Applying Standard Independent Verification and Validation (IV&V) Techniques within an Agile Framework: Is there a Compatibility Issue?

James B. Dabney, UHCL
James D Arthur, VA Tech
IEEE/INCOSE SysCon 2017
Overview

• Conventional mission-critical software lifecycle
• Conventional IV&V process
• Agile software development
• Hybrid Agile variants
• Adjusting IV&V to hybrid Agile
• Conclusions
Conventional Mission-Critical Software Lifecycle

- Traditional lifecycle based on waterfall model
- Sequence of milestone reviews
  - Preliminary design review (PDR)
  - Critical design review (CDR)
  - Test readiness review (TRR)
  - Design certification review (DCR)
- Larger projects incremental model
  - Planned series of waterfall lifecycles
- Certification mandated by regulations (e.g. FDA, UL)
Example Traditional Waterfall Lifecycle

- Requirements
  - PDR
- Design
  - CDR
- Implement
  - TRR
- Test
  - DCR
- Operations
Example Incremental Lifecycle

- Increments can be developmental or operational
- Plan several increments ahead
Conventional IV&V Process

• Reduce program risk by analyzing key artifacts
• Strive to find issues in-phase by mirroring development
• Verify during each lifecycle phase that the product satisfies requirements defined in previous phase
  – Requirements meet user needs, complete
  – No unintended functionality specified
  – Design satisfies requirements and no more
  – Testing fully covers design and requirements
Understanding Agile

Need to respond to constant changes

Agile Values
The fundamental reason for a “new” paradigm
Defines the set of most important beliefs of what is truly important

Agile Principles
Defines a set ways to meet the values

Agile Practices
Defines in detail how this is implemented in practice

Material adapted from "All about Agile", Ahmed Sidky, Presentation for CS 5704, Va Tech Fall 2006
<table>
<thead>
<tr>
<th>Individuals and interactions</th>
<th>Over</th>
<th>Process and tools</th>
<th>Mission-Critical / IV&amp;V Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working software</td>
<td>Over</td>
<td>Comprehensive documentation</td>
<td>Docs required for IV&amp;V, certification</td>
</tr>
<tr>
<td>Customer collaboration</td>
<td>Over</td>
<td>Contract negotiation</td>
<td>End product requirements defined at outset</td>
</tr>
<tr>
<td>Responding to change</td>
<td>Over</td>
<td>Following a plan</td>
<td>Change inevitable, must be managed</td>
</tr>
<tr>
<td>Agile Principles</td>
<td>Mission-Critical / IV&amp;V Consideration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Customer satisfaction</strong> by rapid, continuous delivery of useful software</td>
<td>Often don’t need working software until late in program</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Working software</strong> is delivered frequently (weeks rather than months)</td>
<td>Frequent updates less important than technical rigor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Working software is the principal measure of progress</td>
<td>Safety / health of enterprise principal measure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Even <strong>late changes in requirements</strong> are welcomed</td>
<td>Late changes inevitable but can be costly</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Close, daily cooperation</strong> between customer &amp; developer</td>
<td>Often integrated product teams</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fact-to-face</strong> conversations is the best form of communication</td>
<td>Clear documentation essential due to long operational life</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Projects are built around <strong>motivated individuals</strong>, who should be trusted</td>
<td>Some projects span careers, must be able to retain institutional knowledge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Continuous attention to <strong>technical excellence</strong> and good design</td>
<td>Clearly essential</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simplicity</td>
<td>Always desirable</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Self-organizing</strong> teams</td>
<td>Multi-site, multi-contractor, large staff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regular <strong>adaptation</strong> to changing circumstances</td>
<td>Budgeting, staffing can have multi-year lead times</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Agile Planning: The Scrum Process
Agile Planning:
Release and Iteration Planning

