



Barry McManus



Richie Siconolfi

# AN INTRODUCTION TO AGILE METHODOLOGIES

## CHALLENGES AGILE PRACTICES MAY POSE TO THE AUDITOR

---

Continuing in our agile software development methodology series, we look at what agile practices may pose to the auditor. Agile<sup>1</sup> is a group of software development processes with a reduced onus on prescribed processes and increased focus on good software engineering practices<sup>2</sup>. Given agile increases focus on people rather than processes<sup>2</sup>, what are the challenges to the computer systems (CS) auditor? This article will briefly look at some of the common activities of agile observed during audits and briefly examine what a CS auditor might find.

## INTRODUCTION


Annex 11 4.1<sup>3</sup> states that validation documentation should cover the relevant steps of the lifecycle – for example, requirement, design, appropriate testing and so forth. As a result, there is an expectation (hope) that a CS supplier generates this documentation. Given the problems associated with creating and maintaining documentation, the agile manifesto favours working software versus comprehensive documentation; people over process<sup>4</sup>. Since documentation is associated with prescribed process, what does agile provide instead of documentation? How can they ensure quality and facilitate maintainability? What are the gaps (if any) that may pose a problem from a regulatory perspective? Given that the implementation of the agile approach is often tool driven rather than ‘documentation’ driven, how can the auditor ascertain evidence of traceability, accountability and quality?

## PLANNING AND ANALYSIS

The initial phase of any CS approach is the capture and documenting of the user needs. Traditionally this is documented in the User Requirements Specification (URS). But requirements are a point of software failure<sup>2</sup>; one of the primary causes for introducing false assumptions into software, (which are realised as defects).

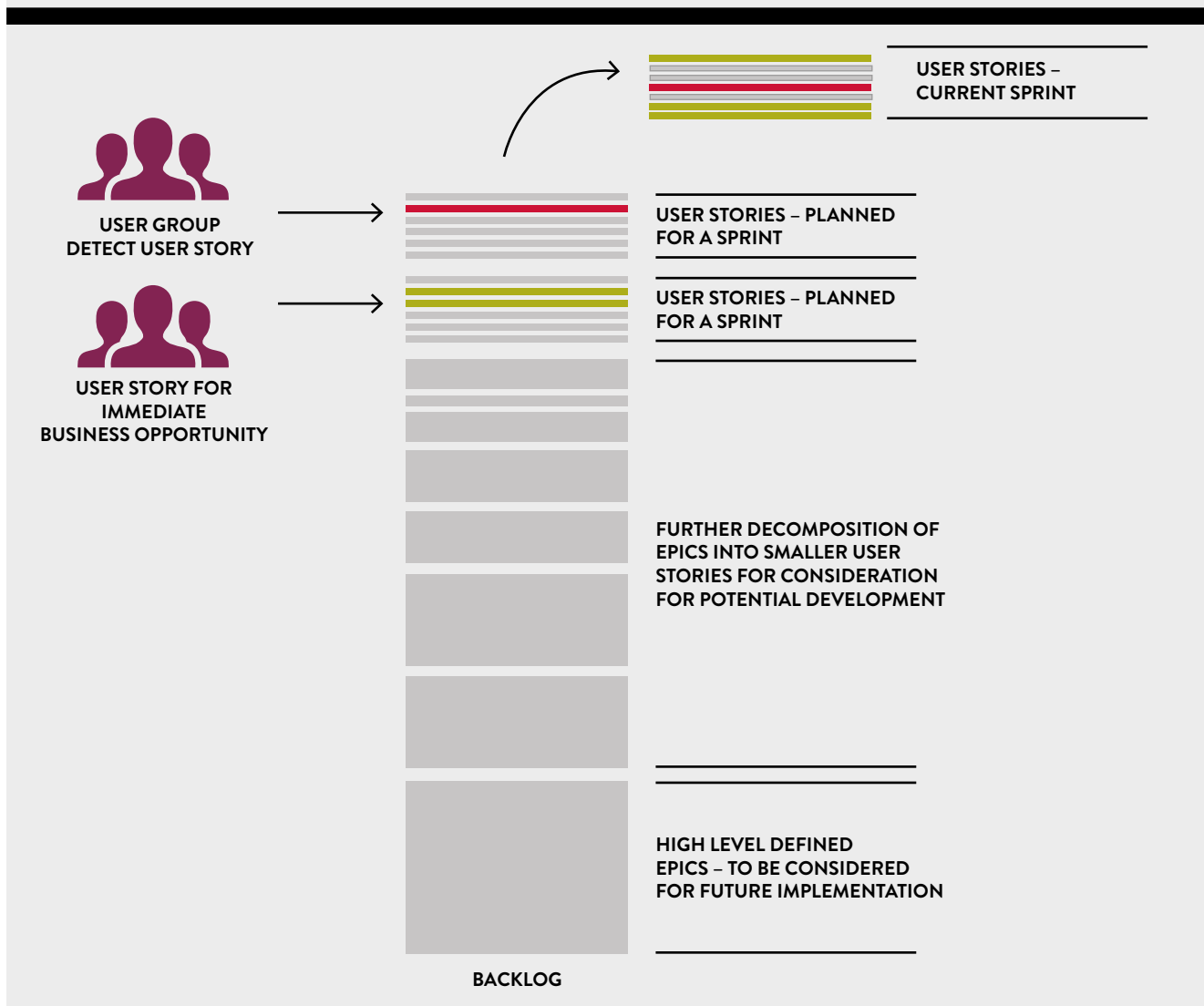
Agile manages user needs as a list of items, (for example as Epics/User Stories<sup>5</sup>). Often this list is called a ‘Product Backlog’<sup>5</sup> (see Figure 1). The level of detail on a Backlog will vary. For example, Epics are high level statements that need to be decomposed into smaller User Stories. User Stories are used to capture the work that must be done to deliver a working software.

