Developing an Effective Software Product Architecture Using Systematic Software Engineering Practices

Richard F. Schmidt

STRATEGIC INSIGHT

Today's Software Industry Challenges

- Software Industry has an appalling history of success
  - Standish Group's Chaos reports:
    - ~ 30% success rate
    - SUCCESSFUL Programs averaged:
      - 70% schedule overrun
      - 50% cost overrun
      - Only 70% of features delivered
- Cause(?)
  - No emphasis on Software Design
  - Focus on "PROTOTYPING" (Cowboy coding, Agile, etc…)
  - Significant amount of "Rework" to fix design flaws *(because there is no coherent design)*

Today's presentation will address the application of Software Engineering Practices with an emphasis on DESIGN SYNTHESIS

Objectives of Software Engineering
(Personal Perspective)

- Develop a complete, effective and sustainable software product ARCHITECTURE which:
  - Embodies the data-processing characteristics
    - » Must address security features which protect Data Assets and System Vulnerabilities
  - Sufficiently detailed to permit efficient coding, Test & Evaluation, and Sustainment
    - » Hand-off the design specifications to ”Implementation Team” (Coders)
      - Software Technical Data Package
    - » Provide a complete “design-to” Software Bill-of-material (SBOM)
    - » Provides the software integration strategy & derived “Integrating Components”
  - Satisfies the Customer’s & Stakeholder’s Total Ownership cost objectives
    - » Development success Rate Improvement (On-Time, On-Schedule, Full Functionality)
    - » Structural Integrity to endure future modification & enhancement

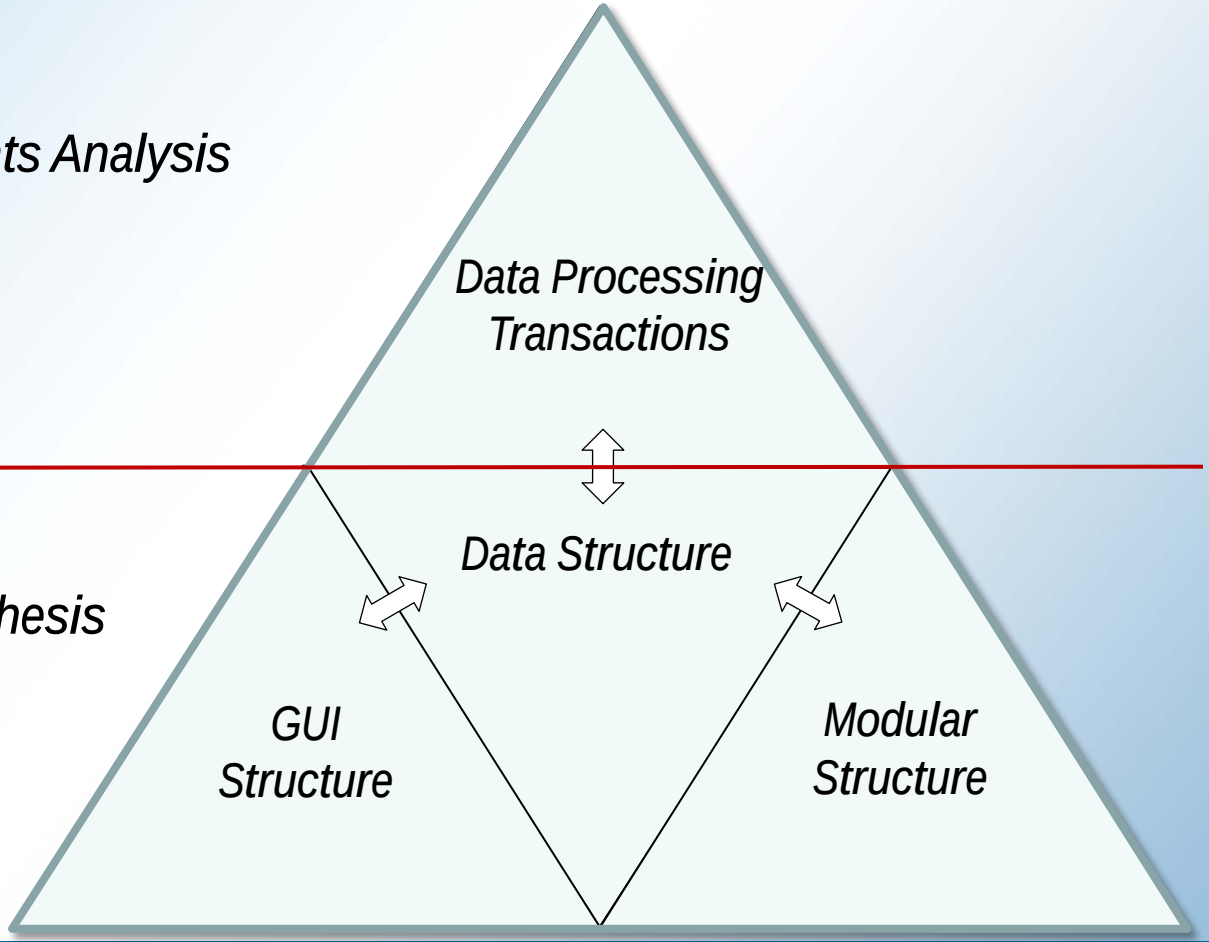
Apply Proven Systems Engineering Practices To Software Products

