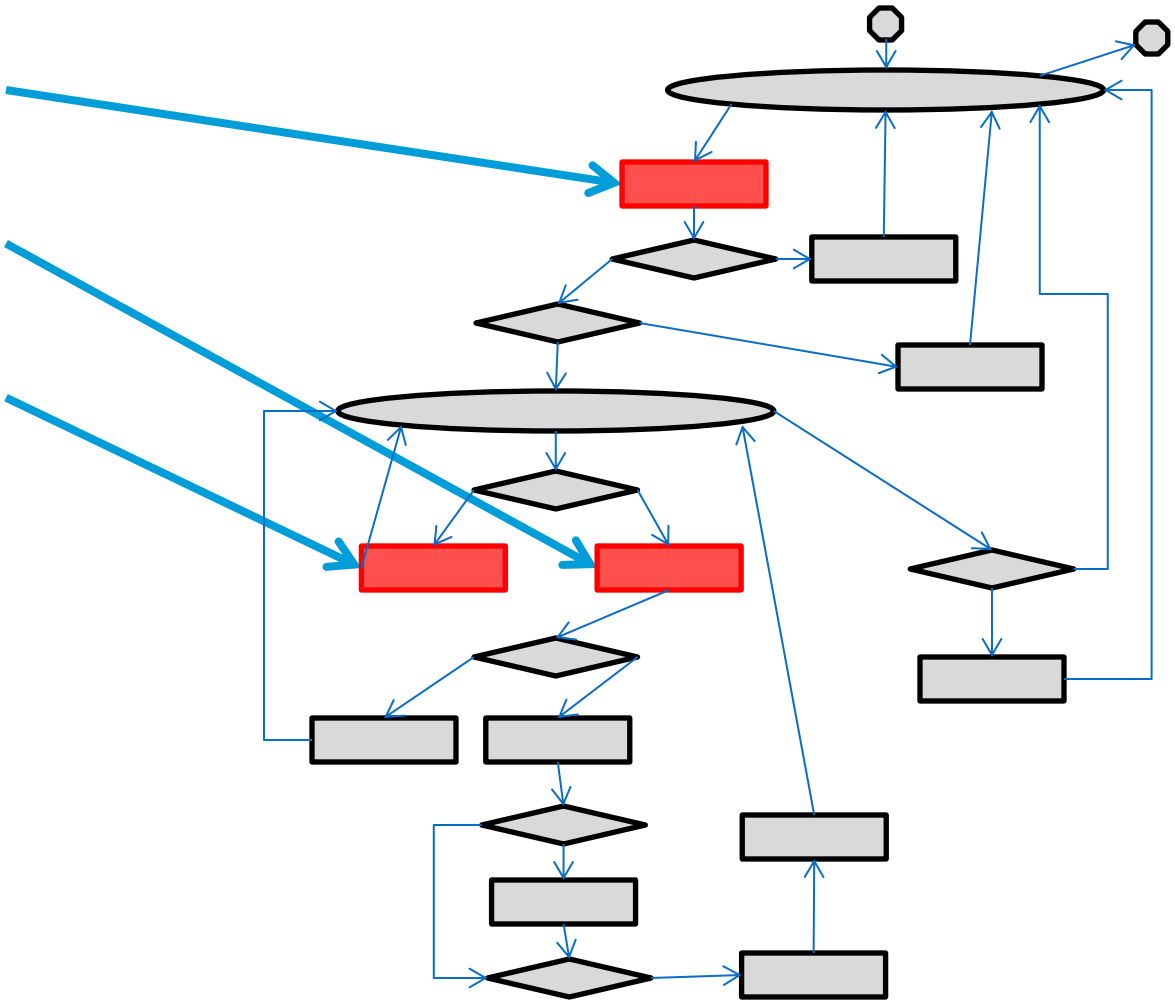
Detailed Design ***Reverse Engineered Design***



End User Artifacts: SW Logic/Algorithm Diagrams - Represents actual, detailed, unambiguous, and straightforward views of the software design