Backlog Grooming<sup>5</sup> is the process of analysing, prioritising, decomposing Epics into User Stories for iteration/sprint planning. This approach is performed by co-located, cross functional teams (developers, testers, QA, business sponsors, etc.), to ensure that the iteration has the best chance of delivering working software.

 The inclusion of a tester perspective at the review of user needs helps identify a wider range of potential issues, (e.g. User Story incompatibility, ambiguity, testability, end-to-end process and so on). A developer will focus on the solution within the confines of code modules. The tester will consider the end to end considerations (e.g. data transfer to remote systems). This is a ‘quality’ Front Loading activity. Quality Front Loading is the practice of reallocating time and effort from the end of development activities to the start of development activities to reduce issues from being built into the system. This practice saves time and effort on detecting and correcting defects in the later stages.

User Stories are selected for an iteration based on their priority and size (that is, whether working software can be delivered (into independent test) by the end of the iteration).

FIGURE 1. PRODUCT BACKLOG



■ User needs (e.g. Epics and User Stories) are managed by workflow-based software tools. Such tools greatly enhance the visibility of actions, facilitate change, impact analysis, history and provide instant monitoring for quality feedback of development artefacts, (e.g. Atlassian JIRA and IBM DOORS). The use of (qualified) tools provide accountability and traceability of each work item at a granular level that traditional documentation simply can't provide. This is a definitive quality plus.

User Stories are popular mechanisms for capturing user needs and deserve merit as their use of objective language can help reduce the problems of poorly defined requirements. The following basic User Story is provided as an example:

- As an ATM user, I want to view my balance, so that I can withdraw £50.00, but only if my balance is greater than or equal to £50.00.

Compared to a typical User Requirement:

- The ATM will allow a user to withdraw money.

User Story creation also involves the writing of acceptance tests in parallel. Writing a test case is a design activity that can quickly identify assumptions and otherwise unforeseen user needs before a developer would design and code the User Story into the software.