Software Engineering Practices
(Adapted from IEEE-1220)

- Software Requirements Analysis
  1. Capture & Model Data-processing Transactions (Behaviors)
  2. Develop Test Cases
  3. Generate Requirement Specifications
- Functional Analysis & Allocation
  4. Decompose Complex Behaviors (Functions)
  5. Combine and Assimilate Common Functions
  6. Generate Functional Specifications (Preliminary Design Document)
- Software Design Synthesis
  7. Identify Components
  8. Derive Structural "Features" necessary to define each Component
  9. Establish Component Integration Strategy
     - » Derived Requirements for additional "Integrating" Components/Features
  10. Generate Detailed Design Documentation

Software Architecture Structural Pillars
(Industry ratings represent my personal perspective)

- ✓ *Software Requirements Analysis*
- ✓ *Functional Analysis*
- ✗ *Software Design Synthesis*

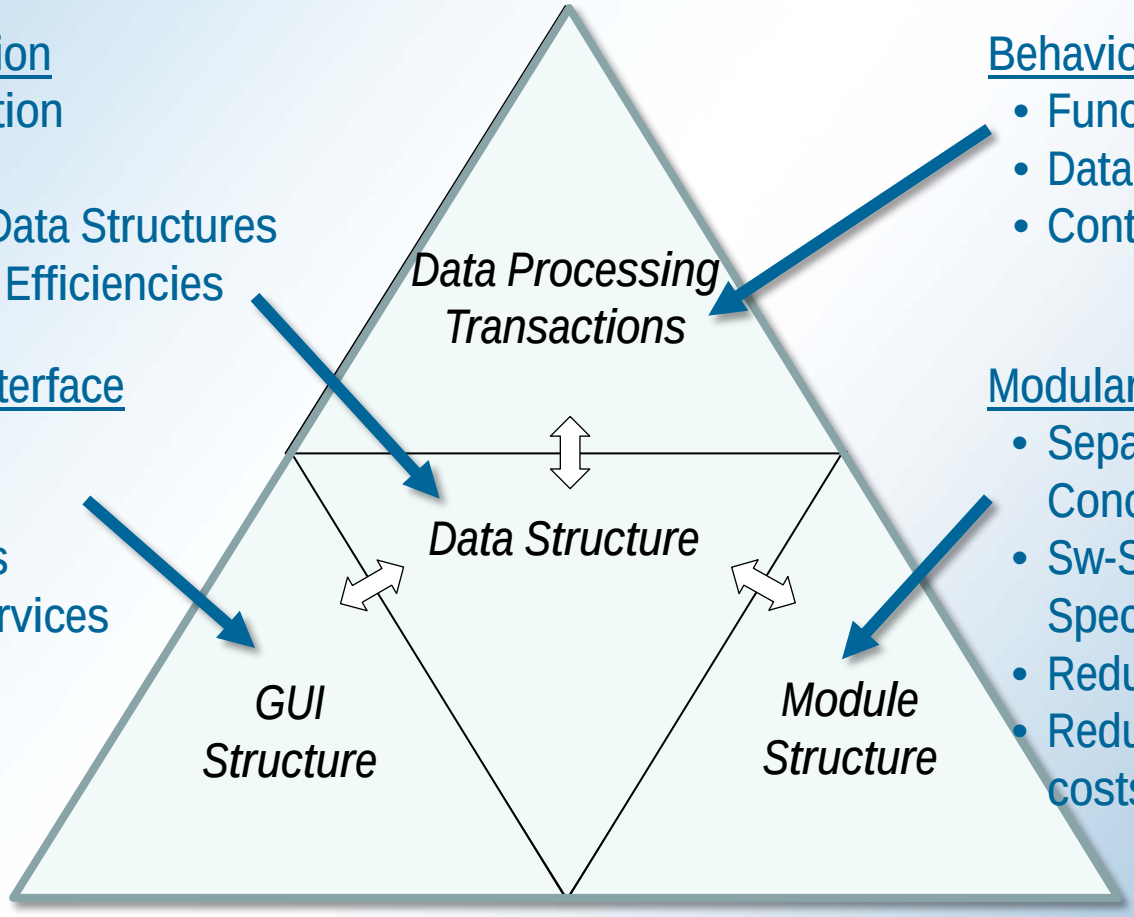


Software Engineering practices impose rigor and discipline to Software Development

Software Architecture Structural Pillars

Object-Orientation
- Class Definition
- Collections
- Compound Data Structures
- Data Search Efficiencies

Event-Driven Interface
- Forms
- Dialogs
- GUI Controls
- Run-time Services



Behavioral Models
- Function Flow
- Data Flow
- Control Flow

Modularity
- Separation of Concerns
- Sw-Sw Interface Specifications
- Reduce Complexity
- Reduce lifecycle costs

Software Engineering practices MUST adequately address the COMPLEXITY of modern programming languages

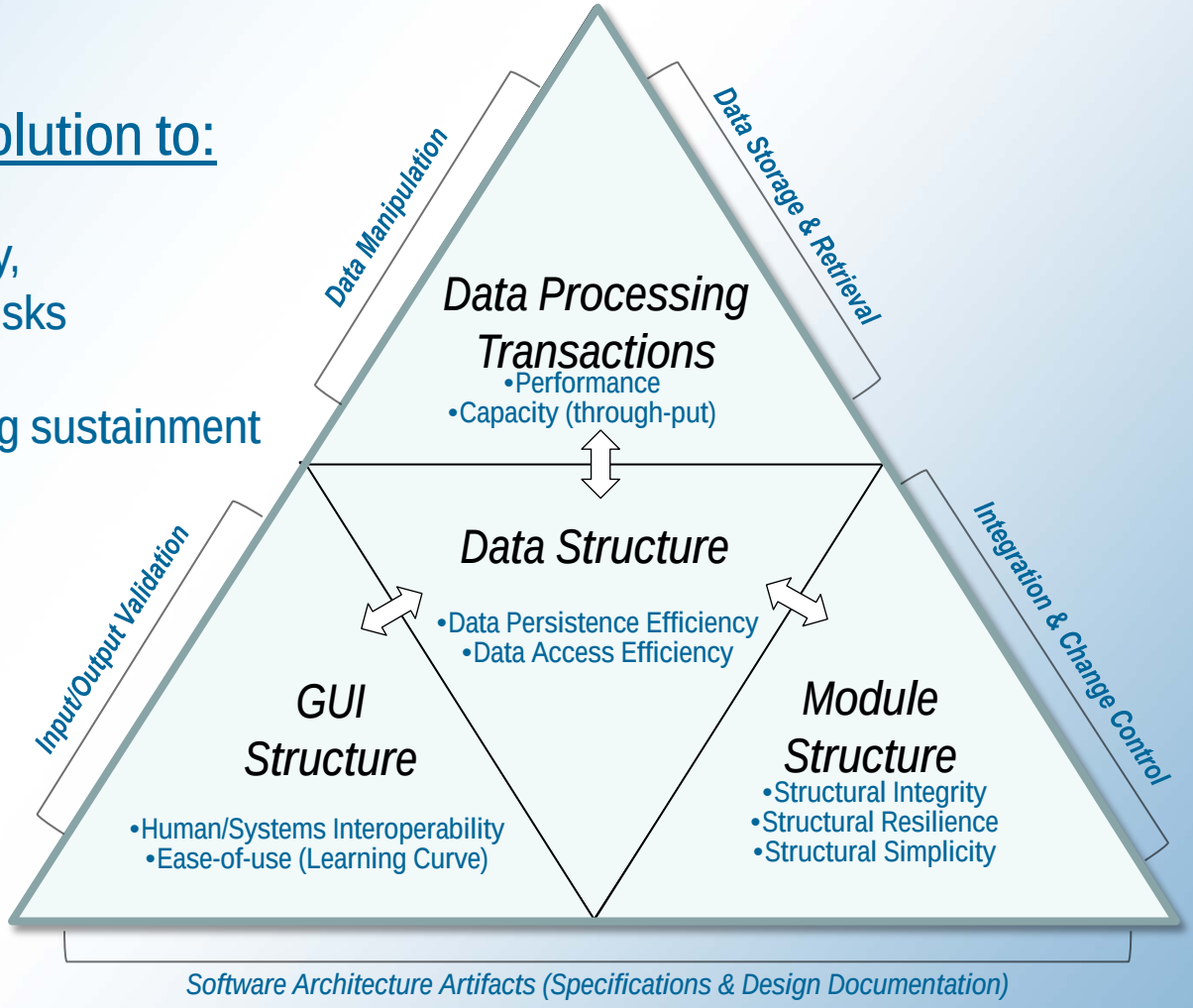
Software Architecture Structural Pillars