<table>
<thead>
<tr>
<th>Product Backlog</th>
<th>Release A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature 1</td>
<td>Feature 1, Feature 2, Feature 3a</td>
</tr>
<tr>
<td>Feature 2</td>
<td></td>
</tr>
<tr>
<td>Feature 3</td>
<td></td>
</tr>
<tr>
<td>Feature 4</td>
<td></td>
</tr>
<tr>
<td>Feature 5</td>
<td></td>
</tr>
<tr>
<td>Feature 6</td>
<td></td>
</tr>
<tr>
<td>Feature 7</td>
<td></td>
</tr>
<tr>
<td>Feature ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Release Backlog</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Story A</td>
<td>Story A</td>
<td>Story C</td>
<td>Story F</td>
</tr>
<tr>
<td>Story B</td>
<td>Story B</td>
<td>Story D</td>
<td>Story G</td>
</tr>
<tr>
<td>Story C</td>
<td></td>
<td>Story E</td>
<td></td>
</tr>
<tr>
<td>Story D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Story ...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Material adapted from "All about Agile", Ahmed Sidky, Presentation for CS 5704, Va Tech Fall 2006
Adapting Agile to Large Projects

• Alistair Cockburn (one of the original agile proponents): “small projects, web projects, exploratory projects, agile is fabulous; it beats the pants off of everything else, but for NASA, no” [AM13]

• “Embedded systems have specific product requirements, e.g. safety, which are not obviously addressed by agile practices such as XP or Scrum” [EOS14]

• Key assumptions of Agile (e.g. co-located teams) are difficult to realize on large projects [TFR02]
Variants of Agile for Large Projects

• Scaled Agile Framework (SAFe) Intended for high-assurance environments (medical)
  – Designed to comply with regulatory requirements (FDA)
  – Gaining acceptance

• Incremental Commitment Model (ICM)
  Merges concepts of classic V-verification, concurrent engineering, Agile
  – Intended for large mission-critical and net-centric systems
Hybrid Projects

• Similar to SAFe methodology

  - Early lifecycle activities follow standard process
  - Requirements, design, test follow Agile process
    – Sequence of releases composed of multiple sprints
    – Work down project backlog
  - Certification follows standard process
Water-Scrum-Fall Process
Mapping Traditional V&V to Agile

• Assessed applicability of standard V&V methods to hybrid Agile

• For each method specified for project elements, assessed
  – Inputs
  – Timing in lifecycle
  – Feasibility of executing method given the timing and available information