■ The activity of writing tests in parallel to User Story analysis is a common quality Front Loading practice and an example where the act of test design can prevent defects (via identification of ambiguities, conflicts, interface issues and so forth). Furthermore, robust Front Loading removes additional false assumptions being inferred from erroneous user needs and thus removes more defects from being built into the software. Check to ensure the qualification of tools.

Note that SOPs will have limited information, with the process description residing in a tool that will be easily amended and changed (e.g. WIKIs). Ensure to check the content and management of these tools.

## PLANNING

The objective of an iteration is to deliver 'some' working software. 'Points'<sup>5</sup> are assigned to User Stories to help size, (estimate), a User Story. The larger the point, the greater the effort of a User Story. An iteration's time is fixed (e.g. two weeks duration) and the total number of points it can deliver is fixed. Functionality (User Stories) delivered is flexible: User Stories are prioritised within sprint planning with the objective of sacrificing low priority stories should

time become constrained. An example of a prioritisation approach is MoSCoW<sup>6</sup>: Must have (during this iteration); should have (during this iteration); could have (during this iteration) and won't have (during this iteration). User Stories are selected on a 60:20:20 split (must, should, could), with could have sacrificed before should have.

At the end of the iteration, dropped user stories are considered for inclusion in the next iteration, returned to the backlog or dropped completely. Some user stories may not be implemented due to constraints or changing business needs.

■ **CAUTION** How this activity is traced and justified, (i.e. why a User Story is dropped and whether it is rescheduled for a subsequent iteration to meet the intended needs of the user), is often a weakness amongst agile suppliers and something to look out for. Check to ensure that the intended needs of the user are being delivered.

A Retrospective is the practice of reviewing the activities of the finished iteration and assessing what went well and what could be improved for the next iteration.

■ The practice of regularly reviewing the implementation of user needs from the backlog and ensuring multiple roles (including test and QA) are involved in the review, prioritising, rapid implementation and early acceptance test writing is welcomed. Although this activity is rarely formally documented, it may be detailed within the history (audit) trail of the tool that is used to manage the work items or within the process repository (e.g. a WIKI).

## RISK MANAGEMENT

Agile organisations (like traditional software organisations) typically do not spend time on formal risk analysis and management. However, they spend time informally risk planning at the daily stand up meeting. The collective use of many 'voices' during this event can provide a good risk discussion around the risk of the User Story not being progressed within that iteration.

■ The inclusion of multiple perspectives (e.g. developer, tester) to discuss implementation risks of User Stories is good practice.

Risks may not be captured or analysed in detail with regards to regulatory, operational or wider technical impact: they may not be formally quantified, impacts specified, or mitigations identified (e.g. designing data redundancy). Risk may not be used to

justify the decisions for performing or not performing activities within the lifecycle. Furthermore, test planning may be relegated to just verifying User Story functionality rather than verifying potential mitigation strategies, (e.g. performance testing).

■ Although risks are discussed, they often focus on the 'here and now' for the daily tasks (coding). Any overarching risks of the system (e.g. performance constraints) may be forgotten over time if they are not documented. As a result, risk analysis and ongoing risk management may be a source of weakness for suppliers.

## DESIGN

Design is a fundamental activity required for ensuring reliability, robustness, maintainability and so forth. Limited focus on this activity will reduce the benefits of 'designing quality in' and supporting regulatory needs, (e.g. data integrity). A lack of design consideration is a hot topic within agile.

'YAGNI'<sup>2</sup> ('You Aren't Going to Need It'), means why spend valuable time on establishing a large design paradigm when the user needs or business needs will likely change. YAGNI favours keeping design down to the absolute minimum to cover just what is being developed in the immediate iteration<sup>6</sup>, which makes sense from a business point of view to gain a competitive edge.