Establishes a balanced, comprehensive design solution to:

1. Improve product quality,
2. Reduce product complexity,
3. Minimize implementation risks
4. Eliminate scrap & rework
5. Increase adaptability during sustainment

GOAL:

- *On Time*
- *Within Budget*
- *Full Functionality*

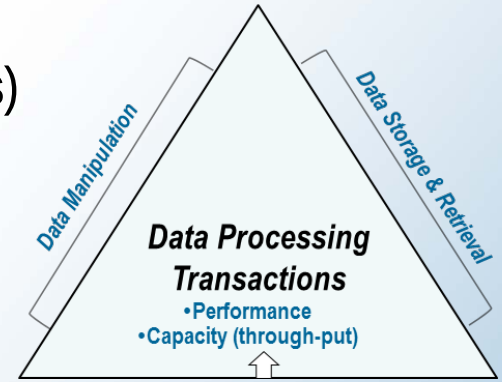


Makes Software Development more predicable & dependable

Software Engineering Practices
(Adapted from IEEE-1220)

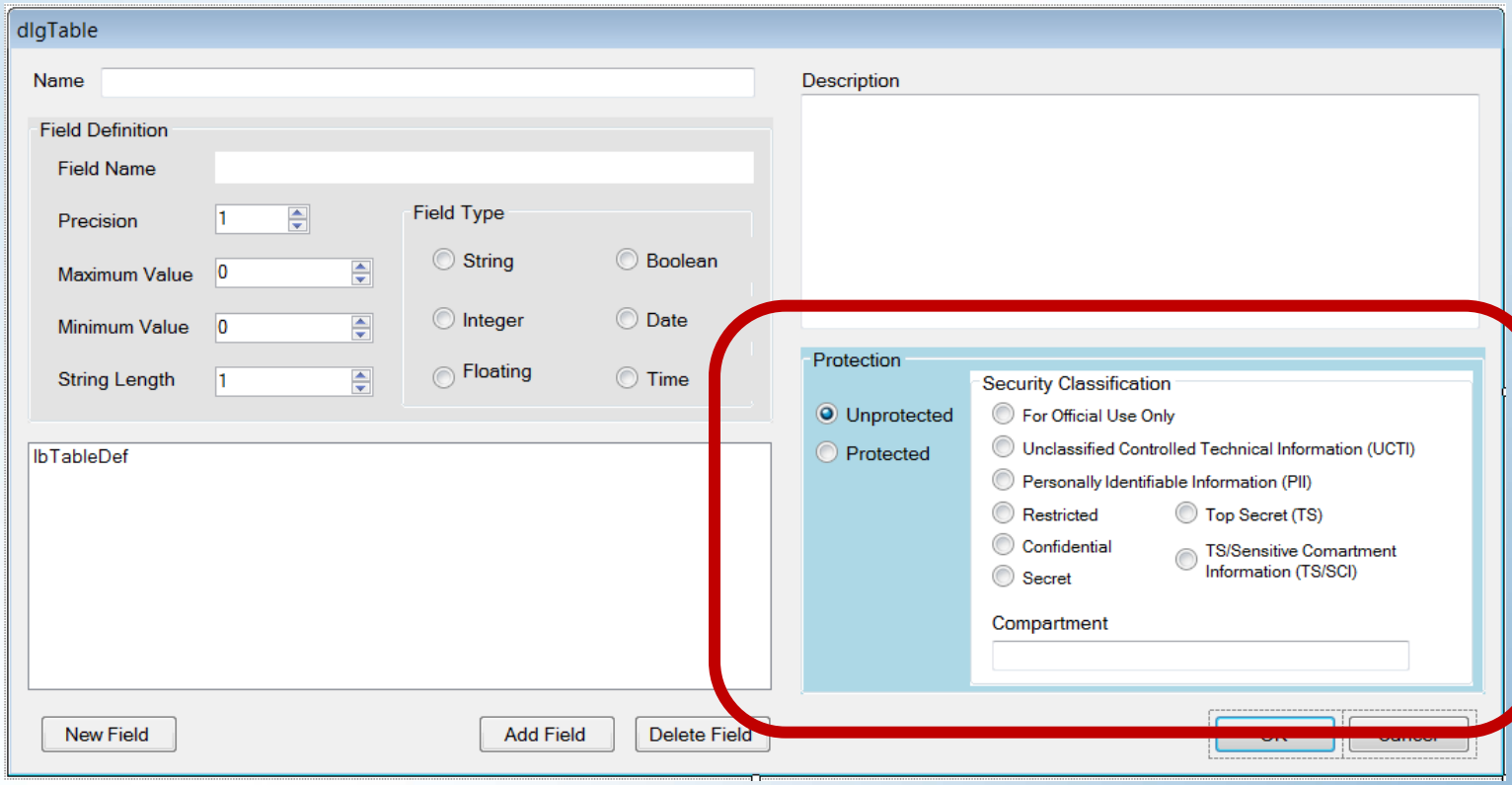
SOFTWARE REQUIREMENTS ANALYSIS

1. Capture & Model Data-processing Transactions (Behaviors)
   - Behavioral Models of Software Interactions with External Sources
   - Data Flows, Control Flows, Function Flows (21 Variations)
   - Data Specification
     - » (12 standard variations/User Defined Data Types/Classes)
     - ✓ Including Security Classifications (PII, Classified, Secret, Top Secret, TS/SCI…)
   - Performance (Execution Timing)
   - Resource Utilization
   - External Interactions (User, DBMS, Computing Environment, External Systems)