• Methods fall into three classes
  – Early lifecycle methods generally compatible
  – Methods involving tracing need to be tailored
  – Methods involving completeness need to be replaced
<table>
<thead>
<tr>
<th>Methods Requiring No Tailoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verify Implementation of Requirements or Design in Source Code or Scripts through Manual Inspection</td>
</tr>
<tr>
<td>Reuse applicability by comparing operational environments</td>
</tr>
<tr>
<td>Validate Safety Requirements by Inspection of Traces to Fault Trees and FMEA</td>
</tr>
<tr>
<td>Verify Software Behavior for Off-Nominal Conditions using Independent Testing</td>
</tr>
<tr>
<td>Validate Software Architecture by Inspecting Traces to Essential Properties</td>
</tr>
<tr>
<td>Verify Critical Software Changes By Inspecting Change Requests</td>
</tr>
<tr>
<td>Verify System/Software Architecture Using a Discrete Model of Performance Requirements in Stressing Scenarios</td>
</tr>
<tr>
<td>Assess Architecture Completeness by Inspection Against an Architectural Standard</td>
</tr>
<tr>
<td>Validate Feasibility Study Conclusions by Inspection</td>
</tr>
<tr>
<td>Validate Test Procedure by Inspection and Traces to Requirements</td>
</tr>
<tr>
<td>Validate Mission Project Operational Concepts by Generating Use Cases from Concept Documentation</td>
</tr>
<tr>
<td>Validate System Security Categorization and Regulatory Security Requirements by Inspection using Security Risk Management Framework (NIST-SP-800-37, Step 1)</td>
</tr>
<tr>
<td>Verify Security Control Selection and Threats/Risks Identification by Inspection using Security Risk Management Framework (NIST-SP-800-37, Step 2)</td>
</tr>
<tr>
<td>Methods Requiring Tailoring</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>Verify Software Code Quality using Static Analysis Tools</td>
</tr>
<tr>
<td>Validate Test Plan by Inspection</td>
</tr>
<tr>
<td>Validate Requirements by Inspecting Bidirectional Traces</td>
</tr>
<tr>
<td>Verify Test Execution by Inspection of Test Cases, Inputs and Results</td>
</tr>
<tr>
<td>Verify SW Interface Implementation by Inspection Against Interface Design</td>
</tr>
<tr>
<td>Verify Critical Software Changes By Inspecting Change Requests</td>
</tr>
<tr>
<td>Validate Test Cases by Inspection and Traces to Requirements</td>
</tr>
<tr>
<td>Verify Scripted Timeline Via Manual Multi-Directional Tracing</td>
</tr>
<tr>
<td>Verify Software Design by Inspecting Traces to Requirements and Software Architecture</td>
</tr>
<tr>
<td>Verify Software Capabilities through Independent Testing of Operational Scenarios</td>
</tr>
<tr>
<td>Methods Incompatible with Agile</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>Validate Interface Requirements by Inspection Against Component Interfaces</td>
</tr>
<tr>
<td>Validate Requirements by Inspecting Against Quality Criteria and System/Software Background Artifacts</td>
</tr>
<tr>
<td>Validate Test Design by Inspecting Traces from Scenarios</td>
</tr>
<tr>
<td>Verify Software Implementation by Inspecting Traces to Requirements</td>
</tr>
<tr>
<td>Verify Software Interface Design by Inspection Against Interface Requirements</td>
</tr>
<tr>
<td>Verify and Validate Requirement Implementation using Flow Diagrams to Uncover Missing, Conflicting, or Unnecessary Behavior</td>
</tr>
</tbody>
</table>
Interface Requirements Objectives

• Correlate integration requirements to specific interfaces and examine coverage to ensure that all interfaces are specified and all interface requirements relate to a necessary interface.

• Correlate integration requirements to ensure they are required, incorrect behavior is prevented, unexpected inputs responded to appropriately.

• Verify interface requirements are correct, consistent, complete, accurate, verifiable, where
Interface Requirements
Hybrid Agile Variant

• Capture the interface requirements as they emerge during each release

• As interfaces are defined to clear blocks, developer artifacts can be used to build a picture of the interface and refine requirements

• Interface map using tool
  – Starts with interface template or estimate
  – Incrementally capture interfaces and track properties
  – Track completeness and measures of risk burndown
Requirement Validation Objectives

• Ensure system requirements satisfy acquirer needs relative to system software
• Ensure software requirements meet system needs from functional and non-functional perspectives
• Ensure requirements for software interfaces are adequate in terms of operational environment, dependability, fault tolerance, and functional and non-functional perspectives
• Analysis steps address
  – Unambiguous
  – Verifiable
  – Consistent
  – Correct
  – Complete
Requirement Validation Agile Variant

- Unambiguous, verifiable, consistent compatible
- Correct, complete are challenge
- Potential solutions
  - Predictive model of requirements
  - Risk burndown model
Test Design Objectives

• Ensure test designs correctly specify a feature or combination of features
• Ensure the test environment is sufficiently complete, correct, and accurate
• Analysis steps
  – Develop a set of scenarios considering correctness and adverse conditions
  – Validate the scenarios with walk-through
  – Trace requirements to scenarios.
  – Trace scenarios to software structure
  – Trace the scenarios to test design and environment
Test Design Agile Variant

• Identify relevant scenarios
• Map requirements to scenarios as requirements emerge
• Potential solutions similar to Method 2
  – Predictive model of test design
  – Risk burndown model
Software Implementation Objectives

• Ensure that software components can reliably perform required capabilities under nominal and off-nominal conditions, perform no undesired behaviors, and that documentation is adequate to support maintenance.