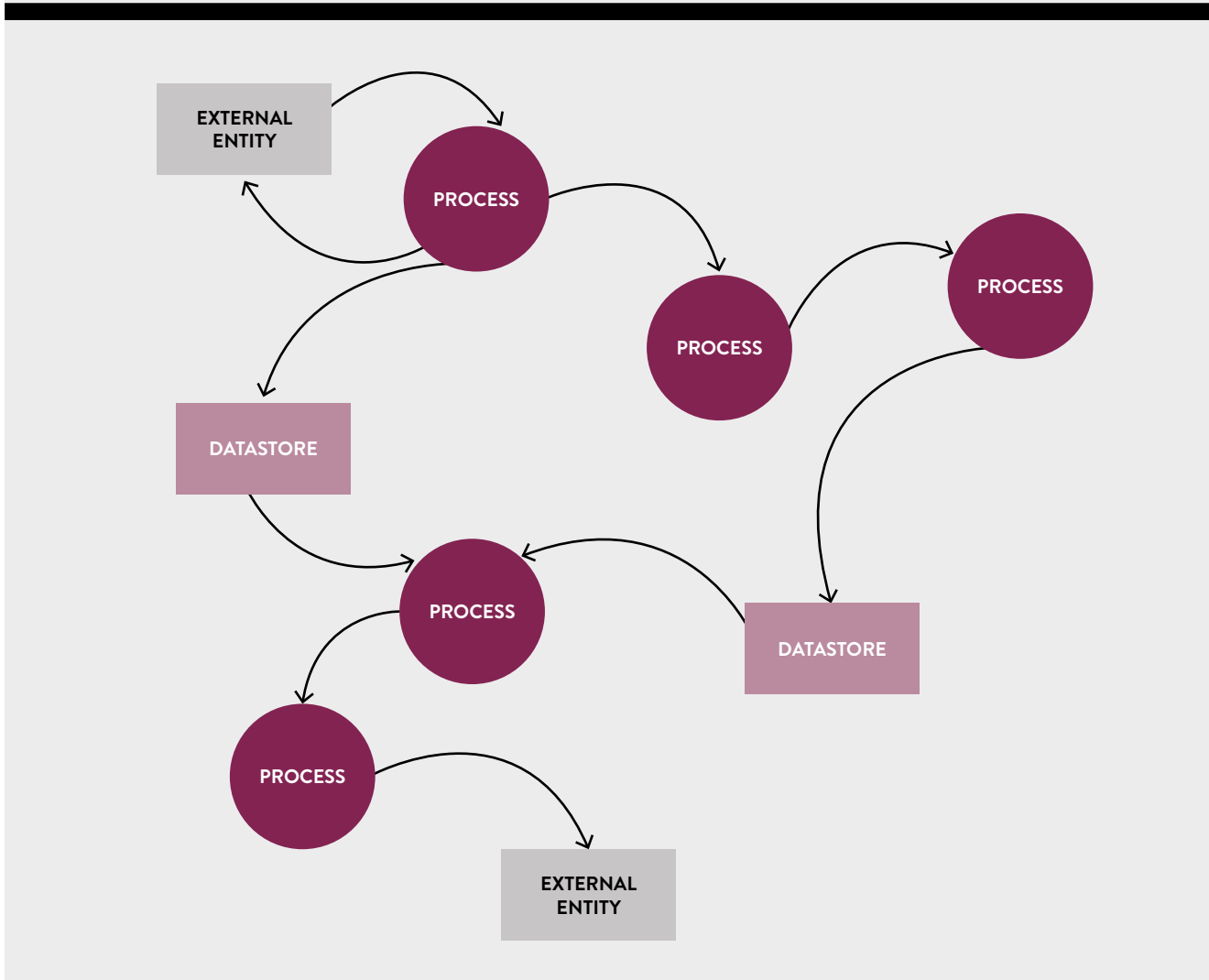
■ Tools such as WIKIs are commonly used to 'document' the key design considerations for an implementation. Ensure to check for the audit history of the WIKI tool.

Some organisations will leverage more 'traditional' computer aided design tools (e.g. RATIONAL ROSE) that will trace user needs into a design of the physical and logical arrangements (e.g. data flow diagrams) and on into code and tests.