   - Derived Security Procedures/Features (Data Bases, Data Files & Computing Resources)
2. Develop Test Cases
   - Verify Data-processing Transaction Models
3. Generate Requirement Specifications
   - Software Requirement Specification
   - Interface Requirement Specification(s)
   - Test Procedures
   - Preliminary Data Dictionary



Design Security In

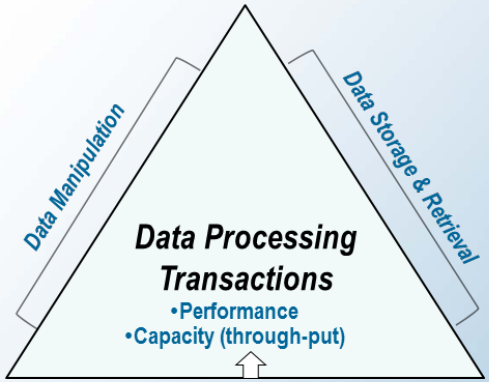
- Enable security-by-design for software assets:
  - Database Records, Files & Computing Resources
- Specify derived Software Asset Security features



Software Engineering Practices Overview

FUNCTIONAL ANALYSIS & ALLOCATION

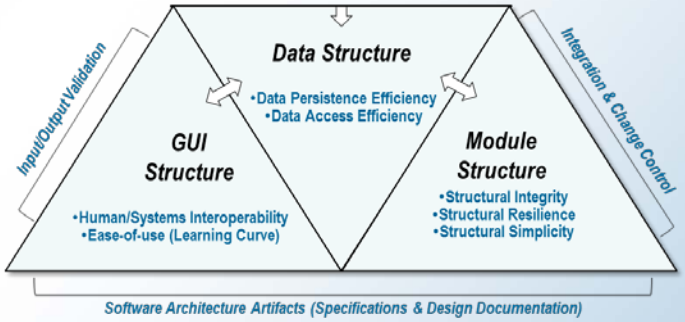
4. Decompose Complex Behaviors (Functions)
   - More detailed behavior models reflecting internal software data processing procedures
5. Combine and Assimilate Common Functions
   - Combined & sanitize specification(s) to resolve conflicts, overlaps and inconsistencies
   - Reduce Complexity
   - Enforce Modularity
6. Generate Software Functional Specifications (Preliminary Design Document)