• Ensure that the code satisfies dependability and fault tolerance requirements, is capable of detecting identified hazards, and introduces no hazards.

• Ensure that all applicable requirements are implemented (for example, from SRS and IRS) and no unspecified behavior is introduced.

• To accomplish these objectives
  – Determine required nominal conditions from operations documentation and technical reference
  – Locate the source code relevant to the required functionality
  – Analyze implementation for completeness, correctness, behavior under unexpected conditions
  – Trace implementation to nominal and off-nominal scenarios
  – Analyze code in terms of fault tolerance and hazard response
Software Implementation Agile Variant

• Partial assessment at end of each release
• An analytical framework to track
  – Implemented functionality
  – Expected functionality
  – Recognize unexpected functionality
• Example techniques
  – Quality Function Deployment
  – Safety cases
  – IV&V reference models
Interface Design Objectives

• Ensure all relevant requirements represented in design documentation

• Find evidence that all relevant assurance goals are achieved for all interfaces with hardware, operators, other software functions, and other systems.

• To accomplish these objectives
  – Compare requirements and design documentation including analysis of algorithms, commanding, state/mode definitions, exception handling, error logging, configuration data, performance criteria (e.g. timing, latency, bandwidth), interface specifications (all layers)
  – Verify that interface requirements are unambiguous, complete, accurate, consistent, testable/verifiable, traceable
  – Verify interface flows are correct and consistent (sequences, flows, control, formats, standards)
Interface Design Agile Variant

• IV&V interface model
  – Captures information from each sprint
  – Builds an understanding of the as-built interfaces
  – Check for common interface errors

• An interface IV&V approach which deals with both requirements (Method 1) and design (Method 41) may be the best approach.
System Software Safety Objectives

• Known software-based hazards are controlled
• Dependability and fault tolerance requirements are satisfied via lower level requirements, software design, and implementation and that testing is in place to verify the fault tolerance behavior is not compromised by modifications
• All required functionality is implemented correctly and no unnecessary behavior is implemented
• To accomplish the objectives
  – Define a set of top-level claims related to critical events such as collision avoidance during docking, parachute operations, deorbit
  – Using hazard reports and system requirements documentation, establish the first levels of supporting claims
  – Continue developing supporting claims as the project proceeds, tracing in turn to requirements, design, code, and test
  – Capture and document evidence (requirements, design, code, test) at the lowest level of the safety case
System Software Safety Agile Variant

• Depth-first approach required
• Develop a complete safety case using postulated claims and evidence
• Update the case incrementally
  – Establish the top level claims using concept documentation, the system architecture, and high-level requirements
  – Develop the safety case as deep as possible using the early lifecycle artifacts
  – Postulate successively lower level claims across the breadth of the safety case, down to the evidence level
  – At each release (or more often if possible), revise the safety case to reflect the functionality implemented
Requirement Implementation
Objectives

• Ensure requirements represented in design
• Design does not introduce capability that is not required
• Ensure all elements of the design are in code components
• Code does not introduce capability that is not required
• Ensure all requirements trace to code
• Ensure code components can reliably perform
  – Nominal conditions
  – Off-nominal conditions
• Documentation (both embedded and stand-alone) adequate for code maintenance.
Requirement Implementation
Agile Variant

• Hierarchical requirements trace tool
• System model
  – UML/SysML
  – Flow diagrams
• Risk burndown tool
Conclusions & Future Work

• Pure Agile not appropriate for mission-critical or safety-critical projects
• Hybrid Agile gaining acceptance
• Adapt IV&V methodology to hybrid Agile
  – Maintain technical rigor
  – Accommodate project flows