However, 'YAGNI' limits the consideration of future risks<sup>6</sup>. For business systems that need to accommodate growth, this may be a source of costly difficulties, (e.g. performance, security, data integrity issues, integration issues with other systems).

Technical debt<sup>7</sup> is a consequence of limited design within code when developers choose to do just enough to code a quick implementation. The debt is the future work that needs to be done to make the code robust and efficient. If the debt is not repaid, it will accumulate interest that make it harder to amend in the future, (e.g. code duplication). This will hinder the speed of new functionality implementation as accumulated issues need to be fixed.

FIGURE 2. GENERIC DATA FLOW DIAGRAM



For databases that were not designed pre-implementation, database schema diagrams produced by a database tool will not demonstrate the intended design needs of the system. Ensure to check for data definitions, data flows, reviews, rework and so forth to ascertain a versioned and controlled design approach. A lack of design will manifest itself e.g. when a system faces scalability issues upon increased usage.

Some agile proponents may state that the code is the most up to date 'document' of the system and that the code contains the design. It is time consuming to demonstrate the intended design needs of the system within the code and difficult to verify that a growing code base reflects intended design.

## IMPLEMENTATION/TEST

Agile rightly places strong emphasis on testing at the code level, (Unit Testing<sup>8</sup>), which is typically automated. This is a test activity that is often negated in other development approaches.

Test Driven Development<sup>2</sup> is the practice where a test is written before the code and then executed to fail as a control. Next the code is written to pass the test. This activity helps reduce the scope of 'gold-plating', (where developers add what they 'think' the system should do in addition to the User Story), which can lead to needless issues (additional effort, integration issues, rework). The results of automated unit tests may be documented within continuous integration tools such as Jenkins.

The use of nightly builds (where the code is automatically compiled, deployed on a test server and automated test scripts are executed), can provide early feedback on the 'health' of the software code after the previous day's development activities, via dashboards (e.g. Jenkins) and email alerts.

'Never Break the Build'<sup>2</sup> is a self-regulating practice among the development team where a broken build (non compiling code within the code repository) will hold up the entire team. This practice enhances quality during development activities.

Some software houses will also incorporate static analysis tools to check coding rules, memory leaks, dead code (code that will never be executed), performance constraints and so on.

Be wary of claims of unit test coverage. Inspection of code modules may reveal a limited coverage of automated unit tests and a reliance on 'manual' unit tests which are not likely to be documented.

## INDEPENDENT TESTS

Independent testing also occurs at the iteration level. The tester will execute the User Story test. As with unit testing, the goal is to automate testing as much as possible.

‘Effective’ automation provides many benefits: increased test coverage, consistent testing across releases, rapid feedback to developers, reduced human error, facilitate non-functional testing during coding (e.g. performance tests) and facilitate rapid regression testing.

Automated testing will only be beneficial if the code itself is stable, (otherwise time is wasted on recoding automated tests to reflect re-engineered code) and only if the tests themselves are effective, (e.g. poor coverage, duplication or redundancy). Agile houses may state that automated tests are reviewed as part of code reviews and evidence of this activity may be captured within the audit trail of the version control tool (e.g. GIT) or a code review tool (e.g. Crucible).

The use of tool audit trails can quickly establish if good practices are being followed. However, the adequacy of the review practices may require further investigation.

Exploratory testing<sup>8</sup> is a technique where the time is ‘solely’ spent on exploring and assessing the software for defects, (there is little or no formal test case writing). This is a valued technique if it is used in addition to other testing activities. Be wary of organisations that solely rely on exploratory testing as it will be extremely difficult to perform a direct replication of an exploratory test and provide assurance that the system works as intended and doesn’t work as unintended.

Ensure that exploratory testing is used in support of other types of testing as part of an over-arching strategy. As a minimum, ensure that exploratory tests are documented in the event a defect is found to facilitate the correct retest and to support future regression scope.