Software Engineering Practices Overview

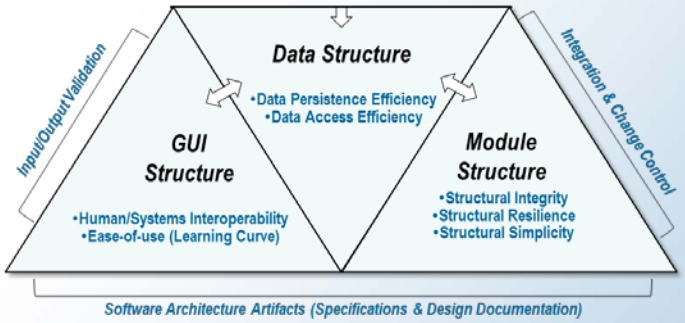
SOFTWARE DESIGN SYNTHESIS

7. Identify Structural Components - 3 Types:
   1) Forms & Dialogs (GUI Structure)
   2) Classes & Collections (Data Structure)
   3) Modules (Module Structure)
8. Derive Design Details necessary to characterize each component
9. Establish the Software Integration Strategy
   - Integration Tiers
   - Derived Integration Components/Features
     - » (not identifiable during Requirements & Functional Analysis)
10. Generate Detailed Design Documentation
    - Component “design-to” specifications
    - Integration Plan
    - Software Bill-of-Materiel (S-BOM)
    - Detailed Data Dictionary



Software Design Synthesis Framework

- For each Data-Processing Scenario & Associated Behaviors:
  1. Identify the Structural Components needed to enable the behaviors
     - » Allocate one or more Behaviors to each Component
  2. Identify & Define the design elements of each Component (based on type)
     - » Identify the Routine Interface(s)
     - » Design Elements perform sub-functions (Software Units) which enable the desired Behavior(s)



GUI Structure

Form/Dialog
- GUI Controls
- Properties
- Routines!
- EventHandlers*

Data Structure

Class/Collection
- Properties
- Routines!

Module Structure

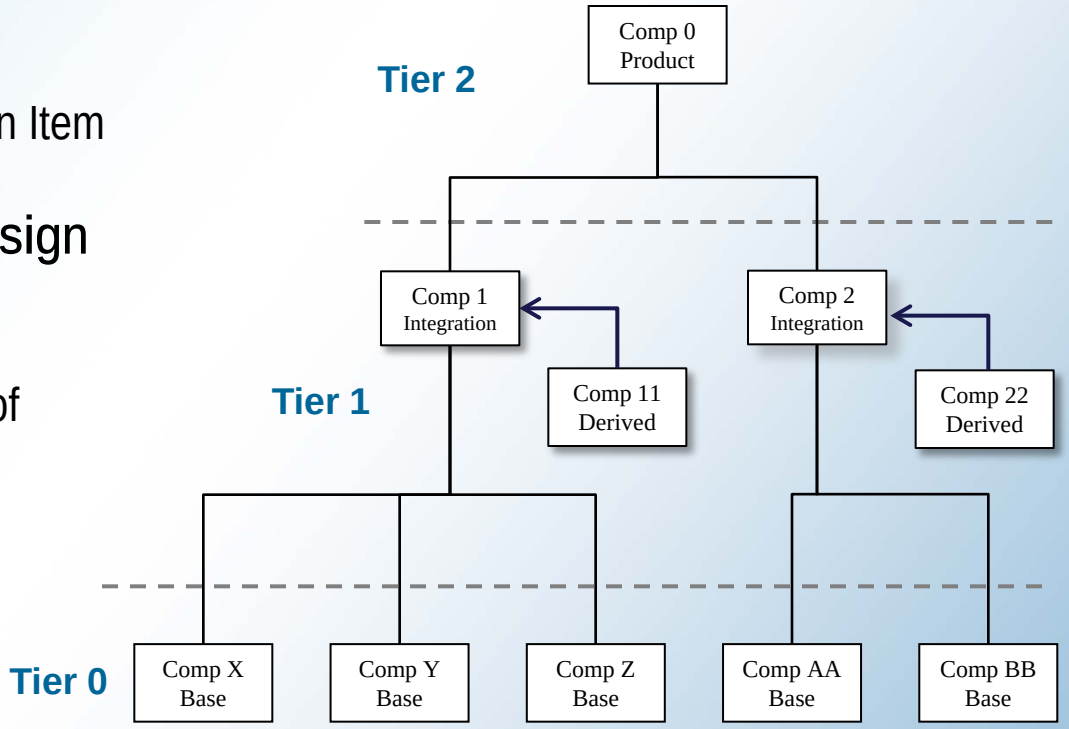
Module
- Properties
- Routines!
- EventHandlers*