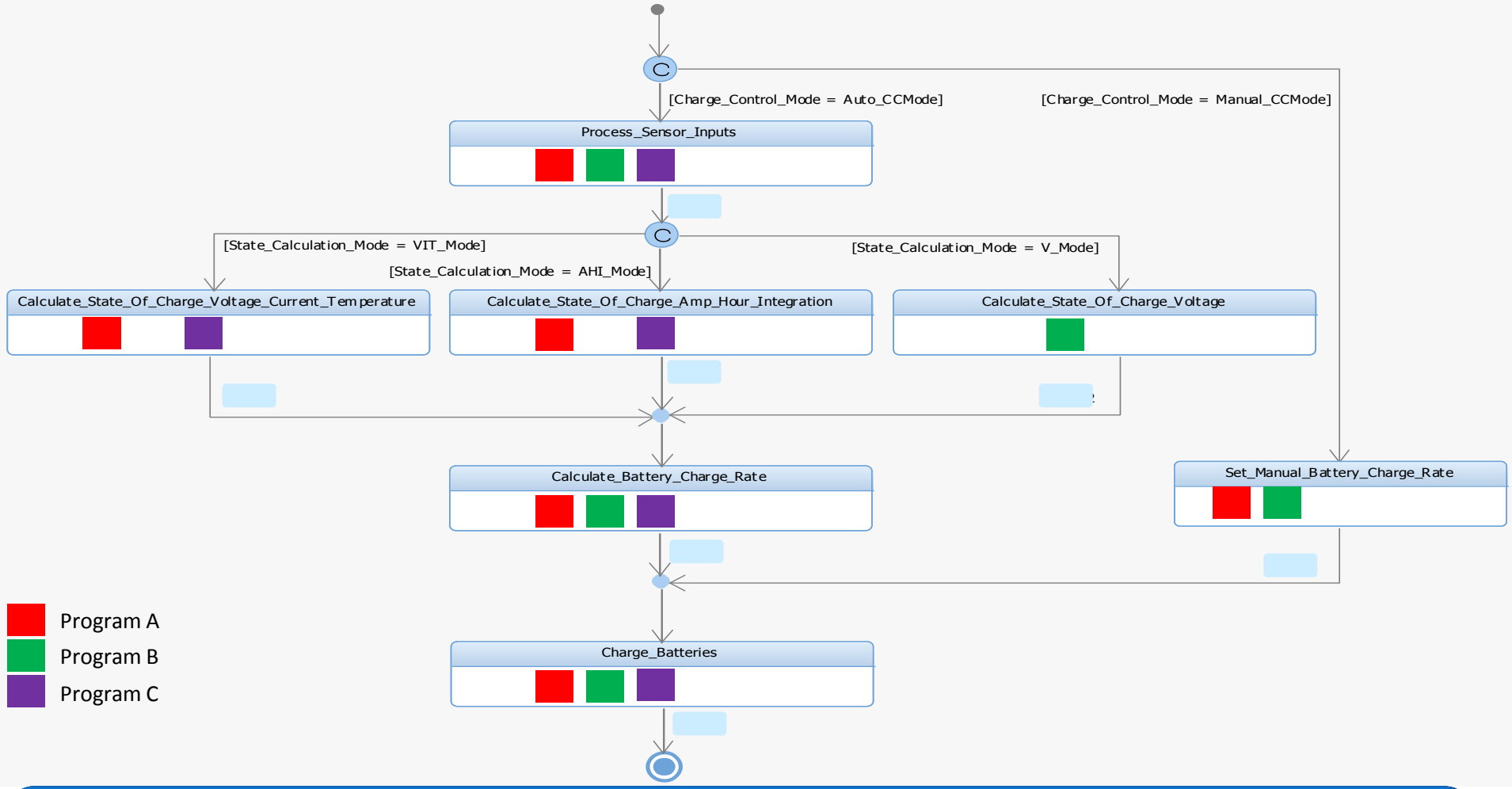
Sample Requirement Traceability

- SRS0001 – FSW shall …
- SRS0002 – FSW shall …
- SRS0003 – FSW shall …



End User Artifacts: Mapping to SW Requirements – Ability to easily map requirements to design, allowing program to determine new, modified, or reuse of SW design

Program Usage of Functionalities



Functionalities that most programs had implemented represent critical functions
Functionalities that only a few programs had implemented represent mission-unique functions

Obstacles & Challenges

- Usually difficult to obtain FSW and FSW Design Artifacts from heritage programs
  - Each program has their own process, priorities, time table, and restrictions
- Creating common designs can be tedious (the tools help greatly, but it's still a manual process)
- Program-specific design decisions must be filtered out from the common design
- Lots of interest for derived designs of a single heritage program, but need more interest in the designs derived from multiple heritage programs
- Comparing different designs can be subjective and difficult as each design has different capabilities

Summary

- We believe there is significant room to improve the productivity of the Flight Software Team by leveraging design reuse from existing programs
- COTS tools have a tremendous capability to efficiently reverse-engineer both software design and metrics that can augment program design artifacts
  - These tools have limitations that require both processes and manual effort to effectively use the artifacts produced by the tools
  - In some cases, the reverse-engineered design is more factual than the existing design artifacts, since it was derived for the actual implementation, not the planned implementation
- Providing reference design information to both individual contributors for specific SW domains, as well FSW technical leads and managers, needs to be organizationally institutionalized in order to effectively improve productivity on programs