The focus of various testing activities (e.g. fault tolerance, load testing) is beyond the scope of this discussion but if there is limited scope on design and test planning, then testing may be focused on a narrow area, (where the value added of that testing will eventually be reduced). For example, a focus on the ‘here and now’ may over emphasise usability to the detriment of scalability.

Be wary where organisations only leverage unit and acceptance testing. Any issue found will only relate to the area of focus, whereas more complex issues relating to, for example, configuration, security or performance maybe overlooked.

Automated testing is not a silver bullet and can involve a large engineering overhead. Passing automated tests does not necessarily equate to robust code. Establish whether ‘testers’ are involved in reviewing the test case coverage within automated tests. What information is available to demonstrate the effectiveness, adequacy and coverage of automated tests?

## TOOLS

Tool integration will easily facilitate a full history and accountability of activities performed to realise user needs.

A great strength of agile houses is their focus on a wide range of tools that can be easily integrated to reduce the scope of human error. The integration of requirement repositories, coding editors, version control tools, audit trails, automatic build tools, automated deployment tools, automated test execution tools, all ensure instant traceability and accountability of activities performed.

Ensure to check that the tools have been qualified for their intended use. Given the fluid nature of agile activities, where the process can change, the (re) qualification of tools may be omitted. Ensure to check that personnel have been trained on the effective use of tools.

## REPORTING (DOCUMENTATION)

Agile focuses on instant information. The use of various, interconnected tools that house the information can ensure full traceability and accountability of actions. This relegates the need for erroneous, documentation. For example, agile houses will display real time dashboards for constant feedback and perform demonstrations of working software to establish completion of User Stories on a regular basis.

Ensure to check that the acceptance of the iteration has been captured (e.g. by an acceptance certificate), by the stakeholders as it is unlikely that a report has been produced to confirm what was agreed to be delivered in a plan has been met. Check if there is an over-arching plan.

The use of a summary report may not be afforded much value by the agile practitioner given the use of dashboards and demonstrations. By the time a report is completed, the following iteration is well under way. On the other hand, the use of User Story demonstrations is not practical for the regulated customer.

To meet documentation needs from regulatory customers, agile houses will often produce the documentation that the regulated user recognises, (specifications, plans, trace matrices, reports), at the end of the various iterations when the software is release ready. This practice can often result in a 2-tier approach; one approach is actively managed daily via tools whereas the subsequent approach is intermittently produced after a release is made by the agile team.

A 2-tier approach can easily lead to problems. For example:

- The author of the ‘traditional’ documents may have limited engagement of the agile activities and have limited support from the agile team
- The plan has limited detail and is written after the event – risk analysis is limited
- User Stories may be rewritten into traditional requirement statements with missing details
- New requirement identifiers are provided to provide a sequential flow within the document and compensate for dropped User Stories
- Traceability is only between the rewritten requirements and their ‘acceptance’ tests and may be missing within the various tools due to the re-identifiers and restructured requirements.

The result is that erroneous documentation that fails to demonstrate the quality activities is captured within the tools during various agile activities. When asked about such a 2-tier approach, agile houses would state that this is what is being asked for by CS auditors.

Ensure to check documented evidence produced against the real time evidence captured in the tools to verify completeness of information.



## SUMMARY

Well considered agile practices can result in enhanced quality outcomes, such as the use of interconnected tools that provide instant traceability and accountability.

A lack of adequate focus on the practice of formal risk management, design and corresponding testing is a risk that doesn't appear to be considered by some agile operators. By overly focusing on narrow areas or negating the benefits of wider quality activities (e.g. design), then hidden issues will manifest themselves into serious issues in the future, (e.g. scalability).

The prevalence of security issues/defects (92% of reported vulnerabilities are located within application<sup>9</sup>) in computerised systems reflects negated design. A formal risk-based approach can provide justification for the scope of these quality activities.