3. Establish the Software Integration Strategy
   - » Create the Integration Components/Subcomponents needed to facilitate the integration of lower-tier components

! Specification of Software Interface (DataExchange) ** Special type of routine associated with a GUI Controls & User-defined Events*

Software Integration Hierarchy

- Top-level Component of type "Product"
  - Typically a Software Configuration Item
    - » (CSCI - Uppermost tier)
- Components identified from Design Synthesis
  - Type "Base"
  - Represent the Lowest-level Tier of Structure (tier 0)
  - Resulting from Design Synthesis
- Integration Tiers provide workplace to identify "Integration" Components
  - Assemble and integrate multiple lower-tier components
  - Identify newly derived "features" needed to support integration
  - Enforce the Integrated Data-processing Behaviors



Summary

- Software Engineering emphasizes “Design” before “Coding”
  - Specifies a complete Architectural Representation (SBOM)
  - Minimizes software life-cycle costs
    - » Reduce rework during initial development
    - » Establishes a tangible architecture which enables change control & enhancements
  - Enhances the effectiveness of other methodologies (e.g., AGILE)
  - Prototyping to reduce *RISKS* and explore design solutions
- Application of “Systems Engineering” practices *ARE ESSENTIAL* to Software Development/Acquisition Success
  - Central to achieving the “vision” of automated software generation from design
- Methodology has been automated (Prototype)
  - Desire a partnership to demonstrate Methodology & Tool effectiveness

Demos available upon request after session

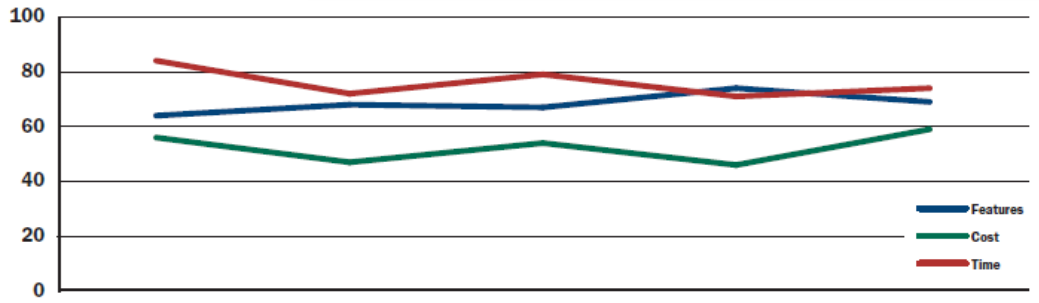
BACKUPS

Current State of the Industry

OVERRUNS AND FEATURES

CHAOS MANIFESTO 2013
The Standish Group

Time and cost overruns, plus percentage of features delivered from CHAOS research for the years 2004 to 2012.



	2004	2006	2008	2010	2012
TIME	84%	72%	79%	71%	74%
COST	56%	47%	54%	46%	59%
FEATURES	64%	68%	67%	74%	69%

CHAOS Reports Summary

Agile Manifesto 2001

	1994	1996	1998	2000	2002	2004	2006	2008
Successful	16%	27%	26%	28%	34%	29%	35%	32%
Challenged	53%	33%	46%	49%	51%	53%	46%	44%
Failed	31%	40%	28%	23%	15%	18%	19%	24%