Documentation is defined as 'evidence'<sup>10</sup>. Documentation was typically captured via software tools such as a word processor. The agile practitioner sees this format of 'documentation' as lethargic and erroneous. This format of documentation is a major source of the quality ills of software development (one of the reasons for agile). Agile has chosen workflow-based tools (that facilitates collaborative information gathering), to capture the documentation which enhances accountability and traceability. Given that the correct use of these tools can enrich quality, then this is an improvement over 'traditional' documentation.

The act of extracting the output of these tools to manually backfill legacy documentation templates appears counter intuitive to the gains made by agile practices. It is unproductive given that the 'evidence' already exists.

The regulatory customer requires oversight of the agile practise and perhaps the agile provider needs to consider other methods of facilitating this, such as building reports, summaries and so forth directly from the tools.

The use of checkpoints or gates during key stages of the approach can help identify (via risk) the scope of activities and subsequently ensure the formal capture of the 'documentation' as it occurs. If this documentation is being captured albeit in a different tool, then perhaps the regulated industry may need to look at things from the other side of the fence to establish the level of quality built in.

Finally, agile can be defined as having flexibility to react to change within the development process. Given that the activities of analysis, design, implementation, verification, validation, deployment and maintenance still need to be performed, there is no guarantee that agile is faster than any other approach.

## ABOUT THE DIGIT AGILE WORKING GROUP

The DIGIT Agile Working Group will provide articles that will seek to explain agile activities, identify their strengths and weaknesses, provide analysis on some of the key challenges, (e.g. documentation) and provide advice on applying agile within the regulated industry.

## REFERENCES

1. An Introduction to Agile Methodologies, Quasar 139, April 2017
2. Applied Software Measurement, Capers Jones, McGraw Hill, 2008
3. EudraLex Annex 11, 2011
4. Agile Manifesto, [www.agilemanifesto.org](http://www.agilemanifesto.org)
5. DIGIT Technical Dictionary: Common terms used by Agile based Suppliers. [www.therqa.com/good-practices/digit/agile/](http://www.therqa.com/good-practices/digit/agile/)
6. Empirical Findings in Agile Methods, Boehm et al, 2002, ACM
7. Code Complete 2, A Practical Handbook of Software Construction, McConnell, 2004, Microsoft Press
8. ISO/IEC/IEEE 29119-1: Concepts & Definitions
9. National Vulnerability Database, National Institute for Standards and Technology (NIST), 2011
10. Heinemann English Dictionary, 1985

## PROFILES

Richie earned a BS in Biology (Bethany College, West Virginia) and MS degree in Toxicology (University of Cincinnati College of Medicine, Ohio). He has worked for The Standard Oil Co., Gulf Oil Co. and Sherex Chemical Co. prior to Procter & Gamble and organised, trained and conducted GLP and GCP audits and compliance activities. Currently he is the Regulated System Compliance Leader for P&G and is a co-founder of the Quality Assurance Roundtable – predecessor to SQA – where he was elected president in 1990. He is a member of the Beyond Compliance Specialty Section, Computer Validation Initiative Committee and Programme Committee. Richie is also a member of the RQA's DIGIT Committee and DIA's the Information Quality, Compliance & Technology Community. RQA appointed Richie as a Fellow in 2014.

Barry is a Principal Consultant for Empowerment Quality Engineering, providing Software Lifecycle (SLC) improvements, validation and audits. He leads computer systems compliance and supplier audits globally. He has an MSc in Computing Systems and worked as a software engineer from 1997, performing every role and working in every phase of the SLC, (traditional and Agile). Barry has used multiple programming and database technologies to write and test software and has conducted complex, technical testing; from unit through to performance, security and disaster recovery within core network telephony systems. He moved into the regulated industry in 2003 to take up management roles in CS QA, QC and validation. His process improvement activities resulted in risk-based SLC systems, on time delivery and right first time validation. His experiences provide a unique grasp of technical, quality and people problems associated with delivering robust and compliant software. Barry is a member of the DIGIT Committee, a member of the ISPE Data Integrity Project team and a member of the DIA core IQCT team.